IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 5, 2021, accepted October 21, 2021, date of publication October 27, 2021, date of current version November 8, 2021.

*Digital Object Identifier 10.1109/ACCESS.2021.3123874*

# The HERA Methodology: Reconfigurable Logic in General-Purpose Computing

**PHILIPP HOLZINGER[1] AND MARC REICHENBACH[2], (Member, IEEE)**
[1]Chair of Computer Architecture, Friedrich-Alexander University Erlangen-Nürnberg, 91058 Erlangen, Germany
[2]Chair of Computer Engineering, Brandenburg University of Technology Cottbus-Senftenberg, 03046 Cottbus, Germany

Corresponding author: Philipp Holzinger (philipp.holzinger@fau.de)

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the Founding Program Microelectronic from Germany Innovation Driver through the Project Künstliche Intelligenz (KI)-Flex under Project 16ES1027.

**ABSTRACT** Due to the ongoing slowdown of Dennard scaling, heterogeneous hardware architectures are inevitable to meet the increasing demand for energy efficient systems. However, one of the most important aspects that shape today's computing landscape is the wide availability of software that can run on any system. Current applications that use accelerators, in contrast, are often especially tailored to a specific hardware setup and therefore not universally deployable. This is particularly true for reconfigurable logic as their internal structure requires the circuits and their integration to be designed as well. This makes them inherently difficult to use and therefore less accessible for a general audience. Nevertheless, their balance of flexibility and efficiency puts reconfigurable accelerators in a unique position between CPUs, GPUs, and ASICs. Therefore, one of the main challenges of future heterogeneous systems is to foster collaborative computing between these vastly different components while still being simple to use. Previous approaches mostly focused on subproblems instead of a holistic view of hardware and software in the context of commonplace usability. This paper analyzes the general demands on a reconfigurable platform and derives their requirements regarding accessibility and security. Hereby, we investigate several key features like hardware virtualization, system shared virtual memory, and the use of wide-spread programming paradigms. Then, we systematically build up such a platform based on the established ROCm GPU framework and its internal HSA standard. This new common HERA methodology is finally also demonstrated as a prototype.

**INDEX TERMS** Automatic synthesis, hardware/software interfaces, heterogeneous systems, reconfigurable hardware, virtual memory.

## I. INTRODUCTION

Computers are affecting almost all aspects of life nowadays. Hereby, the steady endeavor of society for life improvements drives innovation of new, increasingly demanding applications. This imposes a considerable burden on the hardware, which must be able to satisfy these desires. In particular the ongoing slowdown of Moore's law [1] and Dennard scaling [2] already cause current systems to struggle to provide sufficient computational power at high energy efficiency [3], [4]. To solve these problems, heterogeneous systems have become an integral part of modern computing in academia and industry [5]–[7]. As the challenges also increase, this paradigm will be inevitable across all classes of computers in the future [4], [8]. Although heterogeneous architectures offer great benefits, they are also much more difficult to handle. In contrast to common CPUs, there is a large variety of specialized accelerators each with different mechanisms to utilize them. These are often not only explicitly accounted for in application code but also hinder the direct interaction between them. Therefore, such executables are not universally deployable to any heterogeneous system. However, in particular the simplicity to write software that can run on practically any CPU lead to the wide availability of applications that sustainably shaped the computing landscape nowadays. This is therefore a major challenge for heterogeneous systems, which have yet to provide this ability.

One of the main pillars of such novel systems are *hardware reconfigurable architectures (RA)* like FPGAs [9] and CGRAs [10]. With their low-level and fine-grained reconfigurability, they take a unique spot in the spectrum of accelerators between GPUs and specialized ASICs [7], [11]–[13].

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

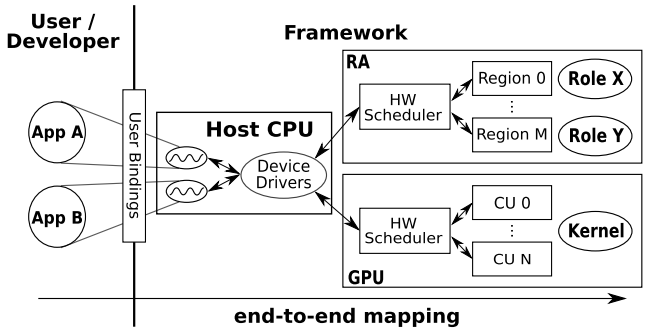



**FIGURE 1.** Schematic overview of "end-to-end" framework components.

While GPUs are flexible and achieve a high throughput, their power consumption is also very high [14]. On the other hand, dedicated ASICs can offer the same performance at a much higher efficiency, but it is impossible to integrate one for every task that might potentially be executed on a system [15]. This trade-off makes reconfigurable hardware ideal for the diverse and unpredictable set of applications running on modern general-purpose systems [16]–[18]. Although RAs, as a programmable solution, grant a great flexibility, their internal structure makes this process also inherently difficult. In addition to the overarching problems of heterogeneous systems, not only software but also the hardware itself must be designed by the users. This is especially complex due to the much larger semantic gap of mapping algorithms to logic circuits than to common programming paradigms. Although High-Level Synthesis (HLS) tools are continuously improving to generate synthesizable code from high-level languages, this is often not seen in a holistic way with software and other types of accelerators such that the integration is not trivial. For these reasons, RAs are not as accessible to a general audience as CPUs or GPUs.

As all types of compute units have their own strengths and drawbacks, it is inevitable to combine these vastly different components and foster collaborative work-sharing between them. Moreover, users should retain the ease of use that was established by the homogeneity of CPUs while gaining the new performance and energy benefits of accelerators. Therefore, the main challenge of future heterogeneous systems is the efficient integration of hardware and software into a complete platform that is still simple to use for developers and end users. This is especially important for RAs as their usage is inherently complex. Hereby, comprehensive frameworks are needed that encompass the whole range from user interaction with a high-level language to the final hardware. The general structure of such an *"end-to-end"* framework can be seen in Fig. 1. Entry points for developers are always the *user bindings* that provide a simple way to access accelerators. Then, *device drivers* and *hardware schedulers* manage the requested tasks and available compute resources. Finally, an *offload compiler* prepares kernel functions as hardware specific *ISA kernels* or *bitstream roles*. This structure can also be found in the two most important industry implementations for FPGAs, Intel's oneAPI [19] and Xilinx' and AMD's new joint technology preview of a converged FPGA and GPU platform [20]. However, only adhering to this architecture is not sufficient for a future-proof system as its underlying properties can still hinder collaborative work. Despite this importance of wide-spread acceptance, previous research focused only on singular aspects like HLS or scheduling instead of a comprehensive view.

This paper, in contrast, explores the design of future heterogeneous systems that encompass reconfigurable logic in a holistic way. First, we analyze the demands on such computers in a broader sense regarding aspects like accessibility and security to derive their requirements. Hereby, we consider several key features such as hardware virtualization, system shared virtual memory (SSVM), and the use of wide-spread programming paradigms. Second, we classify previous concepts in literature according to their suitability for these types of systems. Third, we present and assess the fundamental principles of the ROCm framework [21] and HSA standard [22] that are used in the upcoming production-ready joint Xilinx/AMD data center platform. Fourth, we extent this underlying baseline system with concepts to meet the initially derived stricter requirements for other computing domains. As the official Xilinx/AMD platform demonstrated only a small subset of the necessary features until now, it is even more important to further investigate this topic and show how future extensions and their implementation can look like. Hereby, we also use an FPGA as the underlying technology in the following, since they are readily available nowadays and already exhibit more and more CGRA features through the ongoing inclusion of more coarse-grained building blocks [10]. This further increases their efficiency while retaining the high flexibility. Furthermore, the high-level integration challenges of FPGAs and such CGRAs can be similarly addressed by frameworks. This new common methodology for reconfigurable logic in general-purpose systems presented in our paper is called HERA and demonstrated with a prototype.

The contributions of our paper can be summarized as:

- A new systematic requirement analysis of reconfigurable logic in general-purpose systems that allows developers to comprehend how accessibility manifests itself in software and hardware components and how these parts interact.
- A novel classification of current end-to-end frameworks regarding their limitations for widespread RA usage.
- A comprehensive security analysis for user logic, which gains low-level hardware access by RAs in general-purpose systems, as well as appropriate mitigations.
- A new comprehensive integration methodology that eases the adoption of RAs in new domains with stricter requirements where they are currently not widely used.
- A prototype of such a system based on the established ROCm framework and Xilinx FPGAs.

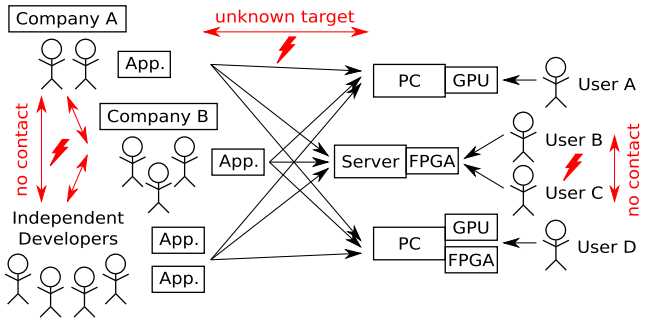



**FIGURE 2.** General life cycle of software. Applications are developed by a great variety of companies and individuals. These products are deployed to customer systems with equally diverse accelerator hardware. Hereby, the limited knowledge of all parties causes platform design restrictions. First, it is not clear if and what kinds of accelerators are present at the client. Second, it is not possible for all developers to coordinate their activity due to the sheer amount. Third, users also work concurrently and independently with these applications on their system.

The paper is structured as follows: Section II analyzes and classifies the demands on easily accessible reconfigurable platforms. These are compared with the capabilities of current literature in Section III. Building upon this, Section IV describes the general concept of such a suitable system that maps the complete flow from user application to task execution. Its security aspects are analyzed in Section V. A prototype realization of this with a description how it can be implemented is presented in Sections VI and VII. Finally, Section VIII concludes the paper.

## II. REQUIREMENT CLASSIFICATION

Although the performance and energy advantages are the main reason to use accelerators, their accessibility severely affects the actual adoption. The concrete requirements on frameworks depend on the domain they are used in. On the one hand there are *embedded systems* (ES) where a specific task runs in a predefined and closed environment. On the other hand a multitude of different applications are used in *general-purpose computing* (GPC) where hardware serves as a generic computing resource. This area of use can be further divided into large-scale *HPC and cloud computing* (HPCC) and small-scale *mid-range servers and personal computing* (MPC). The MPC domain in particular demands great flexibility as applications of many different parties are expected to concurrently run on a very diverse set of hardware. This most general case, as illustrated in Fig. 2, introduces further challenges, since all involved parties want to benefit from specialized hardware but only have limited knowledge about each other. Moreover, oftentimes the protection of intellectual property and less experienced developers must also be considered. Considering these domains, frameworks can facilitate the accessibility of accelerator hardware by providing several additional characteristics:

- **Familiarity**: Writing applications for an accelerator must be possible in languages and paradigms developers are already familiar with.
- **Interoperability**: Different accelerator classes must be addressable with the same source code. It should not be required to know in advance which ones are installed in a computer. However, all available ones should be used when it is beneficial.
- **Independence**: Applications from different companies have to be able to run concurrently on the same platform. It must not be necessary for developers and users to make prior arrangements. Therefore, each process can use the hardware as if it was the only one active.
- **Security**: Malicious accelerator kernels must not make the system vulnerable to information disclosure, tampering or similar threats.
- **Confidentiality**: Applications must be distributable as binary executable to protect a company's intellectual property on their source code.

These desired capabilities are fundamentally enabled by the underlying software and hardware stack. Therefore, we identify the necessary features that frameworks for reconfigurable computing should provide in general first. A summary of the following analysis can be found in Table 1. These are then used to classify and compare previous approaches in Section III.

The most prominent framework features are the **languages** and programming models that act as *user bindings* for the

**TABLE 1.** Classification of FPGA framework features.

| Feature | Subfeature | Option | Description |
|---|---|---|---|
| Language | - | * | supported programming languages |
| Program Features | SSVM | Y/N | system shared virtual memory |
| | CC | N | no cache coherency |
| | | OW | I/O cache coherency |
| | | TW | full cache coherency |
| Device Types | - | * | offloadable accelerator classes |
| Application Binary | - | AD | accelerator dependent executable |
| | | AI | accelerator independent executable |
| FPGA Backend | - | SA | standalone toolchain |
| | | * | required external backend |
| FPGA Virtualization | DPR | N | no dynamic reconfiguration |
| | | SP | single process dynamic reconfiguration |
| | | MP | multi process dynamic reconfiguration |
| | CTD | Y/N | concurrent task dispatch |
| | FIP | Y/N | fully independent process |
| Multi-User Capabilities | SU | N | no simultaneous usage |
| | | CB | simul. usage with common bitstream |
| | | FA | simul. usage when fairly allocated |
| | | Y | simul. usage without restrictions |
| | BS | Y/N | bitstream sharing |
| Security | BIC | Y/N | bus integrity checks |
| | AK | Y/N | abortable kernel |
| | ASI | N | no address space isolation |
| | | SP | single process address space isolation |
| | | MP | multi process address space isolation |
| | TM | Y/N | comprehensive threat model |

hardware. Developers usually prefer those which they are the most *familiar* with. This means offloading expressions must be as close to a standard model as possible. Nowadays, the most common open API for acceleration in heterogeneous systems is OpenCL. While it is well known among developers familiar with GPUs, its verbosity and separation in host and kernel code makes it less accessible for people with little accelerator experience. Therefore, coding styles that do not deviate too much from their usual CPU code simplify offloading considerably. These can be for example standardized directive or lambda based single source approaches like C++ ParallelSTL or OpenMP [23].

Traditional offloading techniques also suffer from the burden of manual memory management between host and device, which usually requires explicit data movement and visibility maintenance. However, this task is challenging for many commonly used data structures like the pointer-rich lists, trees, and graphs. Two **program features** that significantly reduce this complexity are **system shared virtual memory** (SSVM) and **cache coherency** (CC). With according hardware support, these concepts allow pointers to ordinarily allocated memory to be directly shared between CPU and accelerators. All address translations and data visibility updates are done by the hardware and its drivers. Therefore, the programmer is freed from maintaining and synchronizing multiple sets of the same data, which further simplifies development.

Although this paper focuses on FPGAs, it is important not to break compatibility to other **device types**. In the best case the same source code can be offloaded to all devices. This feature encourages accelerator usage because the performance evaluation of GPUs and FPGAs is not dependent on a much higher development effort. Furthermore, it enables writing applications without knowing the hardware of the target computer. For FPGAs, this mapping from high-level language to circuits is done with HLS. Here, it is in particular necessary that these tools are able to generate reasonable hardware without relying on FPGA specific pragmas, since they are not used for other devices.

This accelerator *interoperability* entails further requirements in case a software vendor wants to keep their application code *confidential*. At the present day, closed source software is still the norm to protect intellectual property. While the host ISA is usually stable and fixed for an application, this is not the case for the highly heterogeneous sector of accelerators. As a consequence, the **application binary** must contain the offloaded code exclusively in a vendor agnostic binary intermediate representation (AI).

This aspect also affects is the integration of the HLS tool. Solutions where such kernel binaries first have to be decompiled before passing them to an external framework like OpenCL cause additional overhead. First, it introduces unnecessary compilation steps by restoring a high-level representation. Second, even recent decompilers are prone to generate syntactically distorted or even semantically different sources for more complex code [24]. This can impair the results of an HLS tool. For this reason, it is favorable to eliminate this step and have a tightly integrated **FPGA backend** that can directly process binary kernels.

To eventually execute tasks on these generated accelerators, it is necessary to define a common platform. In GPC it must be assumed that applications of various developers or ISVs are nondeterministically started by the users of the system. A platform must therefore provide sufficiently **virtualized FPGA** resources to allow independent software development and execution when utilizing them. **Fully independent processes** (FIP) realize this on the OS and hardware side. This means that the FPGA is fully transparent and application code can treat device access as if it was the only one active. The platform is then responsible for the virtualization at runtime and guarantees that every task is eventually executed. In this regard it is necessary to distinguish between large scale cloud environments and local single node systems. Host servers of cloud platforms usually have multiple FPGA boards installed that can be passed to a VM as virtual resource. However, each one is usually exclusively assigned to a single tenant at a time [25]. In contrast, this cannot be assumed for workstations where similar to GPUs nowadays, only one or very few FPGAs are realistic. Therefore, a finer grained level of virtualization is necessary that allows sharing a single FPGA between multiple programs. In particular, a single process must not be able to exclusively lock a virtual FPGA resource. Otherwise, an application can indefinitely block the device for others although it might only rarely use it. For this reason, reconfigurable regions may only be implicitly assigned to a process when it actively runs kernels. The associated bitstreams are then automatically loaded by the runtime system on demand. To realize this FIP feature there are two fundamental requirements. First, **dynamic partial reconfiguration** (DPR) support is needed, especially if the FPGA is divided into several regions for concurrent execution of multiple kernels (MP). It enables exchanging functionality without affecting the management hardware or other accelerator kernels running on the same FPGA. Secondly, multiple processes must be able to **concurrently dispatch tasks** (CTD) to the same FPGA. Having these properties not only simplifies programming but also improves the overall FPGA utilization in the system as well.

A typical use case is a local server with a reconfigurable accelerator that is usually shared between multiple people. For this reason, a platform must also provide several **multi-user capabilities**. Most importantly, it must allow **simultaneous usage** (SU) of the accelerator to enable efficient working *independent* of others. This means the system is able to handle several requests at a time and frees the users from ongoing offline arrangements about its allocation. In particular, it is not sufficient if a platform forces users to initially agree on a common bitstream (CB), since this precludes those who have different needs from using it. Moreover, such a system must also not rely on a virtually endless resource pool like a cloud environment to gain this property (FA), since the available hardware is severely limited in MPC.

A **bitstream sharing** (BS) mechanism further improves the usability in this scenario. Since synthesis operations generally take a considerable amount of time and take up a lot of CPU resources, they should be reduced to a minimum. Here, it can often be assumed that programs are repeatedly executed or that multiple applications use the same library subroutines. Furthermore, these are also regularly run by multiple users on the same computer. Therefore, it is beneficial that only the first run performs synthesis and the bitstreams are then automatically provided to the other requesters.

When dealing with low-level hardware access for users, it is inevitable to also take **security** aspects into account. While this is highly design specific and explored in greater detail in Section V, several basic requirements need to be considered. First, **bus integrity checks** (BIC) must be performed to prevent malicious accelerator kernels from bringing down the whole system. Secondly, the administrator must be able to selectively **abort kernels** (AK) without affecting other processes running on the FPGA to prevent misuse. Thirdly, in any case sufficient **address space isolation** (ASI) between multiple processes (MP) is necessary to protect secrets in memory. Finally, to evaluate the security capabilities of a system and to find further design specific requirements, a comprehensive **threat model** (TM) that covers all conventional use cases must be provided.

## III. LITERATURE REVIEW

With the classification derived in Section II, it is now possible to evaluate prior research for their suitability. In this process, a distinction must be made between two different objectives. On the one hand, there are partial solutions that do not attempt to approach all requirements listed in Table 1. Instead, they only focus on specific subproblems like host-accelerator communication protocols [49]–[54]. These are not sufficient in themselves for widespread adoption in GPC due to the complexity to implement the remaining system. However, several partial solutions are still taken into account in the following as they constitute important contributions to their field. On the other hand, there are frameworks that cover the entire flow from high-level language to the final hardware and are thus *"end-to-end"* solutions. Table 2 classifies them to the best of our knowledge with regard to the requirements listed in Table 1. By examining the targeted domain and open source code of these tools, we came to the conclusion that factors that are not mentioned in their publications are highly likely not supported. Therefore, we also consider them to be missing ('N') in Table 2. In the following, these end-to-end approaches are now investigated in more detail, organized by features in groups of one or more columns.

The most basic capabilities of such a heterogeneous environment are *programming language* and *device type* support. In this context, mappings from a multitude of languages to all common kinds of computational hardware have been investigated before. Several toolchains use an FPGA offloading concept that is not inherently reflected by a widely known programming model. Instead, they require the developer to separately create all accelerators and their wrapper libraries [28], [44], [49], need additional architecture guidance [29], [32], or use dedicated languages [35], [46]. Further approaches improve the usability by leveraging the widespread OpenCL standard [26], [36]. Nevertheless, the simplest and therefore preferred ones are the previously presented single source solutions. These either directly implement known standards [19], [31], [33], [39], [40], [43] or require only trivial extensions to known languages [37], [48]. In particular, several toolchains have also demonstrated a unified programming view of CPUs, GPUs, and FPGAs [19], [37], [40], [47]. Therefore, the basic capabilities in this area have been investigated and the research focus is shifting to the efficient management of heterogeneous systems.

Although the basic language and hardware support is important, it does not imply all required characteristics of the underlying implementation. Yet these internal capabilities heavily influence the programming complexity and therefore the acceptance of FPGAs. This is most evident in the memory management between host and accelerator. Several platforms with *SSVM* and *CC* have been presented [19], [26], [36], [55], [56]. However, due to the complexity to fully implement them, explicit memory maintenance remains the default nowadays.

Another important property is the form in which applications can be deployed. In its simplest form the kernels can be distributed as plain source code. This has the advantage that the offloaded sections can be compiled to any supported device after distribution. However, many companies insist on a binary-only distribution to protect their IP. In this case device specific bitstreams can be created and embedded into the distributed binary. This approach requires that the target system is known in advance or that bitstreams for all possible devices must be generated. While this can be acceptable in HPCC, the expected wide variety of system compositions in MPC makes this infeasible. Despite these disadvantages, most existing frameworks rely on one of them [19], [26], [28], [33], [35], [36], [45], [47]. In contrast, a true GPC approach must be able to satisfy both requirements. To solve this issue, several tool flows have been presented that decompile CPU assembler instructions [57], [58] and then use a third-party OpenCL *backend* to target the accelerator. However, this already highly optimized code for a specific CPU often leads to worse results for completely different accelerator architectures. Therefore, a *device independent binary intermediate representation* is better suited for these use cases [22], [59]. For FPGAs a system based on Java bytecode has been demonstrated, but it also relies on a decompilation step [37]. Therefore, we use an own *backend* to directly map from intermediate code to the target hardware.

Nevertheless, at some point this translation has to be performed. For FPGAs this step involves HLS, logic synthesis, and place and route. Especially the latter is very time and energy consuming and must therefore be carried out as rarely as possible. Previous works used shader [60] and bitstream [38] caches to save finalized accelerator kernels

**TABLE 2. Feature matrix of end-to-end FPGA frameworks (Legend in Table 1).**

| Tool | Languages | Device Types | FPGA Backend | Program Features [SSVM,CC] | Application Binary | FPGA Virtualization [DPR,CTD,FIP] | Multi-User Capabilities [SU,BS] | Security [BIC,AK,ASI,TM] |
|---|---|---|---|---|---|---|---|---|
| Xilinx Vitis [26] | C/C++, OpenCL | CPU, FPGA | SA | Y,OW (MPSoC)<br>N,N (PCIe) | AD | MP,Y,N | CB,N | N,N,MP,N (MPSoC)<br>Y,N,SP,N (PCIe) |
| VINEYARD [27], [28] | Library | GPU, FPGA | OpenCL | N,N | AD | SP,Y,N | CB,N | Y,N,SP,N |
| CAOS [29] | C/C++, OpenCL | CPU, FPGA | C or OpenCL | N,N | AD | SP,Y,N | CB,N | Y,N,SP,N |
| TaPaSCo [30] | OpenMP [31] | FPGA | C++ | N,OW (MPSoC)<br>N,N (PCIe) | AD | N,N,N | N,N | N,N,N,N (MPSoC)<br>N,N,SP,N (PCIe) |
| FASTER [32] | OpenMP + XML | CPU, FPGA | C | N,N | AD | SP,Y,N | N,N | N,N,SP,N |
| IBM cloudFPGA [33], [34] | C/C++ (+MPI) | FPGA | C/C++ | N,N | AD | MP,Y,N | FA,N | N,N,MP,N |
| Asiatici [35] | DSL | FPGA | C/C++ | N,N | AD | MP,Y,Y | Y,N | Y,N,MP,N |
| Intel OCL SDK [36] | OpenCL | FPGA | SA | Y,OW (MPSoC)<br>N,N (PCIe) | AD | MP,N,N | N,N | N,N,MP,N (MPSoC)<br>N,N,SP,N (PCIe) |
| Intel OneAPI [19] | DPC++, OpenMP | CPU, GPU, FPGA | SA | Y,OW (MPSoC)<br>N,N (PCIe) | AD | MP,N,N | N,N | N,N,MP,N (MPSoC)<br>N,N,SP,N (PCIe) |
| TornadoVM [37] | Java | CPU, GPU, FPGA [38] | OpenCL | N,N | AI | SP,N,N | N,N | N,N,SP,N |
| ORKA-HPC [39] | OpenMP | CPU, FPGA | LLVM-IR | N,N | AD | SP,N,N | N,N | N,N,SP,N |
| OpenARC [40] | OpenMP, OpenACC | CPU, GPU, FPGA [41] | OpenCL | N,N | AD | SP,N,N | N,N | N,N,SP,N |
| LegUp [42] | C, OpenMP [43] | CPU, FPGA | SA | N,OW | AD | N,N,N | N,N | N,N,N,N |
| Maxeler MaxCompiler [44], [45] | Host: C/C++, Python, Matlab, R<br>Kernel: MaxJ | DFE (FPGA) | SA | N,N | AD | SP,Y,N | CB,N | Y,N,SP,N |
| IBM Liquid Metal [46], [47] | Lime | CPU, GPU, FPGA | SA | N,N | AD | N,N,N | N,N | N,N,N,N |
| HERA Methodology | OpenCL, OpenMP, C++ [48] | GPU [21], FPGA | SA | Y,OW | AI | MP,Y,Y | Y,Y | Y,Y,MP,Y |

for repeated use, but these implementations were only suited for a single application. However, there are common use cases where multiple users use the same applications and basic libraries on a shared computer. Therefore, we present a methodology in Section IV to provide transparent synthesis capabilities and effectively *share bitstreams* between multiple users.

Finally, these applications are executed on the target machine. In a typical use case a user is simultaneously working with several programs. It is also common that multiple users work on the same machine. In this scenario a GPC platform must be able to facilitate FPGA acceleration for all concurrent tasks without expecting any prearrangement. Therefore, sufficient *virtualization* of FPGA logic is necessary. Most research in this area has been conducted for cloud environments. Here, *shell-role architectures* [61] with *dynamic partial reconfiguration* are predominantly used [62]. Hereby, *shell* denotes the immutable part of the FPGA logic while a *role* describes a concrete instance of a reconfigurable region. However, there are different approaches how *role* functions are handled. In most approaches, virtual FPGA regions are exclusively reserved for tenants and are not reassigned as long as the user is paying [26], [33], [63]–[65]. These solutions heavily rely on scale out effects of their platform, since a user is theoretically able to claim and lock all regions but usually lacks financial means. However, in smaller scale MPC systems only a very limited number of regions are available. Moreover, there is no incentive not to occupy FPGA resources, since there are no charges in these use cases. This means that these approaches cannot guarantee that applications are accelerated even when the FPGA does not actually execute tasks. Other approaches try to solve this problem by providing a predefined accelerator pool whose functions are only executed on demand [27], [49], [66]. Even though this improves FPGA resource sharing, they are no longer freely configurable for a user without administrative privileges. Therefore, these approaches partially give up the great advantage of FPGA reconfigurability. Nevertheless, a methodology with both of these capabilities has been presented by Asiatici *et al.* [35]. Here, a local processor running an RTOS is responsible for scheduling all submitted FPGA tasks. However, their results also show a high latency and runtime overhead for many short accelerator invocations. This shows that efficient FPGA *virtualization* is still a key problem in GPC.

When integrating an FPGA into an MPC system, it is essential that such a platform is *secure* and does not put the traditional components at risk. Especially the low level hardware access that is provided to ordinary users poses a great risk for the remaining system. Previous research focused on several specific issues like secure computations in roles [67], mandatory access controls [68], ASI [35], BIC [26], [35] or

side channel leakage of data [69]–[71]. However, an exhaustive exploration of threats to an end-to-end FPGA platform itself has not been presented yet. Therefore, this is further analyzed in Section V to develop a secure concept.

It can be seen that FPGA frameworks have to provide a wide variety of capabilities to foster accessibility, but previous ones typically do not do so for all of them. With the new joint technology preview of Xilinx and AMD [20], there is a second official accelerator class overarching framework next to Intel's oneAPI implementation now [19]. In contrast to the latter, it is based on the already established open source ROCm ecosystem for GPUs [21]. However, only a very limited set of features has been presented in the FPGA demonstrator so far. These are basic task dispatch to a fixed accelerator without HLS and dynamic reconfigurability, SSVM over PCIe, and direct event signaling between an FPGA and a GPU. Although these capabilities mark a promising first step, they are also not sufficient for the requirements we derived in Section II.

Nevertheless, when having a closer look at the ROCm framework, it can be seen that it in turn implements the open HSA Foundation standard [22]. Multiple related works based on this standard have been previously presented, including mappings of OpenMP (GCC), OpenCL [21], C++ ParallelSTL [48], and Python [72] to address GPUs. Furthermore, [54] and [73] have shown basic hardware and runtime components to integrate FPGA accelerators. Additionally, [74] demonstrated that these cores can also be automatically generated from the HSA intermediate language (HSAIL) with HLS. Although all these works are based on an open standard, it does not guarantee a system to meet requirements derived in Section II as not all relevant properties are well defined. In particular, FPGA related HSA literature primarily focused on bare-metal embedded systems. As such, OS integration, virtualization, SSVM, and security related issues have not been considered, which makes these solutions not suitable for GPC systems. Nevertheless, due to the standardization, they can be used as a basis to build up a system that is governed by the same principles as the announced Xilinx/AMD platform. Therefore, in the following we demonstrate how such a framework can be designed and extended to meet the defined requirements.

## IV. PLATFORM DESIGN

In general, "end-to-end" hardware platform frameworks consist of five basic layers. These are *application software*, *language runtimes*, *user space library*, *kernel drivers*, and *hardware*. As such, the ROCm ecosystem [21] also contains the components visualized in Fig. 3 that implement them. Since Xilinx' and AMD's FPGA technology preview [20] does not have all the needed capabilities to meet the requirements derived in Section II, we also extended ROCm to dive deeper into the underlying protocols. Consequently, we adhere to its fundamental mechanisms in the following and extend it only where new FPGA specific capabilities are needed. The implementation is explained in greater detail in Section VI on the example of the Xilinx ZynqMP and Virtex platforms.

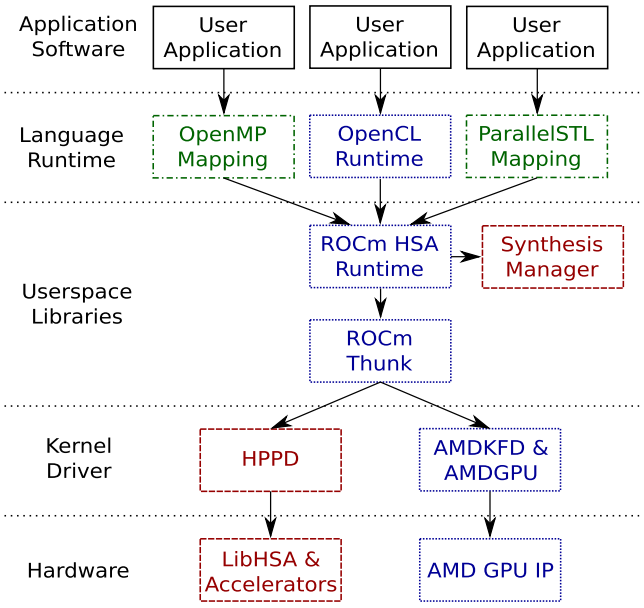



**FIGURE 3.** Software stack to interact with the underlying hardware. ROCm is extended to create a common framework for GPUs and FPGAs. Dotted, blue components are found in the classical ROCm GPU flow. Dashed, green ones are external extensions that are possible due to a common interface. Dotted and dashed, red ones constitute major extensions for FPGAs. For the actual kernel dispatch the *kernel driver* and *user space libraries* layers are bypassed. This is possible, since the content of the memory mapped execution queues can be directly interpreted by the hardware.

```
// prepare arrays on the host

#pragma omp target teams distribute parallel for
for (int i=0; i<length; ++i){ // host ctrl bounds
  a[i] = b[i] + c[i]; // accelerator kernel
}

// process results on the host or an accelerator
```

**Listing 1.** Example offload annotation with OpenMP as used in GCC.

Starting from the uppermost layer, the *application software* interacts with the hardware primarily through dedicated *user bindings*. Hereby, it is always possible to employ specialized higher level frameworks like BLAS or TensorFlow. However, internally these usually make use of standard programming language constructs and paradigms that can express parallelism. Listing 1 illustrates this with OpenMP where the loop is marked for accelerator offloading with a specific pragma directive. These common methods guarantee easy accessibility and a uniform view of the heterogeneous hardware for interoperability between different device classes.

It is then the responsibility of the underlying *language runtimes* to map these function calls and annotations to lower level hardware API calls. In the course of this, the compiler splits the software into the host code that is executed on the CPU and device code for the accelerators. For this purpose, the necessary dispatch information like array pointers or

dimensions is extracted from the language construct (e.g. the bounds of a pragma annotated loop). Afterwards, the respective code section is replaced with device API calls that offload the task. Similarly, the actual kernel is compiled into a binary intermediate representation and embedded into the final executable. This independence of the language and the device through an abstraction layer is crucial to easily retarget code sections to other accelerators even after software deployment, since it can be used for dynamic linking. While both ROCm and oneAPI have a device agnostic API for the host side calls, neither currently support independent binary kernels for FPGAs. Therefore, we additionally extend ROCm with this feature, which allows us to also demonstrate the existing OpenMP (GCC 8) and C++ PSTL [48] frontends.

This device abstraction is implemented with the standardized HSA API [22] in the *user space library* layer. Internally, ROCm splits this layer into *thunk* (ROCT) and *runtime* (ROCR). The thunk takes care of driver specifics by issuing IOCTLs and managing sysfs entries. The overlying runtime then provides the standardized interface that applications are linked against. Since this layer implements the device agnostic API, it must also be able to perform the final compilation step from a device independent binary kernel to the final executable. Therefore, we extended the interface by the optional HSA finalization feature. However, in contrast to GPUs, FPGAs incur very long synthesis times to generate the final executable bitstream. For this reason, we additionally integrate a *synthesis manager* (SSM) as a system daemon. It acts as a centralized *offload compiler* that automatically performs HLS and logic synthesis and caches kernels across multiple users of a system. This sharing can be especially effective for end-user systems, since multiple users often need the same applications and this even more than once.

The next layer below consists of the operating system *kernel drivers*. Their responsibility is the communication and management of multiple processes with the *hardware*. In particular, creation and destruction of dispatch queues, signals, and page tables need to be synchronized. The mainline Linux kernel currently implements this functionality for GPUs in AMDGPU and AMDKFD. Due to the common HSA standard, an FPGA driver can use the same basic Linux user API as ROCm GPUs. However, in contrast to them, FPGAs are not intrinsically able to execute any task, but prior reconfiguration is needed. Therefore, we extend the OS with an additional mechanism in our new HPPD driver that manages this process. It shifts the hardware resource oversight from user to kernel such that multiple users can concurrently use the device and still freely reprogram it. For this purpose, *language runtimes* can register kernels synthesized by the SSM that they want to subsequently use. The driver is then capable of automatically managing dynamic reconfiguration depending on the actually dispatched and running tasks.

The lowest layer is the actual *hardware*. It is split into an immutable shell and several freely runtime reconfigurable areas for role accelerators. A core part of the shell is a *System Manager* that organizes chip and interface status between

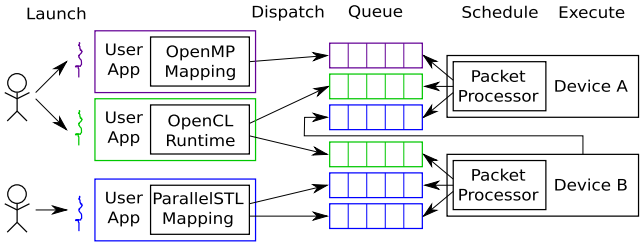



**FIGURE 4. Overview of the task dispatch mechanism. Applications create as many AQL dispatch queues as needed via the HSA Runtime function calls and the driver. Tasks in the form of AQL packets can then be directly written to memory from any device. The packet processor parses and schedules the incoming kernels to its associated accelerators.**

the *driver* and the hardware. However, the main point of interaction between CPU and FPGA is a *hardware scheduler* that is also located in the shell. In contrast to other standards, the HSA model specifies a dispatch mechanism based on memory mapped queues and event signals that are directly processed by the accelerator hardware. An overview can be seen in Fig. 4. Hereby, the structure of such a *dispatch queue* is strictly defined as packets with a specific format (AQL) in memory. This communication happens directly between a *language runtime* and the hardware while completely bypassing the *driver*. The necessary logic to process this protocol on the hardware level is integrated into the *hardware scheduler*, which is thus called *Packet Processor*. It schedules and supervises all tasks from all processes to this accelerator and their associated signals. With such a component, users gain very low-level access to a device. This is favorable for FPGAs as HDL developers usually work with similarly basic operations. Thus, they are able to directly read and modify these memory regions to dispatch own tasks from one accelerator to another. This mechanism significantly reduces the dispatch latency in general, since CPU and operating system are not involved in a kernel launch. Furthermore, the same procedure can also be utilized to grant FPGAs simple access to system calls. Hereby, an application specific CPU host thread can also act a Packet Processor that is able to handle e.g. `malloc` or `free` requests submitted by accelerators [75].

## V. SECURITY CONSIDERATIONS

A platform as described not only gives users low level access to the hardware but even permits them to arbitrarily modify the actual accelerator circuitries. These abilities impose a considerable risk to the integrity of the system as a whole, since exploits are potentially able to bypass protection rings. This makes an FPGA a worthwhile auxiliary for attacks. Therefore, creating a threat model is inevitable to quantify the risks caused by an FPGA integrated in this way. Based on this analysis, suitable mitigations to secure the system can be developed beforehand.

### A. FPGA THREAT MODEL

A GPC system generally has two groups of people. On the one hand, there are unprivileged users like developers or

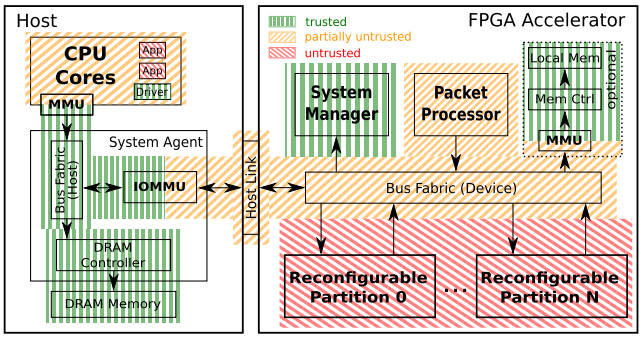



**FIGURE 5.** Trust boundaries of the minimal basic hardware components of our GPC system. Green regions cannot perform any malicious requests by themselves and are therefore trusted. Orange components need to process potentially insecure requests, but there are restrictions on the type of operations that can be performed. In contrast, adversaries can arbitrarily modify the hardware and software of red areas.

end-users that work with the system. On the other hand, super users (SU) have privileged access for administrative purposes. Depending on the domain of use these groups can also overlap. This introduces two main perspectives on security. First, users that want to use the system but do not trust the SU itself, e.g. when sensitive data is processed in cloud environments. In this case, it must be assumed that all stored data can be read or modified due to the high privileges SUs need for administration. With them, they are able to modify the clock, capture the content of memory cells with a configuration readback, or even change them. For this reason, it is inevitable to always encrypt data before storing it in any kind of memory that is contiguous enough to be able retrieve information (e.g. DRAM or SRAM). Second, the SUs who want to provide a secure environment and therefore generally distrust their users. In this case, it is assumed that attackers in a user role try to take advantage of the FPGA platform to cause damage. This scenario is relevant for all kinds of reconfigurable systems. However, in contrast to the first perspective, the adversary has lower privileges here. This implies a broad set of possible counter-measures that are explored in the following.

To analyze the attack vectors, it is first necessary to identify which parts can be trusted and where the boundaries of their influence are. A segmentation of the hardware in this regard is depicted in Fig. 5. Core of this system are the *reconfigurable partitions*, which contain the kernel accelerators that are exchanged at runtime. As part of the concept, a user may arbitrarily define the content of these regions. Adversaries will use this ability to their advantage. Therefore, this logic is generally not to be trusted. In contrast, the fixed shell is only supposed to be modified by the trusted SU. It contains *System Manager*, *Packet Processor* and its firmware, as well as a *bus fabric*. This restriction limits the degree of influence an attacker has over these components. However, a user space program is still able to directly affect their behavior. This is particularly evident in the data structures used for direct communication with the hardware that are located in user space, as described in Section IV. Therefore, the *Packet Processor* firmware must handle all memory accesses necessary for protocol parsing as well as the read values as untrusted. The same must be assumed for memory transactions originating from the accelerator cores, which are also routed via the shell and CPU bus fabrics. However, the *(IO)MMUs* typically also provide protection. Therefore, the bus fabric is divided into a partially untrusted section before *(device)* and a trusted one after them *(host)*. Nevertheless, this assumes that all *process address space identifiers* (PASID) are correctly set up and cannot be altered by the user. While PASIDs are initially assigned by the trusted driver, they need to be updated when the process ownership of a role changes (e.g. when scheduling a new task). This is done by the *System Manager* on behalf of the *Packet Processor*. As a non-programmable component, it can physically only perform a few specific actions that are requested by its bus masters. Therefore, it is trusted as long as the tasks can only originate from a trusted environment.

Based on these trust boundaries, it is possible to create a full threat model. In this process we use the established *STRIDE model* [76] to systematically investigate the attack vectors. It categorizes security threats in six classes that are separately discussed in the following. Each of them is further broken down with the likewise established *attack trees* [77] depicted in the Appendix (Fig. 10) to analyze their requirements. Hereby, we only consider attacks that make use of the FPGA platform, since they extend the possibilities an adversary has compared to a traditional system. The resulting attacks for each goal are then assessed in Section V-B.

**Spoofing** is the first threat category in the STRIDE model and describes situations in which an adversary successfully identifies as another user. In the GPC platform use case this is mostly used as a mean to a privilege escalation. Therefore, we consider it a secondary goal as a step to achieve other ones. Its most important application in our approach is the spoofing of the PASID to authenticate as another process and bypass address space protection checks at an IOMMU. There are three options to perform this attack (Fig. 10h). First, some systems might allow an untrusted *reconfigurable partition* to directly set the respective selection bits at the *bus fabric*. While HLS code generated by the platform itself will not abuse this, the same is not true for specifically prepared malicious bitstreams. Secondly, an adversary might also be able to send own commands to the *System Manager*. As explained before, its status as a trusted component depends on the masters that can send requests. If a user provided accelerator is a bus master or if driver or *Packet Processor* software get compromised, this assumption does not hold any more. An attacker would then be able to set the PASID at will for all ordinary accelerators. However, this indirection is not necessary if an adversary is even able to control the shell hardware itself. This ability is also a common secondary goal, which can be used for several types of attacks. There are two ways to achieve it (Fig. 10k). The direct approach tries to load an own full bitstream, which requires SU rights. In contrast, the indirect method uses partial role bitstreams

that are specifically prepared to also reconfigure circuits outside of their *reconfigurable partitions*. These can be loaded by regular users. In both cases the images incorporate the necessary logic to carry out the attack. In this third alternative to spoof the PASID, it is used to directly modify the pins that drive the IOMMU.

**Tampering** the platform is not only useful for spoofing but is often even the primary goal of an adversary (Fig. 10c). In general, this attack class tries to undermine the integrity of the system. On the one hand it is performed to favorably affect the behavior of operations. This can be seen in the previously presented modification of the shell hardware, which severely compromises the trust in the FPGA platform. Similar effects can be achieved by tampering with the SSM directory (Fig. 10j) that synthesizes and caches bitstreams on disk. When these stored roles are modified, it leaves unsuspecting users to execute wrong kernels or experience delays due to forced resyntheses. On the other hand tampering attacks are also used to falsify or delete data. The main way to exploit the FPGA for this goal is through direct memory access to a foreign address space (Fig. 10e). With PASID spoofing, the first approach has already been discussed. Using this method, it is possible to overwrite the user space data of every process that has been bound to the IOMMU. A further escalation can be achieved by accessing the operating system memory (Fig. 10f). Its associated page table entries are usually mapped into the address space of every user process to allow faster system calls. Hereby, they are protected to be neither readable nor writable from user mode. However, the privilege level of a transaction is determined by the physical bus signals. If an untrusted role or tampered shell hardware can modify these wires, an adversary is able to additionally perform regular transactions to kernel memory. Having this possibility, it can be further exploited to allow read and write accesses to arbitrary memory locations (Fig. 10g). This can be done directly by modifying the page tables or indirectly by escalating the privileges of the host program. Another option to achieve this goal is present if an accelerator is not subject to address translations and protection. This is the case if no IOMMU is available in the system or if it is deactivated. Then direct physical addressing can be used to access any memory location. In any case a regular user would be able to severely tamper with data stored on the device by using only regular memory accesses.

**Repudiation** is a frequently used means to deny committing attacks. In general it is used to reject that an action has been performed by a person. In some cases this class brings an immediate gain, e.g. when maliciously disputing purchases. However, in the context of an FPGA platform it is only of lesser concern, since it does not impede security directly. Nevertheless, having a non-repudiable system can help an administrator to investigate other attacks. In the proposed approach FPGA usage is always associated with a user process. Therefore, it is possible to use logging techniques as long as an attacker does not gain root privileges.

**Information disclosure** breaches the confidentiality of the system, since it allows attackers to gain access to data for that they have no permissions (Fig. 10a). The most effective method is the previously discussed direct access to foreign address spaces from a user process (Fig. 10e). With these exploits an attacker can obtain data of any running process including sensitive cryptographic keys of the operating system. Another, although less effective, approach to collect data is to intercept the *bus fabric* that is shared between all *reconfigurable partitions*. With a modification of this bus, all values read and written by accelerator cores as well as their memory addresses can be logged. However, this requires an attacker to be able to tamper the shell (Fig. 10k). Alternatively, there is also a less intrusive variant that can be carried out with only a user-defined role. It exploits that many interconnects use a shared crossbar to reduce logic. This causes signals to be readable for all masters although they are only valid for one. An adversary is therefore able to record loaded data but not the addresses it is associated with. With both variants that intercept the *bus fabric*, the obtainable data is limited to the working set that is actively used by the accelerators. In contrast to the first method, the main memory is not directly readable. A further restriction is imposed by the third approach, which tries to extract remaining data of other roles. Here, an attacker is only able to see values that a previous kernel execution stored in the same *reconfigurable partition*. This attack imposes the fundamental requirement that neither the previous kernel nor the platform clears the memory cell contents at a context switch. Normally these cells are automatically initialized at reconfiguration when a new role instantiates them. However, it has been shown that partial bitstreams can be tampered in a way that BRAM content is not zeroed [78]. This causes data to remain in the cells. The same is true when reconfiguration is not needed because the current and previous tasks use the same kernel. In both cases a malicious role can read the retained data. Contrary to the approaches presented so far, which directly access foreign data, there are also side-channel attacks to indirectly obtain sensitive information. The first option is the class of power attacks [69]. These usually monitor the power consumption to analyze which instructions are executed and which data is processed by other processes. Traditionally, physical access is needed for these measurements. However, it has recently been shown that they can also be approximated by circuits that can be instantiated in a role [70]. This possibility opens new realistic ways for attackers when FPGAs are integrated into future GPC systems. Similarly cache attacks can also be simplified. These bring a shared cache in a known state first and then try to determine intermediate changes of other processes by comparing access time differences. An FPGA makes these measurements more reliable due to the ability to instantiate precise timers and directly interact with the coherency protocol. Therefore, these side-channel attacks actively leverage FPGA capabilities to gain access to secret information.

**Denial of service** (DoS) attacks, by contrast, have a different objective (Fig. 10b). This class of threats tries to reduce the availability of the system for its users, e.g. by shutting it down or degrading its performance. There are four levels of severity. The first and lowest one causes worse performance and overall responsiveness of applications. Similar to pure CPU solutions, the FPGA platform can also be used to excessively increase the load with pointless computations. An adversary can provoke this in multiple ways. For example by starting not needed synthesis jobs at the SSM, overloading the *DRAM controller* and *bus fabric* with memory transactions, or spawning useless FPGA kernels. In its worst form these accelerator tasks contain endless loops. If the system is not capable of preempting or terminating them, the FPGA would have to be reset completely. The second severity level causes the platform to be unreliable. This can happen if an adversary tampers the metadata of the SSM directory (Fig. 10j). In that case bitstream cache lookups might be wrong, not working at all or not able to synthesize new requests. This leaves the platform incapable of processing tasks that are not already registered for execution at the driver. In contrast to the previous two severities, the third level is able to bring down the complete system. There are two options to realize this threat. First, an adversary can try to deadlock the memory subsystem, which prevents further loads and stores. This method uses the fact that FPGAs allow users to directly manipulate bus signals. With this ability, attackers can issue transactions that undermine the integrity of the underlying protocol. These can be incomplete requests or handshakes, unsupported operation modes, or by deliberately inducing metastability. For the second option to crash the system it is not even necessary to directly interact with the shell. Here, an attacker utilizes synchronized pulses of ring oscillators (RO) to stress the power delivery network (PDN) and generate voltage emergencies [79]. This causes timing failures of circuitry in the same power domain outside the untrusted *reconfigurable partition*. However, this concept to influence the system on an electrical level can be even further extended to perform attacks with the highest severity. In this case, adversaries utilize the reconfigurability of FPGA routing to cause a short circuit and overload the PDN [80]. Therefore, this attack is able to permanently destroy the platform.

**Elevation of privilege** is the last threat class of the STRIDE model (Fig. 10d). An adversary's goal is to bypass the system authentications to gain access to resources that are restricted to users with higher privileges. One way to realize this has already been described with the attacks to gain access to protected OS memory from the reconfigurable logic (Fig. 10f). However, this ability can be further utilized to get system wide super user permissions. For that, an attacker needs to find and modify the credentials in the OS `task_struct` of the process that uses the accelerator. After signaling the modification to the host process, it directly possesses elevated privileges. An adversary is then able to open a root shell and perform any other attack.

### B. RISK ASSESSMENT AND MITIGATIONS

With the threat model established in Section V-A, it is now possible to design appropriate defense mechanisms. The basic assumption is that the adversary has no root privileges yet when using the FPGA platform to attack the system. Otherwise there would be more trivial options to inflict damage. It is then a primary responsibility of the FPGA platform to ensure that this assumption is still true when reconfigurable hardware is utilized. It can be seen that a major opportunity to elevate the privileges is by having direct access to the physical address space (Fig. 10d). This ability is highly critical, since it not only allows attackers to modify OS data structures but also tamper or disclose data of other users. Therefore, it is inevitable to perform address checks when reading or writing any memory. It can be realized through a dedicated *(IO-)MMU* for every storage region that blocks unauthorized requests.

However, this measure is only effective if correct authorization is ensured by the platform. For this purpose, it is necessary to enforce the trust boundaries established in Fig. 5. In particular, the boundary of the untrusted region must be protected, since an adversary is able to abuse unchecked transactions to also threaten the availability, integrity, and confidentiality of the system. Therefore, the platform must synchronize all incoming signals, zero data that is not destined for a *reconfigurable partition*, check protection bits and modes of transactions, and time out overdue ones. This has to be done by special hardware units in the shell that are called *System Guard* in the following. Furthermore, it is necessary that PASIDs cannot be set by user accelerators, but only by trusted components. For this purpose, an *Address Space Manager (ASM)* is inserted into the *bus fabric* in front of the FPGA (IO-)MMUs. This component is directly controlled by the *System Manager* and decodes bus master IDs to their corresponding PASIDs. Finally, it must be ensured that these configurations cannot be compromised by an attacker. Therefore, command and data busses of the fabric are physically split into separate interconnects, such that there are no data paths from *reconfigurable partitions* to the *System Manager*. The remaining masters are solely the FPGA platform driver and the *Packet Processor*. Both can only be altered with elevated privileges. Furthermore, a *Packet Processor* implementation has to strictly separate privileged command transactions and ordinary memory accesses made on behalf of user processes.

The measures presented so far protect shell and memory from being compromised by regular usage patterns. However, the previous analysis showed that users can still modify the shell with partial bitstreams that do not abide to the region boundaries (Fig. 10k). Therefore, it is inevitable to authenticate all roles by a trusted party before they are loaded (Fig. 10n). This task can be taken over by the proposed SSM because it is responsible for synthesizing all designs. It assigns every generated bitstream a *hash-based message authentication code (HMAC)* [81] that is used by the driver to validate authenticity and integrity of a role (Fig. 10l). This transmission can now be considered secure, since it can only

be circumvented in the unlikely cases that the administrator's secret key is revealed or if the HMAC algorithm itself is insecure (Fig. 10i). The architectural updates presented so far also prevent that an adversary can directly modify bitstreams in-memory after the driver verified them. However, it is still possible to interfere with the initial generation by undermining the SSM integrity (e.g. by tampering with scripts). To additionally mitigate this option it is necessary to protect these files from write access by ordinary users. Therefore, both SSM daemon and its cache are managed by a separate dedicated user account that is exclusively accessible by administrators. With these measures, the electrical and design rule checks of the SSM cannot be circumvented and authentication is enforced. This also effectively prohibits attacks that cause electrical shorts and mitigates attacks on the PDN by adding rules to detect ROs and delay-lines (DL) [69]–[71], [79].

The provisions presented so far prevent an adversary from reaching the most severe goals. However, there are several attacks that can only be partially mitigated by the FPGA platform. First, the mildest form of DoS attacks temporarily exhausts common resources like *reconfigurable partitions* or the *bus fabric* (Fig. 10b). The capability to preempt running kernels can improve the situation, but is expensive on current FPGA hardware and does not prevent an adversary from creating a high number of tasks. Moreover, most of the time these attacks are not distinguishable from legitimate high load. Therefore, the best option is to give administrators the ability to selectively kill processes running on the FPGA without affecting others. To further reduce the probability of misuse, it also sensible to restrict access to reconfigurable logic and SSM to a trusted user group (Fig. 10n). Administrators can then easily grant and revoke these permissions. Additionally, they should have the choice to disable the ability to synthesize HDL kernels, since several attacks require their greater expressiveness compared to high-level languages (Fig. 10m). Nevertheless, HDL based attacks are also mitigated by the other presented means.

Another attack concerns information leakage through data left behind in a *reconfigurable partition* (Fig. 10a). An FPGA platform is able to be immune to this attack by loading a clearing bitstream after every executed kernel. However, this is also an expensive operation on current FPGA hardware due to the relatively long reconfiguration times. Nevertheless, even if this is not done, an application can always eliminate this issue by simply erasing its data after execution. Furthermore, the SSM enforcement initializes all memory cells used by an attacker when reconfiguring a region, which clears all content. This means that an adversary would need to use the same kernel as the victim, since this does not trigger a reconfiguration. Therefore, the only way to exploit this vulnerability is to manipulate another user to execute a malicious application. Because such an application could also directly access the data without involving the FPGA, we consider this attack to have a low severity.

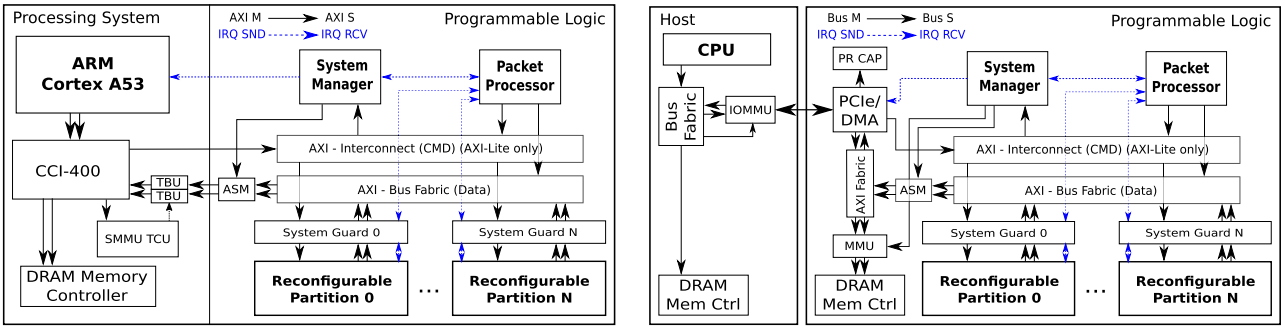
Finally, there are also side channel attacks possible. Compared to direct ones, these methods are usually slower and less reliable due to the necessary statistical analysis of leaked information. However, they are suited to selectively recover sensitive data like cryptographic keys. Mitigations often try to make the application resistant, but there are also several measures a platform can implement. For example, as described before, it is possible to detect ROs and DLs with the SSM. Furthermore, it has been shown that the error rate of power analysis attacks is significantly higher when computation and monitoring circuits are isolated and the attacker has no control over place and route [70]. Therefore, the shell-role architecture and the enforced SSM usage make a successful information disclosure less likely. There are also several traditional CPU approaches like page coloring that mitigate cache attacks on an FPGA platform [82], [83]. Nevertheless, due to the nature and high variety of side channel attacks, they are hard to prevent completely and further research is required.

## VI. SYSTEM IMPLEMENTATION

The technology preview of Xilinx and AMD initially demonstrated a very limited basic system for data center PCIe accelerators [20]. Now, with the concepts of our *HERA methodology* presented in Sections IV and V, more comprehensive concrete implementations for various platforms can be derived. A particularly important domain for the future are highly integrated systems that pack CPU cores, GPUs, and application specific hardware with reconfigurable logic into a single package or even die. A major advantage is their low overhead to access main memory compared to the longer delays and higher energy consumption of traditional PCIe add-in card transceivers. This allows simpler and more fine-grained offloading as well as greater collaboration between the different compute components. Examples of such hardware platforms are the Intel HARP SiP [84] or Xilinx Zynq UltraScale+ MPSoC systems. Even though the latter is nowadays used in embedded environments, its principles can also be applied on a bigger scale to the MPC and HPCC domains. Due to these benefits, its high level of integration, and its simplicity, we demonstrate our *HERA methodology* on this platform in the following. However, as the concepts are universal, they can also be applied to other reconfigurable systems. This is shown with a second PCIe-based Xilinx Virtex design.

### A. HARDWARE DESIGN

Fig. 6 shows two realizations of the outlined HERA hardware architecture to provide suitable platforms for a fully integrated ROCm system. First, we use the tightly integrated Zynq MPSoC system in Fig. 6a as our main target. Second, we provide a reference PCIe based Virtex UltraScale design that can be seen in Fig. 6b. As the concept is universal, both designs have the identical core structure with *Packet Processor*, *System Manager*, several *reconfigurable partitions* as well as the security related *System Guard* and *Address Space Manager*.




(a) Tightly integrated SoC

(b) PCIe add-in card

**FIGURE 6.** Structure of all needed hardware components to realize the outlined framework on Zynq UltraScale+ MPSoC and Virtex UltraScale devices.

The main differences between the designs are the connection to the host processor and the handling of additional local memory. These I/O couplings also define the basic requirements on the FPGA technology itself. First, the bus protocol implementations between participating components must enable SSVM and CC. In case of our SoC, on-chip AXI and ACE busses can be used to directly access main memory and CPU caches. This implementation uses the one-way coherent HPC ports. In contrast, the Virtex design always needs to communicate with the CPU via PCIe serial links. As such it suffers, aside from the much higher latency, from the fact that this protocol by itself does not provide cache coherency. Furthermore, to realize low overhead SSVM for user processes, CPU, mainboard, and FPGA must additionally provide the optional PCIe features *atomic operations*, *Process Address Space ID (PASID)*, *Address Translation Services (ATS)*, and *Page Request Interface (PRI)*. Several of these features are only available in the hardware and IP cores of the latest FPGA generations of Xilinx and Intel and at the moment not easily accessible from programmable logic [85]–[87]. Therefore, our PCIe based design currently also has these limitations. However, as demonstrated with the SoC design, our HERA shell already has all necessary components in place such that it can be integrated in the future. Aside from these bus features, only *Dynamic Partial Reconfiguration (DPR)* capabilities are needed to reconfigure partitions independently from the static shell. As all shell components are not supposed to change during operation, future designs can also fabricate them as dedicated ASIC components. In this case the regions can simply be treated as several smaller complete FPGAs that are regularly reconfigured. As long as these basic FPGA technology requirements are met, devices from any vendor can be used to fully realize such a stack.

The first main hardware component of the HERA framework is the *System Manager*. As outlined in Section V, it is designed as a trusted component that is only accessible via the dedicated *CMD interconnect*. In this way it is completely decoupled from all untrusted masters, which can only access the *data interconnect*. At synthesis time of the shell, the SSM configures several *System Manager* registers with a set of essential user defined shell properties like the number of *reconfigurable partitions*. These are exposed to the driver to perform an automatic configuration at system startup. Afterwards, it regulates the communication between the FPGA components and the driver. This is made possible by two further sets of registers with special functionality. The first group is accessible from user space and implements the so called HSA *doorbell signals*. Writing these informs the *Packet Processor* of new work and can potentially wake it up from energy saving modes. The second group is strictly separated and allows only privileged hardware to exchange command messages. These can be used on the one hand by the driver to e.g. maintain dispatch queues, reconfigure roles, or abort kernels. On the other hand, the *Packet Processor* can e.g. signal malicious behavior of kernels, prompt process changes in the *Address Space Manager*, or maintain memory sections at local MMUs.

The second main component is the *Packet Processor* that is fundamental for all HSA based systems such as ROCm. It parses dispatch packets submitted by the language runtime, schedules tasks, and manages accelerator cores. Generally, when offloading tasks, a process registers one or more memory regions as queues via the user space libraries and the driver. Then a dispatch packet that contains the parameters, the desired kernel function, and a completion signal event handle is written. After the *Packet Processor* is informed of the task via the doorbell signal, it parses the packet and schedules it to an accelerator core. Finally, it updates the completion signal value when the task finished execution. Such a dispatch from one queue to a fixed FPGA accelerator has also been shown by [54] and in the technology preview of Xilinx and AMD. However, this is not sufficient for a system as envisioned in Section II as is lacks reconfigurability and multi-user capabilities.

In our implementation the *Packet Processor* is realized with an embedded 64 bit MIPS CPU. It contains several further mechanisms to deal with concurrent requests of multiple processes and dynamic partial reconfiguration in a secure way. For this purpose, user kernels and their role must be uniquely identified and registered at the *Packet Processor*.

Therefore, the driver associates each of them with a unique handle. A language runtime then passes the desired one to the hardware along with every submitted dispatch packet. When a kernel with an inactive identifier is parsed, it is necessary to load its corresponding role. However, if not enough *reconfigurable partitions* are available, a currently loaded role must be exchanged with the needed new one first. For that the *Packet Processor* keeps track of which kernels were previously executed and then applies an LRU scheme to select a *reconfigurable partition* for removal. This reconfiguration request is signaled to the CPU with an interrupt. The FPGA driver is then expected to serve it with the mechanism described in Section VI-B. For this purpose, the *Packet Processor* provides two additional routines. First, *deregistration* removes an accelerator from scheduling, instructs it to flush its data buffers, and afterwards causes the *System Guard* to decouple it. Second, *registration* resets a newly configured core and adds it for future scheduling.

Further extensions are needed to efficiently handle multiple concurrent processes. To increase performance per watt, accelerators should be allowed to keep data in local buffers for delayed write-backs or reuse. However, this dirty state cannot be directly observed by other applications, since all processes are completely independent. This means data loss can occur if the executing process changes without proper handling. Therefore, the *Packet Processor* first informs a core of a pending context change to perform flushes and invalidations. This opportunity for optimization is directly used by our tightly integrated HLS tool that leverages these platform capabilities.

As identified in Section V, it is strictly required to separate the untrusted regions of the accelerators from the remaining system. For this purpose, we instantiate one *System Guard* for every *reconfigurable partition*. It continuously monitors every bus and interrupt connection between the domains. In case it detects any error message from the system, an attempted violation of a protocol, or unnecessarily stalled handshakes, it immediately shuts down the accelerator and completes outstanding transactions itself. Furthermore, this action is reported to the *Packet Processor*, which marks the queue from where the task originated as erroneous. For severe violations like an attempted access to other memory areas, it additionally instructs all other *System Guards* that monitor a task of this process to terminate them. Afterwards the driver is informed and a segmentation fault is signaled to the application.

To detect these memory access related violations and strictly isolate processes, several further components are needed. As these are, according to Section V, sensitive operations, they must only be handled by trusted parts of the shell. First of all, the address space must be associated with every bus transaction. For this purpose, the *Address Space Manager* contains a set of registers that links every component that accesses the memory to its PASID. For every incoming transaction the AXI requester ID is then mapped to the currently configured PASID. Since the bus fabric interconnection is known, this mapping is unambiguous. Every time the *Packet Processor* schedules a new task, it also initiates an update of the corresponding PASID. This can be directly inferred from its dispatch queue, since they are also uniquely assigned to a process by the trusted driver. However, as parsing dispatch packets involves handling untrusted addresses, this approach also applies to some memory accesses of the *Packet Processor* itself. For this purpose, it implements two privilege levels. The higher one can access the trusted *CMD interconnect* and sets up the processing of a packet. Afterwards, it gives up these privileges and parses it with process restricted permissions.

The data busses with annotated process information can now be checked by further components. In the ZynqMP design we utilize the integrated *SMMU* and its *Translation Buffer Units (TBU)* to realize both platform security and SSVM support. The pages used for translation are directly selected by a so called *Stream ID* when memory transactions from FPGA to DRAM pass through. This ID is defined by the mapping of the *Address Space Manager*. The associated page tables are maintained by the FPGA driver that is explained in Section VI-B. An equivalent setup can also be applied to the PCIe design and its *IOMMU* for main memory accesses when the necessary PCIe features are present in the FPGA. However, as it also contains shared local memory, an additional *MMU* is needed to protect this area. For this purpose, the *Packet Processor* provides memory allocation and deallocation services that simultaneously set up the matching MMU segment entries. If any of these units detects an access violation, it is reported to the *System Guard*, which takes the appropriate steps to terminate the process.

### B. KERNEL DRIVER

The functionality provided by the hardware is managed by a dedicated FPGA driver (HPPD). Due to the common HSA mechanisms, both AMDKFD and HPPD can share the same IOCTLs and `mmap` parameters. To offload tasks to an accelerator, the FPGA/GPU character device is opened first to get access to the platform functionality. Then, the enumerated system composition that is exported to user space via *sysfs* is parsed to find a suitable device. Afterwards, an IOCTL is used to create a dispatch queue for this specific device. Lastly, a *doorbell signal* as explained in Section VI-A is *mmapped* into user space. Now a dispatch packet can be sent via the standard HSA protocol and its content is directly interpreted by the Packet Processor.

For this memory-mapped submission it is necessary to provide access to virtual user space memory for the FPGA hardware. Therefore, the SMMU driver has been adapted to support the Stream ID scheme presented in Section VI-A. Every time a new process opens the HPPD character device, a unique PASID is assigned and bound to a context bank of the SMMU. This shares all page tables of the calling process between CPU MMU and SMMU. The SMMU driver additionally registers a notifier to be informed of changes to the

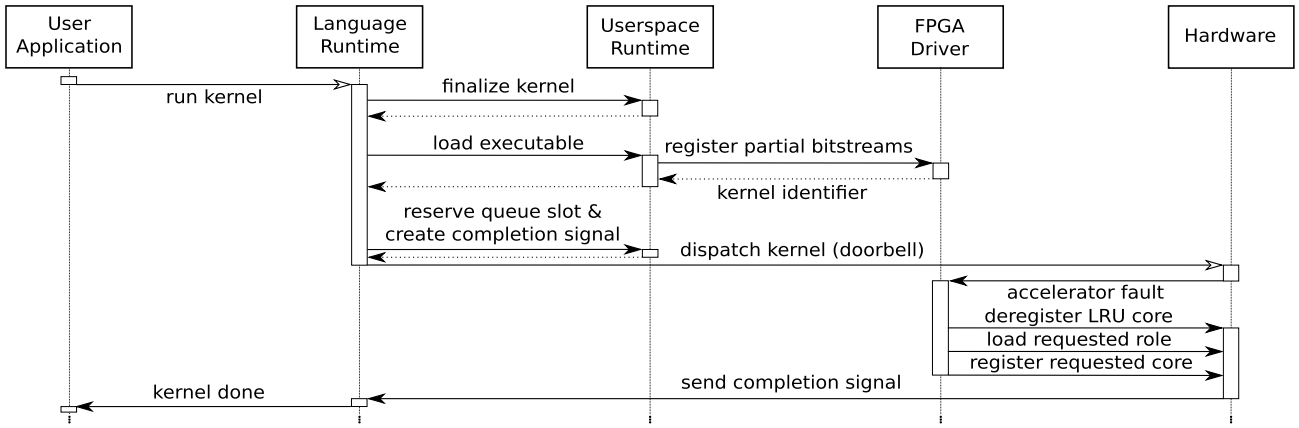



**FIGURE 7.** Sequence diagram of a simplified execution flow. The language runtime finalizes and loads the desired kernel via the HSA runtime. The latter call registers the role at the driver and assigns a unique identifier to the kernel object of the executable. The task can then be dispatched with the usual HSA mechanism. If a currently not configured kernel is inserted into the queue, the Packet Processor raises an *"accelerator fault"*. The driver then replaces an accelerator with a suitable new one.

memory mapping of the process. With this notification TBU TLBs are invalidated when necessary. Then, the associated Stream IDs are assigned to this context bank. From now on, all current and future virtual addresses of this process can be translated by the SMMU. This enables true zero-copy capabilities when passing memory pointers to the accelerators. Additionally, the SMMU driver is able to restore swapped out pages at runtime by handling context faults caused by ongoing FPGA transactions.

A fundamentally new aspect for FPGAs, in contrast to GPUs, is the reconfigurability. An overview over the mechanism that realizes it in the HERA methodology is visualized in Fig. 7. First, a language runtime registers the partial bitstreams of all kernels that can be potentially used by the process at HPPD via *sysfs*. Internally the HMAC hash value is calculated to verify the authenticity and integrity of the role as explained in Section V-B. Furthermore, this hash acts as a fingerprint of the kernel. If it has already been registered by any process, only one instance is stored in kernel space. In every case, a unique *kernel identifier* is returned to the user process. Hereby, all processes get the same handle if they register the same bitstreams. It is referenced in subsequent HSA kernel dispatch packets such that the Packet Processor can identify the needed accelerator. This reduces the number of partial reconfigurations needed and therefore improves performance. As explained in Section VI-A, the hardware raises an *"accelerator fault"* when a dispatch packet with a not loaded identifier is parsed. Upon receiving this interrupt, HPPD checks if a role with this identifier has been registered. If not, the queue is shut down and the associated application is notified. Otherwise, the driver reconfigures the system in three steps. First, the evicted accelerator is *deregistered* by calling the Packet Processor routine. Then, the FPGA Manager driver *programs the partition* with the new compatible role. Finally, the new accelerator is *registered* and execution of the dispatch packet is automatically resumed. With this method, the reconfiguration procedure is completely managed by the hardware and driver layer. Therefore, multiple concurrent processes are completely *independent* and can behave as if they were the only one running. Furthermore, a user is not able to lock the FPGA without actually executing a task. For this reason, an application kernel will eventually be executed.

### C. USER-SPACE COMPONENTS

Although the device drivers for GPUs and FPGAs share several aspects of the Linux kernel user space API, they are not identical due to the different architectures. Therefore, a further user-space layer is necessary to abstract these details and enable simpler access to device functions. This layer is designed as a device agnostic library that creates a uniform view of the platform and that can be dynamically linked against. To span a wide range of hardware, the HERA methodology extends the ROCm GPU ecosystem [21] and its implementation of the HSA runtime API [22] similarly to the joint technology preview of Xilinx and AMD [20]. Due to the common standard, new classes for FPGA devices can be seamlessly integrated.

One drawback of FPGAs is that synthesizing bitstreams costs multiple orders of magnitude more time and energy than compiling a GPU kernel. This also means that a simple GPU-like just-in-time compilation of executables at runtime would incur a significant delay. However, limiting the synthesis process to the developer PC also has significant disadvantages. It requires the exact knowledge of the target systems beforehand at development time. Furthermore, supporting even a low number of different FPGAs and configurations would make deployed executables extraordinarily big as bitstream sizes are often in the order of MB. These options limit one of the biggest strengths of FPGAs, their reconfigurability and customizability. Therefore, we propose another solution as a compromise. We run a user overarching system daemon called SSM on the client PC. It provides HLS and synthesis capabilities as well as a bitstream cache to application

software. The system administrator initially installs it with a freely chosen platform configuration file that specifies the desired FPGA setup like the number of reconfigurable partitions. SSM then automatically creates a hardware design with the correct System Manager settings that can be used by the driver and synthesizes the static shell bitstream. If an application kernel is now *finalized* (Fig. 7) for the FPGA, only the very first call of any user and any library causes a synthesis to get dynamic role bitstreams that optimally match the administrator selected configuration. This is in particular effective considering that popular high-level frameworks like BLAS and TensorFlow are typically simultaneously used by many applications and users of a system. However, even this first call can cause an inconvenient delay of several minutes at runtime. For this reason, it is advisable for developers to include this call in their application installer where longer waiting times are already expected by users. With this method any kernel can be quickly looked up in the cache at runtime and the high flexibility is preserved.

For this bitstream caching mechanism to work, it must be fully transparently integrated into the stack. That means creating an FPGA executable from device independent intermediate code must use the same interfaces as for a GPU. For this reason, the language runtime invokes at application runtime the standard *finalize kernel* procedure of the HSA API as seen in Fig. 7. When an FPGA is selected as the target, the offloaded kernel code is then sent to the SSM daemon via UNIX domain sockets. Otherwise, a standard compiler for e.g. the CPU or GPU is used. The sockets are automatically created at every system startup when the daemon loads the shell selected at installation into the FPGA. The very first invocation then causes an HLS and synthesis run that generates VHDL code and the role bitstreams for the FPGA platform configured by the administrator. Afterwards, the HMAC of these role bitstreams is calculated and jointly stored on disk according to the security requirements in Section V-B. To identify the offloaded kernel for further uses, its intermediate code is also hashed and stored as a reference for subsequent fast lookups without synthesis. In any case, the location of the *finalized* role bitstream is returned. The language runtime then uses the HSA API to register it at the FPGA driver and to *load* it into a device executable as depicted in Fig. 7. As the roles of a kernel are identical for all applications and users, HPPD is also able to save memory and reduce reconfigurations by combining duplicate registrations into a single handle.

SSM's so called *finalizer* that generates FPGA executables from intermediate kernel code consists of two parts. It has a general HLS tool and an FPGA specific synthesis and P&R tool. To provide HLS capabilities, we integrated and improved the program presented in [74] to more effectively use this desired platform. For example, it includes optimizations for typical GPU code, considers information about memory channels and speed, and uses process change and visibility notices from the Packet Processor to cache data. Furthermore, different degrees of vectorization are automatically tested to find a good fit for the available space selected by the administrator through the number of reconfigurable partitions. As a result of this process, a role bitstream for every reconfigurable partition is generated. This number can be reduced with bitstream relocation techniques to only a single one for all [88]. However, since current P&R tools for the FPGAs in our demonstrator give no guarantees for its functional correctness whatsoever, we leave it as a future option. Instead, we rely on bitstream compression to reduce their size and reconfiguration time.

### D. LANGUAGE MAPPING

With an established device independent API, it is possible to write generic language mappings. At that, every programming language has its own constructs to express parallelism and therefore its own language runtime. Their purpose is to map higher level language features to fundamental operations that can be compiled on either host CPU or accelerator device. One commonly known example is the C++ STL that defines basic algorithms. While these have been serial for a long time, parallelized versions haven been added as of C++17. Further common ones are the `parallel` pragmas in OpenMP or by concept OpenCL itself. As these descriptions of parallelism are generic, they can be mapped to any device, e.g. with a thread pool on the CPU, a GPU, or an FPGA. This has the great benefit that developers only have to interact with a language they are familiar with and that is the best suited for the application. The specific accelerator API is hidden, even though it is also device independent. Furthermore, their code can be executed on any device, even if it was not a target during development. This is especially beneficial for FPGAs as they are more specialized than and not as prevalent as GPUs nowadays. However, one peculiarity of current HLS design for FPGAs is its wide-spread use of pragmas to provide more guidance about the hardware to the HLS tool. Since this kind of optimization is device specific and there is a high probability that many applications do not use them, holistic toolchains must not rely on its usage. Instead, the automatic design space exploration of HLS tools will become more important in the future to get more traction in less the specialized domains of use. Nevertheless, both approaches are not mutually exclusive. Pragmas open up additional opportunities for developers that want to optimize their code for FPGAs in particular.

These language constructs need to be interpreted by the compiler and mapped to the function calls of the device independent API. Hereby, only one runtime for all devices has to be developed for a language, since it can subsequently be used for any device type. This makes previous approaches for GPUs in general also suitable for use with our FPGA methodology [21], [48], [72]. Internally the HSA standard defines a grid based execution model similar to CUDA. Therefore, the compiler and language runtime extract this grid information from the parallelization language construct at compile time. This is particularly simple for languages like OpenCL that already use this scheme, such that only a simple function

mapping wrapper is needed. Generally, the source code is first automatically split into a host and device part [31], [48], [72]. The device code represents the kernels that are actually executed on the accelerator. This part is cut out of the initial host source and replaced with the necessary API function calls that enumerate all available devices, select a target accelerator, and orchestrate the dispatch. As PCIe devices usually also have faster accessible local on-board memory, compilers can additionally insert topology query, allocate, and copy HSA API operations to use it. Combined with the actual source that the developer wrote for the host, this constitutes the code for the CPU. Both host and device code are then compiled into the final executable. Hereby, the latter is only lowered to a device independent intermediate ISA like SPIR-V [59] or HSAIL [22] and not yet the final accelerator ISA. This device agnostic ISA ensures the software *interoperability*. The compilation step itself establishes the initially required source code *confidentiality*. Both host and device parts are then combined into a fat binary executable that is shipped to customers.

At the client computer the executable can then simply be run, since all relevant accelerator offloading functions have already been embedded. The installed user space components of Section VI-C then generate the final device code in the *finalization* step. In case of an FPGA, this includes the role bitstream synthesis that matches the shell configuration selected by the customer administrator. If this step fails for example because even the smallest degree of vectorization generates too large accelerators, either language runtime or application code can fall back to another installed device. This is for example done by the existing OpenMP and PSTL runtimes [48] to automatically execute on the CPU if no accelerator is usable. Similarly, a runtime also decides when and if *finalizations* for multiple devices are run in parallel. The existing ones again simply sequentially block until a suitable device is found. Nevertheless, with this concept developers can work *independently* from the target system while the benefits of FPGA flexibility is still taken advantage of.

## VII. DEMONSTRATOR

### A. BENCHMARK SETUP & LIMITATIONS

In the following we demonstrate a prototype of our HERA system. Due to its early development status, it is not intended to give final performance numbers but to show the applicability of the overall concept. This HERA methodology can then for example also be used to extend other ecosystems for future domains of use like the also HSA-based joint technology preview of Xilinx and AMD. Similar to their prototype, we also demonstrate kernel dispatches over an integrated ROCm environment. However, we additionally show how our extensions improve the concept such that the system is capable of ensuring the functional requirements of future systems derived in Section II.

For our main demonstrator we use the low-cost *Ultra96* board that is based on the Xilinx Zynq MPSoC platform as shown in Section VI-A. It features a tightly integrated CPU/FPGA system on a single die where the components share a coherent interconnection structure similar to what can be found in current x86 CPU/GPU SoCs [89]. It allows us to run a fully operational *Debian Linux* with a *4.19 kernel* on it. These features make it a suitable test platform for our purposes despite its original embedded use case. Additionally, we provide a PCIe based *Xilinx VCU108* Virtex UltraScale design as shown in Section VI-A as a hardware reference for expansion card type systems. Due to its hardware limitations regarding CC and SSVM, it can only feature a subset of the HERA methodology and is therefore not presented as a full demonstrator. Nevertheless, since the central components are the same and only the connection to the host needs to be updated, the relative values still give a good indication of our methodology.

**TABLE 3. Tested algorithms.**

| Nr. | Algorithm | Data Type |
|---|---|---|
| 1 | Vector addition | `int32` |
| 2 | Gaussian $5 \times 5$ | `int16` |
| 3 | Sobel $3 \times 3$ in X and Y direction | `int16` |
| 4 | D4 Daubechies wavelet filter | `float32` |
| 5 | FC, bias and ReLU layer | `float32` |
| 6 | FC, bias and ReLU layer with barrier | `float32` |
| 7 | Element search in sorted binary tree | `int32` |

We set up several machines to test our concept. First, the *developer PC* with the VCU108. Second, the ARM based Ultra96 as the *client PC* where the applications were eventually run. One drawback of this embedded platform was that there is no Vivado support for ARM. That means it was not possible to directly finalize the kernels to bitstreams on this device. However, without loss of generality, we circumvented this issue by setting up an additional auxiliary x86 machine to create all bitstreams. It also runs an SSM instance and synchronizes its cache directory with the counterpart on the Ultra96. As a result, we could prepopulate the cache similar to what would have been done with a regular software installation as described in Section VI-C. No changes to the actual application source code were needed despite this indirection. Therefore, our setup behaved from a user and programmer perspective identical to systems where both are done on the same PC.

With this setup we developed implementations with three different programming paradigms (OpenCL, OpenMP, and PSTL) for the algorithms listed in Table 3. As accelerators should eventually also be easily usable by more inexperienced developers, we did not use highly optimized standard benchmarks. Instead, we focused on simple algorithms of several common basic operations. Furthermore, we relied on the SSVM and cache coherency features provided by the platform to further simplify the code. The biggest benefit of this approach is gained by algorithms that use sophisticated dynamically allocated data structures like the binary tree of algorithm 7. It allowed us here to forgo complex own memory management for the accelerator device and use the same data structure as the host CPU. Combined with the

**TABLE 4.** Programmable logic utilization of system components.

| Core | #Busses | LUTs | FFs | BRAM | MHz |
|---|---|---|---|---|---|
| PP | - | 6174 (8.8%) | 4312 (3.1%) | 8 (3.7%) | 200 |
| SM | - | 167 (0.2%) | 212 (0.2%) | 4 (1.9%) | 600 |
| SG | 1 | 245 (0.3%) | 268 (0.2%) | 0 (0.0%) | 450 |
| | 2 | 373 (0.5%) | 355 (0.3%) | 0 (0.0%) | 450 |
| | 4 | 610 (0.9%) | 529 (0.4%) | 0 (0.0%) | 400 |
| | 8 | 1165 (1.7%) | 877 (0.6%) | 0 (0.0%) | 400 |
| | 16 | 2189 (3.1%) | 1573 (1.1%) | 0 (0.0%) | 375 |
| | 32 | 4193 (5.9%) | 2968 (2.1%) | 0 (0.0%) | 375 |
| ASM | 1 | 146 (0.2%) | 257 (0.2%) | 0 (0.0%) | 600 |
| | 2 | 275 (0.4%) | 257 (0.2%) | 0 (0.0%) | 600 |
| | 4 | 531 (0.8%) | 257 (0.2%) | 0 (0.0%) | 600 |
| | 8 | 1110 (1.6%) | 257 (0.2%) | 0 (0.0%) | 575 |
| | 16 | 2052 (2.9%) | 257 (0.2%) | 0 (0.0%) | 500 |
| | 32 | 4198 (5.9%) | 257 (0.2%) | 0 (0.0%) | 500 |
| MMU | 1 | 1506 (2.1%) | 1890 (1.3%) | 0 (0.0%) | 425 |
| | 2 | 2804 (4.0%) | 3566 (2.5%) | 0 (0.0%) | 400 |
| | 4 | 5424 (7.7%) | 6918 (4.9%) | 0 (0.0%) | 400 |
| | 8 | 10654 (15.1%) | 13622 (9.7%) | 0 (0.0%) | 375 |
| | 16 | 21032 (29.8%) | 27032 (19.2%) | 0 (0.0%) | 375 |
| | 32 | 41931 (59.4%) | 53850 (38.2%) | 0 (0.0%) | 375 |

*familiar* programming paradigms, we could directly annotate the reference CPU code. This demonstrates the very simple and non-intrusive offloading capabilities of the concept.

The OpenCL code was compiled with an LLVM based toolchain [21]. OpenMP used the HSAIL backend of mainline GCC 8 and C++ PSTL used the extended GCC presented in [48]. All of them were initially not used for FPGA targets but only for GPUs. Nevertheless, the device agnostic HSA API allowed us to repurpose them. Due to language restrictions and the feature set state of the compilers, it was not always possible to intuitively formulate all problems. For example C++ PSTL currently does not have a notion of 2D or 3D operations like OpenCL. The remaining variants were compiled with their respective language frontends into binary accelerator independent kernels. When inspecting these kernels, we noticed that even with `-O3` only the LLVM compiler optimized the device intermediate code. The existing GCC frontends left this task to the finalizers on the *client PC*. As earlier passes of common optimizations can reduce the application binary size and finalization time, we decided to perform them on our own to e.g. additionally inline functions and combine redundant instructions.

These steps are always performed on the *developers PC* to generate the application executable that is distributed to the end-users. It does not contain any FPGA specific operations such that it could also be executed on other HSA compatible systems that i.e. only contain a GPU. The binary is then deployed to the *client PC* that has been configured and set up according to the user preferences. Thereafter, we executed all test cases to conduct the measurements.

### B. RESOURCE UTILIZATION

The hardware platform itself is composed of the basic components introduced in Sections IV, V-B, and VI-A. Table 4 shows their resource utilization with Vivado 2020.1 in different configurations relative to the size of the Ultra96. The

**TABLE 5.** Programmable logic utilization of shell configurations.

| | #RP | LUTs | FFs | BRAM | DSPs | MHz |
|---|---|---|---|---|---|---|
| MPSoC | 1 | 8066 (11.4%) | 7485 (5.3%) | 12 (5.6%) | 0 (0.0%) | 150 |
| | 2 | 8917 (12.6%) | 8634 (6.1%) | 12 (5.6%) | 0 (0.0%) | 150 |
| | 3 | 9464 (13.4%) | 9798 (6.9%) | 12 (5.6%) | 0 (0.0%) | 150 |
| | 4 | 10508 (14.9%) | 10938 (7.6%) | 12 (5.6%) | 0 (0.0%) | 150 |
| PCIe | 1 | 61501 (11.4%) | 90798 (8.4%) | 127 (7.3%) | 6 (0.8%) | 150 |
| | 2 | 68190 (12.9%) | 101357 (9.4%) | 156 (9.0%) | 6 (0.8%) | 150 |
| | 3 | 70400 (13.1%) | 106493 (9.9%) | 156 (9.0%) | 6 (0.8%) | 150 |
| | 4 | 72787 (13.5%) | 111634 (10.4%) | 156 (9.0%) | 6 (0.8%) | 150 |
| | 6 | 75245 (14.0%) | 117960 (11.0%) | 156 (9.0%) | 6 (0.8%) | 125 |
| | 8 | 77343 (14.4%) | 123893 (11.5%) | 156 (9.0%) | 6 (0.8%) | 125 |
| | 10 | 81211 (15.1%) | 130451 (12.1%) | 156 (9.0%) | 6 (0.8%) | 125 |
| | 12 | 84926 (15.8%) | 136427 (12.7%) | 156 (9.0%) | 6 (0.8%) | 125 |

absolute numbers for the VCU108 are similar, but the FPGA contains 7.6× more programmable logic. Core of every shell are the *Packet Processor (PP)* and *System Manager (SM)*. They communicate with driver and applications to dispatch tasks to the accelerator cores. As fixed components their size does not dependent on any configurable parameter like the number of reconfigurable partitions. It is therefore well suited for cases where processes work concurrently on many cores. For security and usability reasons three further components are needed. The first one is the *System Guard (SG)* that monitors the bus between untrusted core and shell. It is needed once for every reconfigurable partition with as many busses as AXI masters of the partition. Second, the *Address Space Manager (ASM)* that assigns the PASIDs and is also present once in all designs. Its number of busses is equal to the number of memory channels. The third component is the *Memory Management Unit (MMU)* that protects the local board memory from unauthorized accesses. It is only needed once in the VCU108 design, since the Ultra96 does not have local memory and the main memory is protected by the SMMU. Its number of busses is equal to the number of on-board memory channels. In general, these three components scale linearly in size, but they are often not needed in their biggest configurations. The FPGAs on both Ultra96 as well as VCU108 for example only support memory access with up to two channels. With these low numbers of busses their impact on the overall design utilization is very low. However, they can even be used for the up to 32 memory channels of current High Bandwidth Memory (HBM) FPGAs.

These basic hardware components can then be automatically configured and integrated by the SSM daemon into a complete FPGA design on the *client PC*. This process is customized by the system administrator by providing a configuration file according to the needs of this working environment. Possible options are for example the number of *reconfigurable partitions (RP)*, the number of AXI bus masters per core, or the number of memory channels. In the following we selected one channel for the *MPSoC* and two for the *PCIe* design with one AXI bus master for every RP. With these basic settings we synthesized several shell instances. Their resulting resource utilization is listed in Table 5. First of all, it can be seen that our PCIe design was much bigger

**TABLE 6.** Programmable logic utilization of roles of algorithms in Table 3.

| Alg. | Lang | LUTs | FFs | BRAM | DSPs |
|---|---|---|---|---|---|
| 1 | OCL | 5353 (7.6%) | 6126 (4.3%) | 21 (9.7%) | 0 (0.0%) |
| | OMP | 5929 (8.4%) | 7281 (5.2%) | 21 (9.7%) | 0 (0.0%) |
| | PSTL | 5931 (8.4%) | 7327 (5.2%) | 21 (9.7%) | 0 (0.0%) |
| 2 | OCL | 5091 (7.2%) | 4935 (3.5%) | 21 (9.7%) | 6 (1.7%) |
| 3 | OCL | 7881 (11.2%) | 7926 (5.6%) | 21 (9.7%) | 12 (3.3%) |
| 4 | OCL | 10459 (14.8%) | 7325 (5.2%) | 21 (9.7%) | 32 (8.9%) |
| | OMP | 10832 (15.4%) | 8139 (5.8%) | 21 (9.7%) | 32 (8.9%) |
| | PSTL | 11207 (15.9%) | 9006 (6.4%) | 21 (9.7%) | 32 (8.9%) |
| 5 | OCL | 9984 (14.1%) | 8479 (6.0%) | 21 (9.7%) | 22 (6.1%) |
| | OMP | 10565 (15.0%) | 9655 (6.8%) | 21 (9.7%) | 22 (6.1%) |
| | PSTL | 10955 (15.5%) | 10479 (7.4%) | 21 (9.7%) | 22 (6.1%) |
| 6 | OCL | 9501 (13.5%) | 7851 (5.6%) | 23 (10.6%) | 8 (2.2%) |
| 7 | OCL | 7837 (11.1%) | 9699 (6.9%) | 21 (9.7%) | 0 (0.0%) |
| | OMP | 8938 (12.7%) | 11068 (7.8%) | 21 (9.7%) | 0 (0.0%) |
| | PSTL | 8162 (11.6%) | 10228 (7.2%) | 21 (9.7%) | 0 (0.0%) |

than the *MPSoC* one. This was caused by the additional soft logic components for the PCIe endpoint, DMA units, memory controllers, channel access interleaver, MMU, and partial reconfiguration access port. These are either not needed or implemented as ASIC parts in the *MPSoC*. Therefore, the reduced *MPSoC* design can be seen as the core part of our HERA methodology. It not only utilized less than 15% of the Ultra96 resources, but it was also significantly smaller than the mandatory logic to bring up a *PCIe* board design. Second, when increasing the number of RPs, only a small fraction of the resources for one RP was needed for every new one. This was due to the fact that most components in Table 4 are not replicated and the SG that was, had an insignificant impact on the size. The major part of the increase could be attributed to more complex AXI bus fabric interconnects. This was dependent on the bus width of the memory controller. Since the *PCIe* design used four times wider 512 bit busses than the *MPSoC*, its increases were bigger. In general, it can be seen that with more than 84% free resources our static shell forms a cost-effective entry point to the accelerator hardware. Furthermore, additional logic of bigger FPGAs of the same type can almost completely be used for further function offloading.

In the following we selected the *MPSoC* design with two RPs and a lower RP resource occupation ratio for further tests, since this allowed a faster generation of role bitstreams. Although the synthesis delay at runtime can be largely avoided with the methods presented in Section VI-C, they could occasionally happen if developers do not update their installers. Therefore, we chose a less dense design with bigger RPs to present a more convenient option and a lower bound on the performance values. This is ultimately a matter of preference for the administrator as a trade-off between more compute power and faster finalization times on the *client PC*.

We then finalized all algorithms of Table 3 into their respective role bitstreams for our selected design. The utilization of the resulting accelerator cores is listed in Table 6. In accordance to the original definition, we from now on refer to the operation as ''algorithm'' and to ''the application logic itself'' [61] in hardware as ''role''. Furthermore, an offloaded code section is called ''kernel''. All tested roles needed less than 16% of the FPGA resources. This allowed a comparatively fast synthesis in the range of several minutes. However, OpenMP and PSTL roles were in general up to 14% bigger than their OpenCL counterparts. This was directly caused by the code generated by the external GCC language frontends and not a language intrinsic impairment. Both instantiated parameter and lambda expression capture call stacks in regular memory and only passed the stack pointer via dedicated `kernarg` memory. Since this stack is not distinguishable from real data for the HLS backend, it requires an accelerator to fetch the real pointers from there first. While this is not an issue for instruction based accelerators as only a few additional `load` instructions are used, it requires additional logic to be synthesized for FPGA hardware. In contrast, the OpenCL paradigm itself requires kernel arguments to be explicitly passed with dedicated `setArg()` methods and is translated as such by the compiler. These stricter `kernarg` properties allowed HLS tool and PP to optimize its usage with extra registers. Nevertheless, even for OpenMP and PSTL multiple accelerator cores could be instantiated in parallel. Since the number of CPU cores and FPGA capacity can vary widely in GPC systems, we limited both to one core per process to make them more comparable in the following.

**TABLE 7.** Runtime overhead of FPGA ROCm [µs].

| Operation | Category | ROCm | Bare |
|---|---|---|---|
| Init & find devices | global initialization | 2048 | 681 |
| Create queue | per device | 165 | 105 |
| Look up role | per device & kernel | 1761 | 1743 |
| Register role (full) | per device & kernel | 34948 | 34939 |
| Register role (compr.) | per device & kernel | 25600 | 25589 |
| Create completion signal | per dispatch (reusable) | 18 | 1 |
| Set up kernel arguments | per dispatch (reusable) | 83 | 3 |
| Reconfiguration (full) | runtime (not configured) | 7424 | 7422 |
| Reconfiguration (compr.) | runtime (not configured) | 6127 | 6124 |
| Dispatch latency | runtime | 11 | 11 |

### C. PERFORMANCE

When offloading these kernels, the overall runtime of an application is composed of two main parts. There is a constant overhead of the environment and the performance of the actual kernel execution on the accelerator. The first factor is mostly independent of the role as it has to be equivalently done for all. This can be further separated into a general offload setup time to prepare the dispatch and a task execution overhead for packet processing and signaling at runtime. These influences are itemized in Table 7. It compares the runtime of several *operations* when the user was interacting with the *bare* driver and hardware to a version where the *ROCm* user-space components were used. Hereby, the cost of an *operation* can be assigned to several *categories* depending on how often it has to be done. First, *global initialization* is only done once for every process. It enumerates all devices and sets up the whole library. Then users *create a queue* to dispatch

AQL packets. This is done at least once *per device* whereby multiple allow out-of-order processing. On this queue any kind of kernel can be dispatched, but their roles must be individually *looked up* at the daemon and *registered* at the driver. When this is done, users *create completion signals* and *set up kernel arguments*. While this is needed once *per dispatch*, the allocated structs are *reusable*. Therefore, they can be used without additional cost for multiple queues and packets if the function signature matches. Finally, the packet itself can be dispatched and processed. These costs are incurred at *runtime* and as such need to be taken into account of for every offloading operation. They can potentially be higher if the role is *not configured* such that a *reconfiguration* is triggered.

Although Table 7 shows that ROCm as a middleware library generally increases the runtime of operations, it only does so for initialization and setup tasks that are executed once. These costs are not critical runtime costs but can be seen as additional time to first launch an application. Therefore, they amortize over longer application lifetimes. Furthermore, these delays mainly arise from increased bookkeeping efforts for larger systems that e.g. provide error handling with automatic memory deallocation. For that reason, it also has the beneficial aspect of increased program stability and comfort.

The by far largest delay to set up kernel offloading is currently the *registration of roles* at the driver. In this process, the partial bitstreams for both RPs have to be transferred from the persistent storage to kernel space. The speed of the storage medium therefore has a bigger influence on this number. In case of the Ultra96 board, this is an SD card, which slowed down the process. All uncompressed bitstreams have the same combined size of 3.86 MB for every role in Table 6. This lead to a registration time of 35.0 ms. With bitstream compression, we could reduce their size by 23.5% to 2.96 MB with a small standard deviation of only 12.638 kB. As expected, the registration was in this case with 25.6 ms also up to 26.7% faster. Therefore, the time to transfer the data to the kernel has a big influence on this setup time. A further additional factor is introduced by the HMAC to validate the SSM origin, as explained in Section V-B. It is by definition performed in two steps [81]. First, the bitstream is appended to the key that is XORed with the inner padding and hashed. Length extensions of Merkle-Damgård hash functions like SHA256 cost linearly more time due to the block processing. Since the padded key is always orders of magnitude smaller than the bitstreams, its overhead compared to a regular hash that would be needed to find duplicate registrations, is not significant. Then, the result is appended to the key that is XORed with the outer padding and hashed again. This hashing is completely new, but since the result of the first step is only the output length of the hash function itself and much smaller than the bitstreams, it is again a negligible cost. Our measurements confirmed that this HMAC calculation was indeed less than 0.1% slower than the regular hashing. For this reason, we do not consider this security update a relevant time factor for any typically used hash function when registering roles.

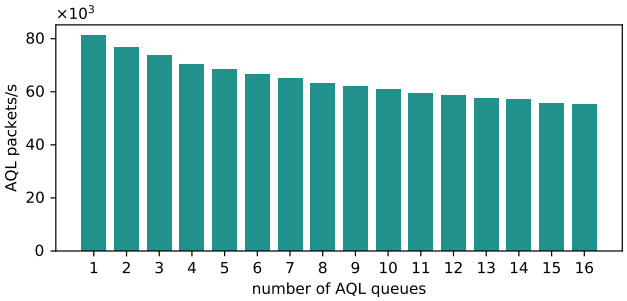



**FIGURE 8.** AQL packet throughput with parallel dispatches over multiple queues.

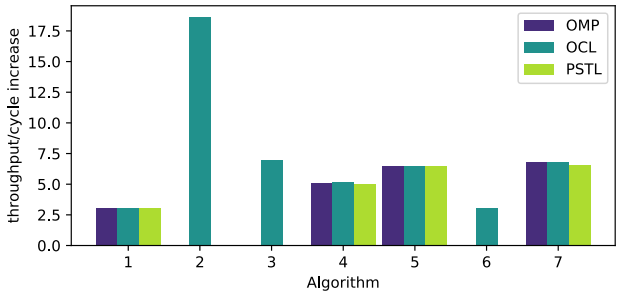



**FIGURE 9.** Work item per cycle throughput improvement of FPGA offloading relative to CPU only execution.

Furthermore, the resulting handle can generally be used for an arbitrary number of offloading invocations and is therefore only a one time cost.

Once registered, a role must still be loaded into a partition of the FPGA. This *reconfiguration* sequence was transparently carried out by our "accelerator fault" mechanism at runtime. It took on average 7.4 ms for our uncompressed 1.93 MB roles. When switching to compressed bitstreams, it could be reduced by 17.5% to 6.1 ms for 1.46 MB. This shows that the biggest part was again spent by data transfer from DRAM to Configuration Access Port and the FPGA configuration itself. Therefore, newer FPGA generations are, according to recent industry trends, expected to significantly speed up this process.

Nevertheless, this reconfiguration mechanism is only triggered if the role is currently not loaded. This happens for example when it has been previously evicted from the FPGA area. Usually, the *dispatch latency* at runtime is merely 11 µs for every offloaded task. This value includes the complete round-trip time for a vector copy kernel with one element from CPU dispatch until the completion signal event is received. Since this kernel is very lightweight, it puts the most stress on the Packet Processor and is therefore a worst case scenario. Compared to the most comprehensive prior approach, we could improve their stated runtime manager overhead of 185 µs by a factor of 16 [35]. Fig. 8 shows the overall packet throughput when these kernels were concurrently dispatched over up to 16 queues. In this case, the Packet Processor had to switch queue contexts when packets from different ones were to be processed. This further increased

the stress on the accelerator hardware. As a result, the packet throughput decreased by 32.2% for 16 concurrent queues. However, even in this case more than 55 000 kernel dispatches per second were processed. This shows that the direct communication of a language runtime with the hardware offers a low latency interface with only a small overhead on the tightly coupled MPSoC platform. This high throughput makes it suitable for multi-user systems.

The second factor that influences the offloading efficiency is the runtime of the actual kernel on the accelerator core. It is determined by the memory access performance and the quality of the circuit the HLS tool generates. Our concept as presented in Section VI-A includes up to four components in addition to the bus fabric interconnects in the data path to memory.

On the programmable logic (PL) side, we used the SG, ASM, and for designs with local memory the MMU as presented in Table 4. These components increase the access latency by a constant of 2, 0, and 4 cycles respectively but do not directly affect the throughput. This delay could impair the performance of latency sensitive accelerators, which typically use low burst lengths and few outstanding transactions. However, these are comparatively already more limited by the usually much higher DRAM access latency. Furthermore, the roles generated by the SSM also mainly transferred multiple bigger chunks at once. Therefore, this impact was not significant.

On the processing system (PS) side of the MPSoC design, we used the SMMU to translate and check virtual addresses. These translations happen either in the distributed TLBs (TBU), the central TLB (TCU), or through a full page table walk (PTW) depending on TLB hits or misses. As it is located in a different clock domain than the PL, the exact delays depend on their ratio and the exact timing. For 533 MHz PS and 100 MHz PL reference clocks, we measured a 1 cycle delay for a TBU lookup, 34 cycles for the TCU, and 75 cycles if a full PTW is needed. These delays resulted in a performance decrease of 1.0% for a $5 \times 5$ stencil, 2.2% for a matrix-vector multiplication, and 1.2% for a tree traversal with 100 000 nodes. This overhead of the tested memory access patterns is sufficiently small to still be acceptable. Therefore, the thus established SSVM is suitable to ease programming and make copy operations to device memory not needed any more.

A further factor is the cost of cache coherence. It is realized by the HPC AXI ports in the PL and the cache coherent interconnect (CCI) on the PS side. While this is a convenient feature, specifically Xilinx' ZynqMP implementation has been shown to noticeably reduce the performance [90]. We also confirmed this for our system with even slower LPDDR4 DRAM by setting up a benchmark with two AXI masters. The frequency was set to 214 MHz such that 80% of the theoretical Ultra96 DRAM throughput was requested per port. When our up to 64 kB test data resided in the cache, only 45% of the theoretical DRAM bandwidth for reads and 15% for writes were measured. By contrast, when the data was not present in same test case, 74% for reads and 84% for writes could be achieved. In comparison, the non-coherent HP ports always delivered 82% for reads and 85% for writes. Therefore, a certain degradation due to the coherency is to be expected for the currently available ZynqMP FPGAs. However, we also assume that this overhead will be smaller in future generations, similar to current x86 multi-core designs.

Based on this analysis of the memory data path, we executed several applications with our role accelerators of Table 6 on the system. Fig. 9 shows the increase in work item throughput per cycle over a pure CPU equivalent for these roles. It can be seen that all roles improved the efficiency of the application even though offloading itself entailed a fixed runtime overhead. As our previous tests showed a decreased memory throughput due to the Xilinx' cache coherency implementation, we also compared them to a VCU108 equivalent. Hereby, we used the one RP variant of the PCIe design in Table 5 and moved all data to local memory beforehand. Furthermore, we instantiated the exact same amount of computational resources as on the MPSoC, even though much more logic and memory bandwidth was available, to make them directly comparable. It could be seen that even the kernels with the highest (algorithm 2 OCL) and lowest (algorithm 6 OCL) benefit on the MPSoC platform, had an equally significant performance increase of 74.0% and 90.4% respectively. This shows again that efficient memory subsystems are important for future tightly integrated heterogeneous platforms.

Nevertheless, the overall offloading efficiency was for the most part determined by the quality of the circuit the HLS tool [74] generated, despite its less mature development state and the complete omission of pragmas. It took advantage of the HSA grid structure generated by the frontend compiler to exploit both data and instruction parallelism. Combined with their guarantees about data alignment that were inserted into the intermediate code, it enabled for example to automatically vectorize data access and form larger bursts. As developers annotate offloadable regions for any accelerator in the high-level source code, it can usually be assumed that kernels have this higher degree of data parallelism, since they are also suitable for e.g. GPUs. Furthermore, this common set of characteristics defined by the languages can ease performance portability across device classes for developers. When comparing the programming paradigms among each other, it can be seen that the OpenMP and PSTL derived roles were 2.1% slower than their OpenCL counterparts. This effect was caused by their more complex parameter transfer that also increased their resource utilization. However, this decrease was in the same small range as the SSVM overhead and therefore not crucial. This makes these programming paradigms from a performance perspective also suitable for kernel offloading.

All in all, these presented aspects of our HERA methodology show that a unified GPU and FPGA environment has indeed the potential to considerably simplify programming of heterogeneous systems in various domains. Moreover,

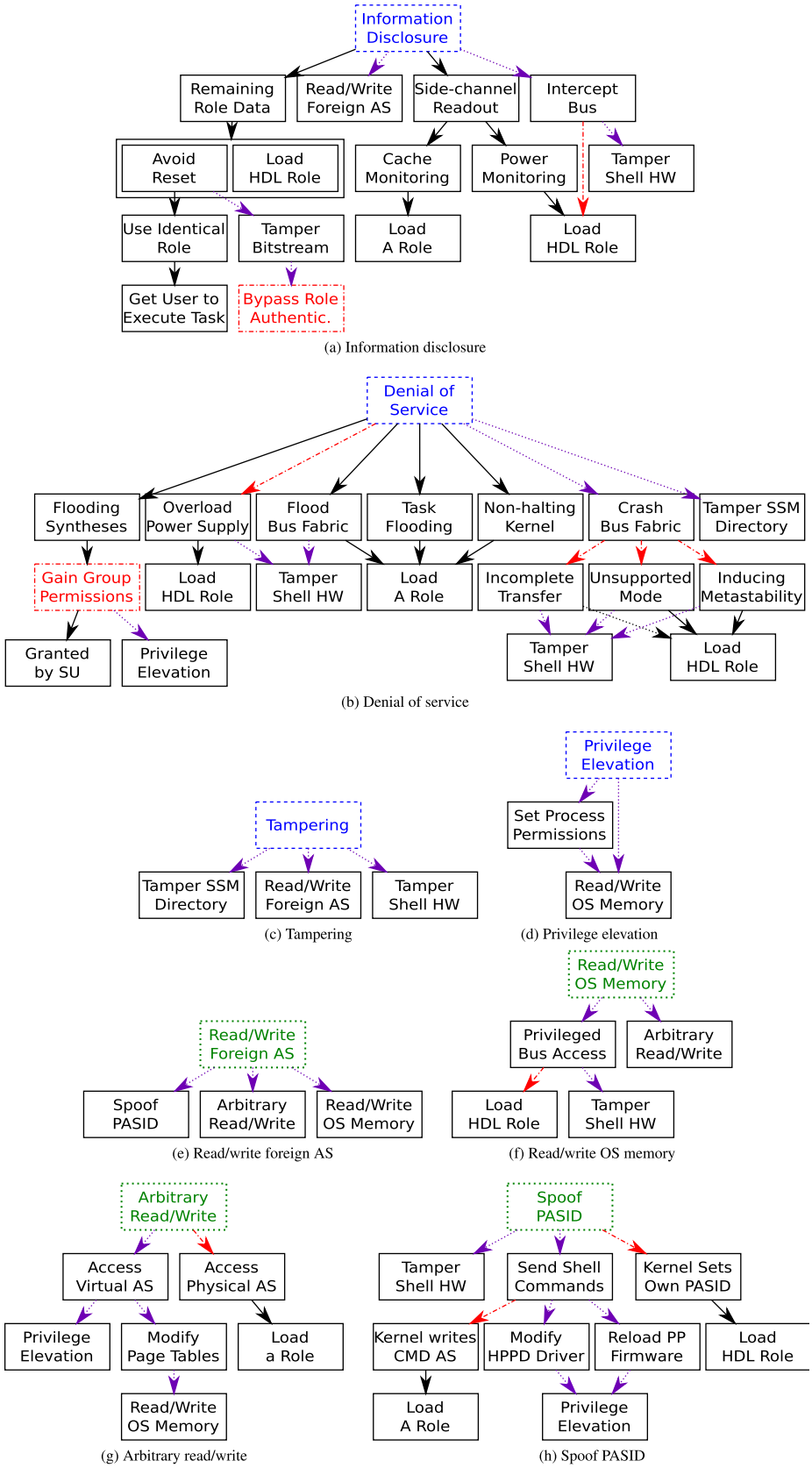



**FIGURE 10.** Attack trees that characterize the platform security.

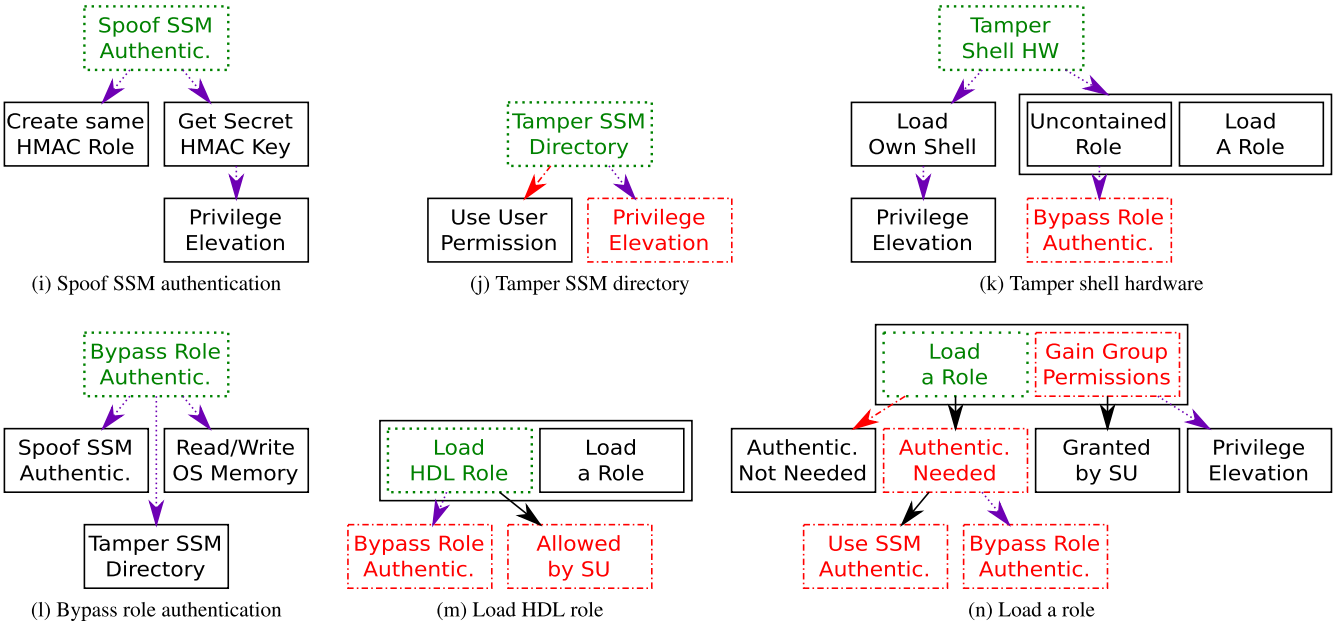



**FIGURE 10.** ***(Continued.)*** **Attack trees that characterize the platform security.**

we demonstrated that a framework as previewed by Xilinx and AMD can be extended to also fulfill stricter requirements in future MPC use cases. Since the underlying HSA standard is the same, we expect their future production-ready platforms to show similar characteristics.

## VIII. CONCLUSION

In this paper we analyzed the requirements of FPGA systems in the general-purpose computing domain and how these were previously approached. Based on this, we explained how several techniques like hardware virtualization, system shared virtual memory, and the use of wide-spread programming paradigms can be combined to build up future FPGA systems. Hereby, a special focus was also placed on the topic of security where we created a comprehensive threat model and applied risk mitigations. Based on the established ROCm ecosystem for GPUs, we derived a common methodology called HERA to build up heterogeneous systems. Finally, a prototype implementation thereof for the ZynqMP and Virtex platforms has been demonstrated.

The presented methodology and results can guide platform developers in academia and industry to design systems that are energy efficient as well as accessible. Furthermore, as it shares a common basis with the joint converged GPU and FPGA runtime technology preview of Xilinx and AMD, we could also showcase its extensibility for future midrange and personal computing use cases.

## APPENDIX
## ATTACK TREES

The attack trees used in Section V that characterize the platform security can be seen in Fig. 10. Dashed, blue root nodes are primary goals of attackers. Dotted, green nodes represent secondary goals, which enable reaching the primary ones. Child nodes represent options that can be taken to achieve a goal. Boxes that contain multiple sub-boxes require all to be fulfilled. Several possible attacks are mitigated by either introducing new conditions (red boxes with alternating dots and dashes) or removing options (red arrows with the same stroke). Dotted violet arrows signalize fully mitigated possibilities.

## REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998.
[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. JSSC-9, no. 5, pp. 256–268, Oct. 1974.
[3] R. S. Williams, "What's next? [The end of Moore's law]," *IEEE Comput. Sci. Eng. Mag.*, vol. 19, no. 2, pp. 7–13, Mar. 2017.
[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2011, pp. 365–376.
[5] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69–76, Feb. 2008.
[6] L. T. Su, "Architecting the future through heterogeneous computing," in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2013, pp. 8–11.
[7] A. DeHon and J. Wawrzynek, "Reconfigurable computing: What, why, and implications for design automation," in *Proc. Design Autom. Conf.*, Jun. 1999, pp. 610–615.
[8] N. S. Kim, D. Chen, J. Xiong, and W.-M. W. Hwu, "Heterogeneous computing meets near-memory acceleration and high-level synthesis in the post-Moore era," *IEEE Micro*, vol. 37, no. 4, pp. 10–18, Aug. 2017.
[9] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proc. IEEE*, vol. 103, no. 3, pp. 332–354, Mar. 2015.
[10] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling Simulation (SAMOS)*, Jun. 2016, pp. 235–244.
[11] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2012, pp. 47–56.

[12] X. Niu, T. C. P. Chau, Q. Jin, W. Luk, and Q. Liu, "Automating elimination of idle functions by run-time reconfiguration," in *Proc. IEEE 21st Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2013, pp. 97–104.

[13] D. Chen and D. Singh, "Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 5–12.

[14] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 93–96.

[15] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2016, pp. 77–84.

[16] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[17] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–10.

[18] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *Proc. IEEE Int. Conf. Embedded Softw. Syst. (ICESS)*, Jun. 2019, pp. 1–8.

[19] Intel Corporation. *Intel FPGA SDK for OpenCL: Programming Guide*. Nov. 2019. [Online]. Available: https://software.intel.com/sites/default/files/oneAPIProgrammingGuide_5.pdf

[20] Xilinx. (Nov. 2020). *AMD and Xilinx Demonstrate Converged ROCm Runtime Technology Preview*. [Online]. Available: https://forums.xilinx.com/t5/Xilinx-Xclusive-Blog/AMD-and-Xilinx-Demonstrate-Converged-ROCm-Runtime-Technology/ba-p/1175091

[21] Advanced Micro Devices. (Apr. 2016). *ROCm—A New Era in Open GPU Computing*. [Online]. Available: https://gpuopen.com/compute-product/rocm/

[22] HSA Foundation. (May 2018). *HSA Specification Version 1.2*. [Online]. Available: http://www.hsafoundation.com/standards/

[23] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2016, pp. 338–347.

[24] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "The strengths and behavioral quirks of Java bytecode decompilers," in *Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2019, pp. 92–102.

[25] Amazon.com. *Amazon EC2 F1-Instances*. Accessed: Dec. 6, 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[26] Xilinx. (Nov. 2019). *Vitis Unified Software Platform*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1400-vitis-embedded.pdf

[27] I. Stamelos, E. Koromilas, C. Kachris, and D. Soudris, "A novel framework for the seamless integration of FPGA accelerators with big data analytics frameworks in heterogeneous data centers," in *Proc. Int. Conf. High Perform. Comput. Simul. (HPCS)*, Jul. 2018, pp. 539–545.

[28] S. Mavridis, M. Pavlidakis, I. Stamoulias, C. Kozanitis, N. Chrysos, C. Kachris, D. Soudris, and A. Bilas, "VineTalk: Simplifying software access and sharing of FPGAs in datacenters," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–4.

[29] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio, "Heterogeneous exascale supercomputing: The role of CAD in the exaFPGA project," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 410–415.

[30] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The TaPaSCo open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *Proc. 15th Int. Symp. Appl. Reconfigurable Comput.* Cham, Switzerland: Springer, 2019, pp. 214–229.

[31] L. Sommer, J. Korinth, and A. Koch, "OpenMP device offloading to FPGA accelerators," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2017, pp. 201–205.

[32] D. Pnevmatikatos, K. Papadimitriou, T. Becker, P. Böhm, A. Brokalakis, K. Bruneel, C. Ciobanu, T. Davidson, G. Gaydadjiev, K. Heyse, and W. Luk, "FASTER: Facilitating analysis and synthesis technologies for effective reconfiguration," *Microprocessors Microsystems*, vol. 39, nos. 4–5, pp. 321–338, Jun. 2015.

[33] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey, "System architecture for network-attached FPGAs in the cloud using partial reconfiguration," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 293–300.

[34] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *Proc. IEEE 12th Int. Conf. Ubiquitous Intell. Comput., IEEE 12th Int. Conf. Auton. Trusted Comput., IEEE 15th Int. Conf. Scalable Comput. Commun. Associated Workshops (UIC-ATC-ScalCom)*, Aug. 2015, pp. 1078–1086.

[35] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized execution runtime for FPGA accelerators in the cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017.

[36] Intel Corporation. (Dec. 2018). *Intel FPGA SDK for OpenCL: Programming Guide*. [Online]. Available: https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[37] J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, J. Clarkson, and C. Kotselidis, "Dynamic application reconfiguration on heterogeneous hardware," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2019, pp. 165–178.

[38] M. Papadimitriou, J. Fumero, A. Stratikopoulos, and C. Kotselidis, "Towards prototyping and acceleration of Java programs onto Intel FPGAs," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2019, p. 310.

[39] M. Knaust, F. Mayer, and T. Steinke, "OpenMP to FPGA offloading prototype using OpenCL SDK," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2019, pp. 387–390.

[40] S. Lee and J. S. Vetter, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, 2014, pp. 115–120.

[41] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 544–554.

[42] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2011, pp. 33–36.

[43] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2013, pp. 270–277.

[44] Maxeler Technologies. (Jun. 2013). *Programming MPC Systems*. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf

[45] N. Trifunovic, H. Palikareva, T. Becker, and G. Gaydadjiev, "Cloud deployment and management of dataflow engines," in *Proc. 1st Int. Workshop Next Gener. Cloud Archit.*, Apr. 2017, pp. 5:1–5:6.

[46] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *Proc. 22nd Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 2008, pp. 76–103.

[47] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, 2012, pp. 271–276.

[48] P. Jääskeläinen, J. Glossner, M. Jambor, A. Tervo, and M. Rintala, "Offloading C++17 parallel STL on system shared virtual memory platforms," in *Proc. Int. Conf. High Perform. Comput. Workshops*. Cham, Switzerland: Springer, 2018, pp. 637–647.

[49] W. Wang, M. Bolic, and J. Parri, "PvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES ISSS)*, Sep. 2013, pp. 1–9.

[50] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–7.

[51] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 1–23, Sep. 2015, Art. no. 22.

[52] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with FPGAs and GPUs," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2010, pp. 115–124.

[53] M. Weinhardt, A. Krieger, and T. Kinder, "A framework for PC applications with portable and scalable FPGA accelerators," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, Dec. 2013, pp. 1–6.

[54] M. Reichenbach, P. Holzinger, K. Häublein, T. Lieske, P. Blinzer, and D. Fey, "Heterogeneous computing utilizing FPGAs," *J. Signal Process. Syst.*, vol. 91, no. 7, pp. 745–757, May 2018.
[55] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for zero-copy sharing of pointer-rich data structures in heterogeneous embedded SoCs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1947–1959, Jul. 2017.
[56] P. Vogel, A. Marongiu, and L. Benini, "Exploring shared virtual memory for FPGA accelerators with a configurable IOMMU," *IEEE Trans. Comput.*, vol. 68, no. 4, pp. 510–525, Apr. 2019.
[57] M. Damschen, H. Riebler, G. Vaz, and C. Plessl, "Transparent offloading of computational hotspots from binary code to Xeon Phi," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 1078–1083.
[58] H. Riebler, G. Vaz, T. Kenter, and C. Plessl, "Transparent acceleration for heterogeneous platforms with compilation to OpenCL," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, pp. 14:1–14:26, Apr. 2019.
[59] K. Group. (Jan. 2019). *SPIR-V, Extended Instruction Set, and Extension Specifications*. [Online]. Available: https://www.khronos.org/registry/spir-v/
[60] S. Narayanan, N. Haemel, and J. A. Bolz, "Application load times by caching shader binaries in a persistent storage," U.S. Patent 20 140 043 333 A1, Feb. 13, 2014.
[61] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, and M. Haselman, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 13–24.
[62] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on FPGA virtualization," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 131–137.
[63] O. Knodel, P. Genssler, F. Erxleben, and R. Spallek, "FPGAs and the cloud—An endless tale of virtualization, elasticity and efficiency," *Int. J. Adv. Syst. Meas.*, vol. 11, nos. 3–4, pp. 230–249, 2018.
[64] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, "The Feniks FPGA operating system for cloud computing," in *Proc. 8th Asia–Pacific Workshop Syst.*, Sep. 2017, pp. 1–7, Art. no. 22.
[65] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. IEEE 22nd Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2014, pp. 109–116.
[66] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Proc. 11th ACM Conf. Comput. Frontiers*, May 2014, pp. 1–10, Art. no. 3.
[67] K. Eguro and R. Venkatesan, "FPGAs for trusted cloud computing," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 63–70.
[68] F. Hategekimana, J. M. Mbongue, M. J. H. Pantho, and C. Bobda, "Secure hardware kernels execution in CPU+FPGA heterogeneous cloud," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 182–189.
[69] G. Provelengios, D. Holcomb, and R. Tessier, "Characterizing power distribution attacks in multi-user FPGA environments," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 194–201.
[70] M. Zhao and G. E. Suh, "FPGA-based remote power side-channel attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 229–244.
[71] C. Ramesh, S. B. Patil, S. N. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier, "FPGA side channel attacks without physical access," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2018, pp. 45–52.
[72] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proc. 2nd Workshop LLVM Compiler Infrastruct. (HPC LLVM)*, New York, NY, USA, 2015, pp. 7:1–7:6.
[73] W. Bauer, P. Holzinger, M. Reichenbach, S. Vaas, P. Hartke, and D. Fey, "Programmable HSA accelerators for Zynq UltraScale+ MPSoC systems," in *Proc. Euro-Par, Parallel Process. Workshops*. Cham, Switzerland: Springer, 2018, pp. 733–744.
[74] P. Holzinger, M. Reichenbach, and D. Fey, "A new generic HLS approach for heterogeneous computing: On the feasibility of high-level synthesis in HSA-compatible systems," in *Proc. 18th Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation*, Jul. 2018, pp. 18–27.
[75] W. H. Wen-Mei, *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. San Mateo, CA, USA: Morgan Kaufmann, 2015.
[76] F. Swiderski and W. Snyder, *Threat Modeling*. New York, NY, USA: Microsoft Press, 2004.
[77] B. Schneier, "Attack trees," *Dr. Dobb's J.*, vol. 24, no. 12, pp. 21–29, 1999.
[78] A. Wild and T. Güneysu, "Enabling SRAM-PUFs on Xilinx FPGAs," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–4.
[79] D. R. E. Gnad, F. Oboril, and M. B. Tahoori, "Voltage drop-based fault attacks on FPGAs using valid bitstreams," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–7.
[80] I. Hadžić, S. Udani, and J. M. Smith, "FPGA viruses," in *Field Programmable Logic and Applications*. Berlin, Germany: Springer, 1999, pp. 291–300.
[81] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, document RFC2104, New York, NY, USA, 1997.
[82] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proc. 41st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2011, pp. 194–199.
[83] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.
[84] H. Schmit and R. Huang, "Dissecting Xeon+FPGA: Why the integration of CPUs and FPGAs makes a power difference for the datacenter: Invited paper," in *Proc. Int. Symp. Low Power Electron. Design*, New York, NY, USA, Aug. 2016, pp. 152–153, doi: 10.1145/2934583.2953983.
[85] Xilinx. *DMA/Bridge Subsystem for PCI Express V4.1*. Accessed: Aug. 26, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf
[86] Xilinx. *UltraScale+ Devices Integrated Block for PCI Express V1.3*. Accessed: Aug. 26, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/pcie4_uscale_plus/v1_3/pg213-pcie4-ultrascale-plus.pdf
[87] Intel Corporation. *PCIe Intel FPGA IP*. Accessed: Aug. 26, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/interface-protocols/m-pci-express-protocol.html
[88] A. Lalevée, P.-H. Horrein, M. Arzel, M. Hübner, and S. Vaton, "AutoReloc: Automated design flow for bitstream relocation on Xilinx FPGAs," in *Proc. Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2016, pp. 14–21.
[89] Intel Corporation. *Intel Processor Graphics Gen11 Architecture*. Accessed: Aug. 16, 2021. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf
[90] S. W. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, and W.-M. Hwu, "Analysis and optimization of I/O cache coherency strategies for SoC-FPGA device," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 301–306.

**PHILIPP HOLZINGER** received the master's degree in computer science from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2017. He is currently a Researcher with the Chair of Computer Architecture, FAU. His research interest includes the design of heterogeneous system architectures with a focus on reconfigurable and near-memory computing.

**MARC REICHENBACH** (Member, IEEE) received the Diploma degree in computer science from Friedrich Schiller University Jena, Germany, in 2010, and the Ph.D. degree from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2017. From 2017 to 2021, he worked as a Postdoctoral Researcher with the Chair of Computer Architecture, FAU. Since 2021, he has been heading the Chair of Computer Engineering, Brandenburg University of Technology Cottbus-Senftenberg (BTU), Germany, as a Substitute Professor. His research interests include novel computer architectures, memristive computing, and smart sensor architectures for varying application fields.