Apiary: An OS for the Modern FPGA

Katie Lim
katielim@cs.washington.edu
University of Washington

Matthew Giordano
mgiordan@cs.washington.edu
University of Washington

Irene Zhang
irene.zhang@microsoft.com
Microsoft Research

Baris Kasikci
baris@cs.washington.edu
University of Washington

Thomas Anderson
tom@cs.washington.edu
University of Washington

Abstract

Many datacenter operators have deployed FPGAs as hardware accelerators because their reconfigurability allows them to be repurposed as the application mix changes. Directly attaching the FPGA to the network further reduces latency, improves cost-performance, and reduces energy use relative to mediating network communications with CPUs. However, building accelerated applications or services for direct-attached FPGAs is challenging, especially with the complex I/O and multi-accelerator capacity of modern FPGAs. To address this, we propose Apiary, a microkernel operating system for direct-attached FPGA accelerators. The key idea in Apiary is to raise the level of abstraction for accelerated application code, with security, virtualization, threaded execution, and interprocess communication provided by the hardware OS layer.

CCS Concepts

• **Hardware → Hardware accelerators**; • **Software and its engineering → Operating systems**.

Keywords

FPGA, Operating System, Hardware accelerators

1 Introduction

Datacenter operators and the research community have increasingly turned to specialized hardware for higher performance, lower cost, and better energy efficiency than general-purpose processors. Two models have emerged for communication with accelerators: host-mediated versus direct-attached. Our focus is on direct-attached accelerators, where the accelerator communicates with the datacenter network via a hardware network stack. By bypassing the CPU, a direct-attached accelerator reduces CPU overhead, lowers latencies, and further reduces energy. For example, NVIDIA has introduced GPUDirect, which allows an RDMA smartNIC to communicate directly with the GPU [33]. Microsoft has deployed direct-attached FPGAs to accelerate ML inference with significant energy and latency benefits [14].

FPGAs are an especially intriguing option as an acceleration platform for cloud computing due to their reconfigurability. ASICs, although more efficient for a fixed workload, have a high initial cost and cannot evolve with customer needs. As a result, many large cloud providers have deployed FPGAs for their own services [4, 7, 17] and to rent out to customers [2, 3, 5, 8, 20, 31].

Unfortunately, accelerating applications to take advantage of the capabilities of FPGAs is challenging, especially with the complexities introduced by modern FPGAs. Compared to the software development environment on a CPU, the current state of FPGA development infrastructure is like being handed a CPU with the BIOS and bootloader, and little else - roughly where application development stood before the development of operating systems. Developers are exposed directly to low-level and non-portable interfaces of various I/O devices such as memory controllers or the Ethernet MACs, and they must wrestle with device-specific details in order to build next-level services, such as memory allocation and network protocols, that are needed before one can even start to consider application logic. Modern FPGAs exacerbate this with a wide range of potential I/O devices (e.g. Ethernet vs PCIe vs CXL for accelerator communication). The logic capacity of modern FPGAs is sufficient to instantiate multiple accelerators on the same FPGA, but that introduces resource multiplexing, interprocess communication, and fault isolation issues. While software application code can leverage a convenient, portable OS interface that addresses these issues, this does not exist in the direct-attached hardware setting. In other words, we need a hardware operating system to support portable, modular applications for direct-attached FPGAs, akin to what the OS research community achieved for CPU software in the early 70s.

To fill this role, this paper proposes Apiary. Building off lessons from the OS community, Apiary is structured as a microkernel with a message passing layer that connects hardware OS services and application logic. The question we ask

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05
https://doi.org/10.1145/3713082.3730385

is what hardware primitives do we want for application logic portability, programmability, and fault isolation in complex, multi-function network-attached FPGAs that do not rely on CPUs. Earlier efforts to build FPGA operating systems, such as Coyote [26] and AmorphOS [25], delegate key operating system functions such as memory management and virtualization to an attached server CPU. By providing all-hardware OS primitives, Apiary can improve latency, latency variability, resource overhead, and energy efficiency.

Our initial target is services within a microservice application or data processing pipeline. These applications are already decomposed into loosely-coupled elements, so portions of application can be accelerated without converting the entire application. At the same time, typical microservices are more complex than a simple CPU offload. An accelerated service could have its own state that it needs to maintain between invocations, it may be part of a complex call chain, and debugging/monitoring support is essential in practice. Calls to other modules may be local or remote, and each module may be independently scaled up or down to match demand. Microservices are often throughput or latency-sensitive, making it valuable to avoid CPU mediation in the common case.

As with microkernels, both internal Apiary services and application logic use the same interconnection model — message passing over a switched interconnection fabric, with hardware-enforced capabilities for access control to shared resources such as memory regions. Each module is wrapped in an Apiary shell that interfaces to the fabric and manages capabilities on the module's behalf similar in style to Barrelfish [12]. This model enables us to support a portable, high-level, device-independent interface for applications. It also supports modules being scaled out to meet the specific use case and allows new OS services and abstractions to be easily inserted and implementations changed without affecting other modules. Implementation errors in one module do not propagate to other modules except through defined message-passing interfaces.

2 Motivation

FPGAs are on a long-term upward trend in size and board complexity. This means they are applicable to more use cases, but this has also resulted in a more complex developer experience. Consider the variety of different I/O devices available on FPGA boards from Intel and AMD. Modern boards include high speed versions of I/O devices present in previous generations, such as PCIe Gen 5, 100 Gbit networking, and HBM memory [6, 22]. For any single type of I/O device, developers are expected to interact through IP cores provided by the vendor, which differ between boards and even between speeds. Modern boards have also added new types of I/O, such as storage [9] or CXL[10, 22].

Family	Year Released	Part Number	Logic Cells
Virtex 7	2010	XC7V585T	582,720
	2010	XC7VH870T	876,160
Virtex	2016	VU3P	862,000
Ultrascale+	2018	VU29P	3,780,000

Table 1: Logic cell counts for the largest and smallest FPGA parts in the previous Virtex family and the current Virtex family.

To compare size, we looked at Xilinx FPGAs from the most recent UltraScale+ and the previous 7 series, as shown in Table 1. Comparing the smallest parts, the number of logic cells has increased by about 50%, while the largest parts have scaled up by 3x between generations. Using these larger FPGAs is still a developing field, with work exploring multi-accelerator systems [11, 13, 25, 26, 39].

To better illustrate complexities in the design process, consider customizing a video encoding service to accelerate part of a video processing pipeline. Requests to the service are a chunk of video, which the service processes and then sends to the next stage of the pipeline. We would like to incorporate third-party accelerators, such as compression; to reduce resource stranding, the FPGA should be shared with other users.

The first difficulties are the complexity of FPGA boards and the variety of I/O devices. These pose challenges related to programmability and portability and must be overcome to even run the accelerator on the FPGA. Software OSes provide portable abstractions for common I/O devices, but typically there is no equivalent when developing for FPGAs. As a result, developers are exposed to the full complexity of interfaces and operational details and need to build higher-level services which would often be taken for granted in software (e.g. memory allocation, reliable network protocols). Because implementations are typically designed for a specific project, reusing previously developed capabilities often requires substantial engineering.

Further, when an FPGA board is chosen for development, the accelerated application is locked into that board. For a given type of I/O device (e.g. memory, PCIe, networking), interfaces for the IP cores managing those devices often differ between boards, even within a single vendor. For example, the interface and reset process for Xilinx's 10 Gbit Ethernet IP core and 100 Gbit Ethernet IP core are different, so additional infrastructure is needed to support both 10Gbit and 100Gbit IP cores.

A second set of challenges are in the complexity of the accelerated systems that can be built with larger FPGAs. For example, we might replicate the encoding accelerator to provide additional encoding throughput. If the encoding accelerator uses DRAM, there is no subsystem that allows isolation for sharing the DRAM address space. This means that the original encoding accelerator now must be modified to enable this sharing. This may not be feasible, e.g., if the accelerator was originally developed by a third party. Consider if we were to

use the extra FPGA area to instantiate accelerators specialized for other functionality. These accelerators could be composed with existing accelerators on the board or they could be independent, even belonging to another user. For example, the encoding accelerator could be composed with a compression accelerator to produce a compressed, encoded video stream. Since compression is a common function, we might want to use a third-party accelerator. This accelerator would not be designed to participate in a bespoke memory partitioning setup and would require memory isolation.

Isolation is even more important if the FPGA is used to host accelerators for another user's application. For example, another user might want to use the FPGA to host an independent key-value store (KV-store) application. In this case, isolation is required, both for multiplexing, and also security. Because of the current lack of infrastructure, every accelerator is by default considered trusted, but this is not a good model. We do not want, for example, any accelerator of the KV-store application to be able to communicate with any accelerator in the encoding application. This could occur due to misbehavior from a bug or maliciously, if the KV-store is attempting to interfere or snoop on the computation of the encoder.

3 Design Goals

To facilitate implementing complex, multi-accelerator FPGA systems, Apiary has the following design goals:

- **Portability:** Apiary should define a stable, standard interface portable across FPGA boards.
- **Programmability:** Apiary should simplify the developer experience by providing common utilities such as networking and memory management, as well as debugging and tracing support at the message passing layer.
- **Composability:** Apiary should support complex multi-accelerator applications, including reuse of existing accelerators.
- **Flexibility:** Apiary should put as few restrictions on accelerator implementations as possible, including choice of implementation language and performance goals.
- **Simplicity:** Apiary subsystems should be designed for simplicity. Logic resources reserved by Apiary for its subsystems are not subsequently available for user functionality, regardless of whether the provided functions are used or not.
- **Scalability:** Apiary should be designed to easily support numerous accelerators, and to support scale out of those elements, without manual optimization.
- **Isolation:** Apiary should follow the principle of least privilege and provide isolation to limit the impact of bugs and reduce security vulnerabilities.

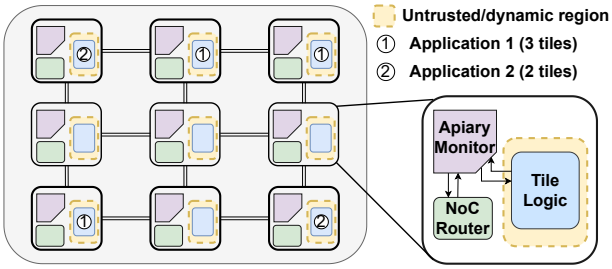

Figure 1: An overview of Apiary's architecture. This configuration has two applications composed of multiple accelerators. Each tile contains a NoC router for communication, Apiary's monitor to provide isolation and manage capabilities, and an accelerator or Apiary service. The monitor and router are trusted, while the accelerator logic is untrusted and may be dynamically reconfigured.

3.1 Limitations

While we do our best to address isolation problems that occur at the application level, it is difficult to completely eliminate circuit-level interference. Addressing these side channels is an active field of research. We briefly discuss some examples here.

In the multitenant setting, there is a potential for power virus attacks by a malicious accelerator using specific power-wasting circuits [35]. These are typically mitigated by the vendor FPGA build tools themselves using design rule checking during bitstream creation or bitstream analysis after the build process [27]. Multitenant FPGAs can also be vulnerable to circuit-level side channels where a malicious colocated accelerator can implement certain sensors that allow it to characterize the computation of a victim accelerator [16]. Potential mitigations for these types of attacks include design rule checking or placement constraints.

Ultimately, we consider circuit-level attacks out of scope for Apiary due to the vast majority of the mitigations requiring modification or participation in the FPGA bitstream build process itself. However, we believe that providing isolation at the application level is still valuable when constructing multi-accelerator systems. These systems can manifest complex application-level interactions around multiplexing or composition that Apiary and application-level isolation can address.

4 Apiary Design

Basic OS concepts in software that programmers take for granted, such as "what is the boundary of isolation" or "how do processing elements communicate with one another", do not have agreed upon answers in hardware. In this section, we discuss hardware versions of common OS abstractions and describe how we might realize these abstractions in Apiary to accomplish our goals.

4.1 Apiary Architecture

Apiary is a Network-on-Chip (NoC) based hardware microkernel. Each tile on the NoC contains an untrusted accelerator,

an Apiary monitor, and a NoC router, as shown in Figure 1. The accelerator slot can be used either by an OS service such as networking or a user accelerator. These untrusted tile slots are dynamically instantiated regions, while Apiary's framework resides in the static area, so accelerators can be reprogrammed independent of each other and the rest of Apiary's framework. We omit discussion of scheduling what is configured into the untrusted slot; AmorphOS and Coyote both explore scheduling of partial reconfiguration. We instead focus on inter-accelerator interaction.

Accelerators communicate with each other or services via message passing over the NoC. The Apiary monitor serves an accelerator's interface to the OS, so all messages go through it. This allows the monitor to implement the isolation properties we desire, which we elaborate on in following sections.

In Apiary, an application is one or more accelerators that communicate with each other to complete a computation. This can be a pipeline as in the example in Section 2, a replicated accelerator with internal load balancing for higher bandwidth, or it could involve a more complex series of interactions. We aim to support mutually distrusting applications running on the same board with the restriction that distrusting applications may not share the same physical tile because Apiary cannot prevent data leaks within an accelerator.

4.2 Process Granularity

In hardware, there are no trust boundaries or fault isolation. This means that all accelerators and OS services in current systems implicitly trust one another, and any faults that occur can affect all other processing entities in the system. This is a difficult model to develop under. Different users will not trust each other's applications, and even assuming mutually trusting users, there are no guarantees on how errors or faults will be handled. Recall that we want to be agnostic as to application implementation choices. In software, the OS uses a process as a trust and fault isolation boundary. To provide isolation, we explore how to go about constructing a process abstraction for Apiary.

For Apiary, we define our process granularity as one user context running on one accelerator. Applications can span more than one accelerator and therefore are multi-process. Processes on different physical accelerators are considered distrusting unless they specifically establish interprocess communication discussed further in Section 4.5. Processes or contexts on the same physical accelerator are mutually trusting, but should still be fault-isolated.

4.3 Accelerator Interfaces

Accelerators have interfaces at two levels. At the physical level, the interface refers to the wires, what they represent, and their topology. Examples include AXI (used often by Xilinx) or Avalon (used often by Intel) buses and these have to do with the mechanics of how two modules exchange data. The accelerator also has an interface at the "API" level: the description of what functionality is available from an accelerator and what data needs to be passed over the physical interface to invoke it. We are concerned with Apiary's physical interfaces for composability and scalability and its API interface for programmability and portability.

We want a physical layer for Apiary that can scale to a large number of different services, both for invoking I/O services and for composing with other accelerators. In previous work [25, 26], the number of physical interfaces is coupled with the number of services available; one set of module ports is used for connecting to the networking stack while a separate set is used for communicating with memory. This means that when adding or removing services, the number of physical interfaces and the underlying wires are directly impacted. This is an important consideration as the number of I/O options on an FPGA board increase.

Apiary's physical interconnect is a NoC. The NoC allows us to move service naming to an API-layer interface by making the destination ID a message field, so we can use the same physical interface to communicate with multiple services, enabling Apiary to scale. Using a NoC is also advantageous because FPGAs have begun to offer hardened NoCs that are instantiated in dedicated logic, so they can run more efficiently and leave more FPGA logic resources for users [1, 21, 36].

At the API level, we are concerned with the interface that Apiary presents to accelerators. In software, an OS typically comes with an established set of syscalls and runtime libraries which make up the interface. An application can depend on the OS to provide this interface across a large number of different hardware configurations (e.g. different x86 CPUs, different speed NICs) and as a result, the application can run on different possible hardware configurations without modifcation. There is simply not a set of expected services for FPGA accelerators, which as discussed in Section 2 poses programmability and portability challenges.

Apiary addresses API-level challenges by defining a standard interface to higher-level system services that is the same on every tile across FPGAs. The per-tile Apiary module is responsible for presenting this interface. As part of this interface, the Apiary module provides a table that maps logical service names to underlying physical units since service identification is now in the API layer. It also maintains a table of capabilities for each tile, such as which other physical components the application logic is allowed to communicate with and which memory regions it is allowed to access.

4.4 Fault Handling

Today, it is standard for accelerators to be assumed to always be correct and trusted, so there is no error-handling infrastructure or defined behavior in hardware. This is ultimately

unrealistic when building complex applications or ones where features are rolled out to users over time. At minimum in Apiary, we would like to have a fail-stop model for an accelerator: if an accelerator it encounters an error in a process and cannot complete its computation, it should not be able to affect other Apiary services or other unrelated accelerators. Ideally though, we would like to bring the fault behavior more on par with software. If an error occurs in one user context within an accelerator, other independent processes on the accelerator can keep running. Which model is achievable depends on the parallel processing model we have for accelerator processes.

In software, processes are typically concurrent and preemptible. The OS can interrupt a process running on a core and resume it later, because a CPU exposes well-defined architectural state to software that captures the context of the program. This allows the OS to swap out any misbehaving processes and repurpose the core for another process. In contrast, accelerators all have different sets of architectural state that is not readily available to access, so it is much harder to capture the context of a process. Previous FPGA OSes [25, 26] have provided an interface to the accelerator, allowing it to externalize the state it needs and yield cooperatively. This interface allows accelerator processes to be concurrent, but all processes are trusted to yield. If an error is encountered, the process may never yield. To deal with untrusted processes, accelerators need to be preemptible. Preempting an accelerator means identifying an accelerator's architectural state for a particular user's context, such as a particular network connection, so that it can be stopped at any cycle. This is difficult, because there may be intermediate state during an invocation that would not be externalized in the concurrent setting. One method used by previous work on FPGA virtualization [29] was to do static analysis to identify and expose state that would need to be saved and restored to swap out an accelerator.

If an accelerator is only concurrent, then the best Apiary (or any FPGA OS) can achieve is a fail-stop model when an error occurs. The Apiary monitor can prevent it from further interacting with the rest of the system by draining all outgoing or incoming messages and returning an error to any accelerator that tries to communicate with it. If an accelerator is preemptible, then the Apiary module can instead swap out the process when it detects an error, and the other processes on the accelerator can continue executing.

4.5 Communication

One of the crucial privileged components in a microkernel is interprocess communication (IPC). The purpose of IPC is to allow OS-managed communication between two computations that may not trust each other. In software, the kernel is responsible for implementing these channels which should be performant as well as accessed controlled. For composability and isolation, Apiary also needs an accessed controlled communication primitive.

A form of IPC already exists between accelerators on FPGAs in the form of queues that are used to pipeline accelerators [19, 26, 39]. Because accelerator computation is usually trusted, these queues are not accessed controlled in any way. With untrusted accelerators, having permissioned access and rate limiting are necessary to prevent malicious accelerators from either accessing unauthorized resources or causing resource exhaustion. Even in the case where all accelerators trust each other, rate limiting or access control can help mitigate unintentional behavior that degrades performance.

In Apiary, the infrastructure to do message passing and routing already exists since we use a NoC. The Apiary monitor sits between the accelerator and the NoC, so it can inspect all communications between the accelerator and the rest of the system to enforce access control or other policy. By using a NoC we can take advantage of prior NoC research for implementing our IPC layer, because prior work has addressed topics such as security [37, 38], application message level deadlock [30, 32], quality of service guarantees [18, 34], and other common concerns in software IPC infrastructure.

4.6 Memory isolation

Memory isolation is also an important feature that is not provided by default on an FPGA. This is in sharp contrast with page-based address space virtualization for CPUs. Previous virtual memory systems for FPGAs have focused on managing shared virtual memory pages between CPUs and FPGAs [26, 28]. Because CPU memory translation units are hardware, these page sizes have a single or a small, fixed choice of page sizes. Shared memory can be implemented by mapping the same physical pages into multiple address spaces.

However, it is unclear that a fully paged translation system is necessary in Apiary for memory isolation and address translation between accelerators. Accelerators often gain much of their performance from specializing to their memory access patterns, and page sizes limit flexibility in allocation sizes. It is also unclear that the complexity of a paged system is necessary. Paged systems are good for providing a flat, infinite address space unified with the CPU, but Apiary's primary goal is to provide isolation so that if an accelerator behaves in an unanticipated way, either maliciously or due to a bug, it cannot corrupt the memory of unassociated accelerators.

For simplicity and flexibility, we choose to do memory isolation via segments with capabilities in Apiary [15]. Segments allow more flexibility in the size of an memory allocation, reducing resource stranding, while capabilities give us isolation properties. Apiary is responsible for creating capabilities for memory regions. Capabilities are stored in a partitioned manner by having the Apiary monitor manage the capability

list, so the accelerator can only obtain a reference to the capability and not the capability itself. To enforce capabilities, the monitor interposes on every message and checks that the process has the correct capability to send to the destination.

5 Related Work

AmorphOS[25] and Coyote [26] are two closely related FPGA OSes. Both assume a hosted model rather than the standalone model used by Apiary. AmorphOS is an FPGA OS that dynamically recompiles FPGA bitfiles and uses partial reconfiguration to multiplex an FPGA between different applications. AmorphOS does not provide higher-level services or address inter-accelerator interactions. Coyote is more similar to Apiary by providing both higher-level services and multiplexing for applications, and it provides basic interprocess communication in the form of queues. However, it does not provide any isolation for those channels or discuss multi-application interactions in other subsystems. Every accelerator is attached to a specific CPU process on whose behalf it is acting, with permissions managed by the host OS.

Other work focuses specifically on hosting multiple users on a single FPGA. One [24] modifies Caribou [23], an FPGA-accelerated key-value store, to allow it to support multiple users. The resulting accelerator is able to support multiple connections, but only supports Caribou and does not generalize to other applications. SYNERGY is a framework for FPGA virtualization that uses static analysis in order to identify state that needs to be stored if an accelerator is paused at a given cycle, so it can be resumed seamlessly at a later time. SYNERGY only runs one application on the FPGA at one time, so does not address isolation for inter-accelerator interactions.

6 Open Questions & Challenges

Alongside our proposal for Apiary comes a number of open questions and challenges that come with realizing its implementation. We discuss some of the main open questions we have considered frequently:

What is the overhead of the per-tile monitor? One of the main implementation challenges will be managing the logic utilization of the Apiary. Much of the functionality and enforcement of isolation in Apiary is provided in the per-tile Apiary monitor. It is important for scalability that this monitor's resource utilization remain low since the amount of FPGA logic resources devoted to Apiary grows with the number of tiles. The overhead will also influence the flexibility of Apiary, because the number of tiles supported determines the granularity of logic within the tiles. More tiles means Apiary can support decomposing functionality into finer granularity, opening up more opportunities for reuse of functionality through composition of tiles. Apiary's design goal of simplicity for its abstractions will hopefully help keep resource utilization low, but ultimately we will have to find out via implementation.

What level of abstraction or specialization is needed? This question is important for all operating systems, but is especially important for those hosting accelerators. Accelerators gain much of their advantage from specialization to the application and specific low-level device details, so services that abstract away enough of the details to ease the developer experience, but that must be balanced with exposing enough of the details that the accelerator can specialize to the device. It is highly likely that different accelerators will need different levels of abstraction. Ideally, services themselves can be flexible enough to support, these different levels, but it may be possible accelerators require different implementations of the same service. We would like Apiary would be able to support multiple versions of the same service, but it is still a question of how to do this in a generalizable manner.

Can we reasonably completely avoid an on-node hosting CPU? Apiary aims to implement its primitives completely in hardware. This is to preserve efficiency gains of the direct-attached setting and potentially also provide better granularity on datacenter resource allocation since an FPGA could be provisioned independent of a CPU. However, it may not be worth implementing certain functionality directly in hardware if it is either rarely used or exceptionally complex. Ideally, we could take advantage of the network capabilities of Apiary and place the service on any remote CPU, maintaining the ability to use an FPGA independent of its on-node CPU. However, there may be cases this is not feasible. We are interested to see how far we can get in pure hardware.

7 Conclusion

Modern FPGA boards have advanced support for I/O and the capacity to support multiple accelerators, but FPGA developers must build a large amount of infrastructure to take full advantage of these devices. In this paper, we presented Apiary, a proposal for a hardware FPGA OS designed for the the complexity of modern FPGAs. Apiary is structured as a microkernel with message passing via a hardware NoC to support composability, flexibility, and scalability of hardware primitives. Apiary also defines an isolation and threaded execution model to address multiplexing needs. However, Apiary is an early attempt at defining a portable OS for standalone FPGAs, and there are many open questions around the right levels of abstraction or specialization. We hope this paper encourages others to also engage with these questions.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and feedback. This work was supported by grants from VMware Research, the National Science Foundation (CNS-2104548, CNS-2213387), the University of Washington Center for the Future of Cloud Infrastructure (FOCI), the Intel TSA center, and the NSF Graduate Research Fellowship.

References

[1] Achronix. [n. d.]. *Revolutionary New 2D Network-on-Chip.* Accessed: 2022-7-4.

[2] Alibaba. [n. d.]. *Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances.* Accessed: 2020-01-29.

[3] Amazon. [n. d.]. *Amazon EC2 F1 Instances.* Accessed: 2019-10-11.

[4] Amazon. 2021. *AQUA (Advanced Query Accelerator).* Accessed: 2025-1-4.

[5] Amazon. 2021. *Deep Dive on Amazon EC2 VT1 Instances.* Accessed: 2025-1-5.

[6] AMD. [n. d.]. *Alveo V80 Product Brief.* Accessed: 2025-1-5.

[7] AMD. 2023. *AMD Powers Alibaba Cloud FaaSwith AI Acceleration Solution for E-Commerce Business.* Accessed: 2025-1-5.

[8] AMD. 2023. *AMD Provides Twitch with Plug and Play VP9 Transcoding Solution for Live Video Streaming.* Accessed: 2025-1-5.

[9] AMD. 2023. *SmartSSD® Computational Storage Drive.* Accessed: 2025-1-5.

[10] AMD. 2024. *Versal Architecture and Product Data Sheet: Overview.* Accessed: 2025-1-6.

[11] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. 2023. FlexCNN: An End-to-end Framework for Composing CNN Accelerators on FPGA. *ACM Trans. Reconfigurable Technol. Syst.* 16, 2, Article 23 (March 2023), 32 pages. https://doi.org/10.1145/3570928

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09).* Association for Computing Machinery, 29–44. https://doi.org/10.1145/1629575.1629579

[13] Andrew Boutros, Mathew Hall, Nicolas Papernot, and Vaughn Betz. 2020. Neighbors From Hell: Voltage Attacks Against Deep Learning Accelerators on Multi-Tenant FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT).* 103–111. https://doi.org/10.1109/ICFPT51103.2020.00023

[14] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 1–13.

[15] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (March 1966), 143–155. https://doi.org/10.1145/365230.365252

[16] Colin Drewes, Olivia Weng, Keegan Ryan, Bill Hunter, Christopher McCarty, Ryan Kastner, and Dustin Richmond. 2023. Turn on, Tune in, Listen up: Maximizing Side-Channel Recovery in Time-to-Digital Converters. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23).* Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/3543622.3573193

[17] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th International Symposium on Computer Architecture, 2018.* ACM.

[18] K. Goossens, J. Dielissen, and A. Radulescu. 2005. AEthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers* 22, 5 (2005), 414–421. https://doi.org/10.1109/MDT.2005.99

[19] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (Dec. 2023), 31 pages. https://doi.org/10.1145/3609335

[20] Huawei. [n. d.]. *FPGA Development Suite.* Accessed: 2025-1-5.

[21] Intel. [n. d.]. *Intel Agilex 7 M-Series Hard Memory NoC Subsystem.* Accessed: 2025-1-9.

[22] Intel. 2024. *Key Features and Innovations in Agilex™ 7 FPGAs and SoCs.* Accessed: 2025-1-5.

[23] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1202–1213. https://doi.org/10.14778/3137628.3137632

[24] Zsolt István, Gustavo Alonso, and Ankit Singla. 2018. Providing Multi-tenant Services with FPGAs: Case Study on a Key-Value Store. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL).* 119–1195. https://doi.org/10.1109/FPL.2018.00029

[25] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* USENIX Association, Carlsbad, CA, 107–127. https://www.usenix.org/conference/osdi18/presentation/khawaja

[26] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[27] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. 2019. Mitigating Electrical-level Attacks towards Secure Multi-Tenant FPGAs in the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 12, 3, Article 12 (Aug. 2019), 26 pages. https://doi.org/10.1145/3328222

[28] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. 2023. Reconfigurable Virtual Memory for FPGA-Driven I/O. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3582016.3582048

[29] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J. Rossbach, and Eric Schkufza. 2021. Compiler-driven FPGA virtualization with SYNERGY. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21).* Association for Computing Machinery, New York, NY, USA, 818–831. https://doi.org/10.1145/3445814.3446755

[30] Andreas Lankes, Thomas Wild, Andreas Herkersdorf, Soeren Sonntag, and Helmut Reinig. 2010. Comparison of Deadlock Recovery and Avoidance Mechanisms to Approach Message Dependent Deadlocks in On-chip Networks. In *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip.* 17–24. https://doi.org/10.1109/NOCS.2010.11

[31] Microsoft. [n. d.]. *FPGA Web Service: Deploy Models on FPGAs.* Accessed: 2020-01-29.

[32] Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. 2006. Designing Message-Dependent Deadlock Free Networks on Chips for Application-Specific Systems on Chips. In *2006 IFIP International Conference on Very Large Scale Integration.* 158–163. https://doi.org/10.1109/VLSISOC.2006.313226

[33] NVIDIA. [n. d.]. *NVIDIA GPUDirect.* Accessed: 2025-1-4.

[34] Jin Ouyang and Yuan Xie. 2010. LOFT: A High Performance Network-on-Chip Providing Quality-of-Service Support. In *2010 43rd Annual*

IEEE/ACM International Symposium on Microarchitecture. 409–420. https://doi.org/10.1109/MICRO.2010.21

[35] George Provelengios, Daniel Holcomb, and Russell Tessier. 2020. Power Wasting Circuits for Cloud FPGA Attacks. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 231–235. https://doi.org/10.1109/FPL50879.2020.00046

[36] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. 2019. Network-on-Chip Programmable Platform in VersalTM ACAP Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 212–221. https://doi.org/10.1145/3289602.3293908

[37] Yao Wang and G. Edward Suh. 2012. Efficient Timing Channel Protection for On-Chip Networks. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. 142–151. https://doi.org/10.1109/NOCS.2012.24

[38] Hassan M.G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2014. Networks on Chip with Provable Security Properties. *IEEE Micro* 34, 3 (2014), 57–68. https://doi.org/10.1109/MM.2014.46

[39] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Shixin Ji, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Yiyu Shi, Deming Chen, Jason Cong, and Peipei Zhou. 2024. CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture. *ACM Trans. Reconfigurable Technol. Syst.* 17, 3, Article 51 (Sept. 2024), 31 pages. https://doi.org/10.1145/3686163