The Thesis Committee for Tiffany Ya Yang

Certifies that this is the approved version of the following thesis:

Supporting Compiler-Driven FPGA Virtualization

SUPERVISING COMMITTEE:

Christopher J. Rossbach, Supervisor

Simon Peter

Supporting Compiler-Driven FPGA Virtualization

by

Tiffany Ya Yang

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

August 2020

to my parents.

Acknowledgments

This work would not have been possible without the guidance of Dr. Christopher J. Rossbach. Chris, I truly appreciated all of your insights and Café de Cuba. Thank you for humoring me during our discussions and giving me opportunities to explain memes.

Joshua Landgraf also played an invaluable role in this work. Joshua, thank you for the time you took to answer my questions at all hours and for pointing me in the right direction for other open-ended questions.

I would also like to thank both Eric Schkufza and Ahmed Khawaja, whose work was critical to the implementation of the ideas presented here. Conversations with the other students in the Systems for Concurrent and Emerging Architectures lab provided helpful suggestions for my research.

Thanks to Dr. Simon Peter for your feedback and the discussions that have helped shape this thesis. I also appreciate you teaching me how to use the espresso machine, which added much-needed variety to my caffeine intake during the course of my studies.

My largest thanks go to my family, friends, and partners. Thank you all for your relentless support and encouragement; I certainly could not have done this without you. Finally, special thanks to Dany Haddad for dealing with formatting this template.

Supporting Compiler-Driven FPGA Virtualization

Tiffany Ya Yang, M.S.C.S.
The University of Texas at Austin, 2020

Supervisor: Christopher J. Rossbach

Many cloud providers now support on-demand FPGA acceleration in data centers. Though FPGAs can exceed general-purpose CPU performance by orders of magnitude, they introduce challenges that limit their popularity. Virtualization provides programmability and cost-effectiveness, two features that are critical to making FPGAs accessible as a mainstream acceleration technology. One recently proposed system to virtualize FPGAs is Synergy. Synergy provides suspend and resume, program migration, spatial and temporal multiplexing, and portability. These capabilities are enabled by extending features from the Cascade JIT compiler [52] and the AmorphOS runtime system [31]. Because Synergy provides compiler-based state capture, it does not require hardware support. This makes it deployable on devices available in data centers today. Using Synergy to virtualize an application comes with a performance cost of $3 - 4\times$ that of an unvirtualized application and an increase in FPGA resource utilization. These overheads are primarily due to the mechanism Synergy uses to manage each stateful value in the application. Many applications periodically reach quiescent points during execution, when their state is limited to a subset of stateful variables, which can reduce these overheads. We implemented a quiescence interface in both AmorphOS and Synergy. A benchmark with substantial in-flight state was modified to use the quiescence interface, and it suffers one order of magnitude less performance degradation during state capture. This work also discusses the challenges of adapting a deep neural network accelerator

on Synergy.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A Field Programmable Gate Array (FPGA) accelerator combines the functional efficiency of hardware with the flexibility of software. The ability to implement specialized instructions or algorithms allows hardware to scale under a different paradigm than software, which is scaled by increasing the number of operations executed per instruction (SIMD) or the number of cores. By definition, a general-purpose architecture has components that not all workloads will use. Rather than duplicating all of the elements used to implement a general-purpose processor, a hardware implementation of a function can scale with just the necessary circuitry.

As hardware acceleration is shown to achieve substantial performance benefits over general-purpose cores for more workloads, the case for making specialized hardware available to data center customers is strengthened. Still, in many cases, providing specialization in the form of fixed-function devices is an inadequate solution for modern cloud platforms. Data center servers must be provisioned to meet the ever-changing needs of diverse applications. The long development cycle and substantial cost of developing specialized accelerators are incompatible with how

quickly those needs may evolve. In contrast, the programmability of an FPGA makes it naturally suited to implement a specialized solution while retaining adaptability.

FPGAs can outperform general-purpose CPUs by several orders of magnitude [48, 15]. While the breakdown of Dennard scaling [20] has caused CPU clock rates to stagnate, reprogrammable hardware has provided improved performance in several domains including machine learning [61, 59, 62, 56, 17, 43], databases [40, 14, 29], finance [21, 35, 27], graph processing [19, 18, 47], and networking [48, 12, 25]. Compared to Application-Specific Integrated Circuits (ASICs), FPGAs can be developed more cheaply and with a faster time to market [33].

This ability to support diverse specialized applications with a single hardware technology makes FPGAs attractive for infrastructure providers. The appeal is further bolstered by economic factors; the past few decades have shown climbing FPGA compute density and clock speeds with plummeting cost-per-logic cell [31]. Now, FPGAs are becoming more common in commercial cloud platforms. Amazon provides F1 instances with large FPGAs attached and Microsoft deploys them in new data center construction [5]. Add-on cards are also available off-the-shelf from several hardware vendors. Meanwhile, FPGAs are becoming easier to use, with cloud providers offering libraries of popular accelerators to customers [7], and the emergence of higher-level hardware programming abstractions [54, 10, 9, 45].

Despite these advances, hardware development is still a challenging endeavor. Compared to software, the abstractions available to a hardware programmer are extremely limited. Hardware programs are usually targeted toward a specific device, relying on vendor-specific interfaces. As a result, adapting an application to another vendor's device usually requires learning another set of interface specifications. This

issue effectively locks a cloud provider into a relationship with a single manufacturer, as its customers' designs are not portable to other companies' devices. Consequently, creating a system that decouples FPGA application design from the underlying hardware has been the goal of much research. Virtualizing hardware interfaces is a critical component of any system that aims to integrate FPGAs into data center applications in a similar way to Graphics Processing Units (GPUs).

While the difficulty of programming FPGAs is an impediment to their widespread adoption, the cost is an additional factor. Currently, cloud providers manage their FPGA resources on a per-device basis, meaning a single customer has sole access to a specific physical FPGA for the duration of their application's runtime. This reality also makes using them rather expensive. Since the FPGAs being offered in the cloud have hundreds of thousands to millions of logic elements [2, 4], many workloads utilize only a fraction of the fabric. The cost to reserve the entire FPGA may be untenable. Virtualization enables more than programmability; it makes FPGAs a dynamically shareable resource.

Unfortunately, the industry lacks a standard technique to virtualize FPGAs. More specifically, there is no widely agreed upon method for *workload migration* (suspending and resuming the execution of a hardware program or relocating it from one FPGA to another mid-execution) or *multitenancy* (spatially or temporally multiplexing multiple hardware programs on a single FPGA). Because virtualization is the core technology that enables cost-effective operation of a data center, this is a serious obstacle to unlocking the potential of FPGAs as a mainstream accelerator technology.

For an operating system (OS) to effectively manage access to a system resource, such as an FPGA, it must be able to revoke that access. Virtualizing the

resource adds complexity because it typically implies that each user perceives exclusive access to that resource. For this to work, the OS must be able to save and restore the user's state on that resource. Efficient state capture is the main challenge for FPGA virtualization. For CPUs, software state is fully encapsulated in memory and a limited set of the processor's registers; the ability to save and restore virtual memory and the register file (and thereby state) is program-agnostic. The reprogrammable fabric that makes FPGAs so adaptable is also what makes them so challenging to virtualize, as an FPGA program's state is distributed throughout it in a program-dependent way. The only way to perform program-independent state capture of a hardware application is to access the entire register state of the FPGA. Along with scaling poorly with device size, this technique limits the fabric to a single application because all of the FPGA's registers are assumed to be part of its state.

Rather than attempting to support virtualization at the level of hardware, we propose a combined compiler/runtime hypervisor called Synergy. Synergy extends Cascade, the first Just-In-Time (JIT) compiler for Verilog, and AmorphOS, an FPGA runtime environment. Composing features from both into a single system provides all of the necessary infrastructure for virtualization. Synergy is able to leverage transparent state capture to provide both workload migration and multitenancy with OS-level guarantees such as process isolation, fair scheduling, memory protection, and cross-platform compatibility. Synergy's support for virtualization makes the benefits of hardware acceleration significantly more accessible by increasing both hardware design portability and cost efficiency.

These abilities make Synergy a very attractive system for managing FPGA applications in a data center context, but they operate with a performance gap of

$3-4\times$ that of an unvirtualized program and use $3-9\times$ as much area. The state access and execution control logic added by Synergy not only requires device resources to implement but can even prevent the use of specialized device resources in favor of more-contended generic ones. Even after accounting for these overheads, offloading computation onto reprogrammable hardware still offers a substantial performance boost over a standard CPU (about an order of magnitude instead of orders of magnitude). These gains may be enough to retain some users, but in order to for Synergy to address the needs of all cloud FPGA users, it must support diverse workloads and programmers with varying levels of hardware development experience. Consequently, we must expose a way for the application programmer to limit the overheads associated with automatic state capture and show that these techniques would work for real-world workloads.

For certain classes of workloads, there are points in their execution where only a subset of their stateful variables are representative of their state. We call these *quiescence points*, which represent application-specific consistent states. At a quiescence point, we can discard the values of variables disjoint from this subset without losing any information about the state of the application. This provides a natural way to reduce the number of variables that Synergy must track, thereby reducing both the communication between the hardware application and the runtime system and the FPGA resources necessary to support it. To this end, we provide a *quiescence interface* to allow the application to inform the system that it has reached a quiescent state and retain only the values of any variables that are necessary to continue execution. This optimization allows an application to be virtualized in Synergy with 30% less logic utilization and a $10\times$ performance speedup during hardware migration.

This work supports Synergy with two contributions, implementing support for a quiescence interface and accelerating Deep Neural Network (DNN) inference. After providing background, we discuss how each of these components contributes to our primary goal: demonstrating Synergy's suitability for virtualizing FPGAs in the cloud. Next, we describe how each component was designed, and we discuss the challenges faced during implementation. We then evaluate how these pieces work together with Synergy on Xilinx FPGAs running on Amazon's F1 cloud instances, and we consider how to address the limitations of our approach in the future.

Chapter 2

Background

2.1 FPGAs

FPGAs are circuits that can be reconfigured to implement custom logic. They can be deployed in a system in a number of ways. A *discrete* FPGA can be used on its own without a host processor, for example to implement a network switch, or may include in-silicon host processors to simplify management or communication with the FPGA. *Bump-in-the-wire* FPGAs can accelerate IO pipelines, e.g., to accelerate encryption for network cards. This work focuses on FPGAs attached to a system to serve as compute offload accelerators, as this is the configuration cloud providers use to provide on-demand FPGAs to customers.

In order to specify the behavior of an FPGA, a hardware developer usually writes a program in a Hardware Description Language (HDL). The two standard HDLs used to program an FPGA are Verilog and VHDL. We will focus on Verilog for the remainder of this discussion, but note that VHDL is essentially isomorphic.

We broadly categorize variables as either stateless or stateful. The value of a stateless variable is functionally dependent upon its inputs. Conversely, a stateful

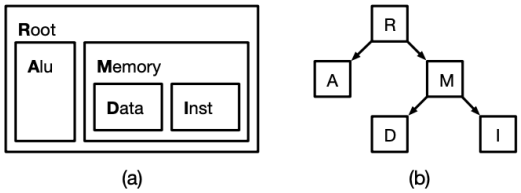

Figure 2.1: (a) A simplified CPU. (b) The hierarchical relationship between its components. This figure was adapted from Figure 2 in [34].

variable's value can be updated at discrete intervals. Typically, a Verilog `wire` type corresponds to a stateless variable, and a `reg` type corresponds to a stateful variable.

Verilog programs are written in units called modules, which are composed hierarchically. Typically, there is one top-level module called the root, which instantiates other modules. This relationship is illustrated in Figure 2.1, where each arrow in (b) represents an instantiation of the corresponding module in (a). Each module has a set of input and output ports, which specify how it interfaces with other modules. Within each module, there may be any number of arbitrary width wires and registers, logic gates, primitive arithmetic operators, and nested submodules that define its semantics. A module can access variables in another part of the hierarchy through a series of input/output connections. One key aspect of a hardware program is that there is no way to dynamically instantiate modules, storage, or connections.

A Verilog statement is synthesizable if it can be lowered onto an FPGA. The language also contains unsynthesizable constructs, including tasks such as print statements. These constructs make Verilog more debuggable and increase its expressive power, but their implementation also requires interaction with software. Consequently, unsynthesizable constructs are traditionally limited to use in simulation. Developers commonly write programs that target interfaces specified by

the FPGA's *shell*. A shell makes up a partition of each FPGA and implements vendor-specific library support for access to the IO capabilities of the device.

Once the Verilog program is complete, it must be converted into a binary called a *bitstream* to be deployed on an FPGA. The bitstream configures the FPGA's fabric to implement the design. The transformation from HDL to bitstream happens in two stages: *synthesis* and *place-and-route* (PAR). Synthesis translates the HDL into a *netlist*, a mapping of the resources and connections between them that are necessary to implement the design. Synthesis generally takes on the order of minutes to complete. Next, the netlist is routed on the fabric of the specific FPGA in the PAR stage. This step attempts to minimize the physical communication distance between connected resources while also respecting the spatial placement of those components. Observe that PAR is essentially a constraint satisfaction problem; in fact, both placement and routing have been shown to be NP-hard [51]. As a result, PAR takes on the order of hours for complex designs. The resulting bitstream takes on the order of tens to hundreds of milliseconds to be loaded onto the FPGA.

HDLs are used to design digital circuits, so a Verilog program gives the hardware programmer the ability to configure how the FPGA's heterogeneous resources are arranged and connected. The resources on an FPGA's fabric generally comprise look-up-tables (LUTs), flip-flops (FFs), on-chip memory (distributed and block RAM (BRAM)), interconnect, and other "hard resources", such as adders or digital signal processors. A single HDL design can be synthesized using a variety of resources. For example, a FIFO can be implemented either using BRAM or LUTs [32]. Similarly to a software program, a hardware program can be designed to take up more resources and require fewer cycles to complete than an implementation

with a smaller footprint. Unlike a software application, the state of a hardware application is distributed throughout its fabric.

2.2 Cascade

Cascade is the first JIT compiler for Verilog [52]. It was designed to improve hardware programmability by hiding the latency associated with compilation. This effect is achieved by starting the program in simulation and then lowering it onto the FPGA fabric once compilation is complete. Additionally, applications compiled with Cascade can execute unsynthesizable Verilog, even after being moved to hardware.

Along with making FPGAs more accessible to developers with limited experience with hardware programming, Cascade's implementation provides several benefits that are complimentary to FPGA virtualization. In order to target both hardware and software backends, Cascade provides the programmer with a virtualized interface to IO peripherals. This allows developers to design applications in a hardware-independent fashion, increasing portability. The mechanism Cascade uses to move programs from software to hardware enables transparent state capture.

Cascade's user interface is a Read-Eval-Print-Loop (REPL), which is similar to a Python interpreter. Each Verilog statement is lexed, parsed, and type-checked. If it passes those checks, it is immediately integrated into the user's program, at which point its side effects also become visible. Once the user is done writing the program and the corresponding bitstream has been compiled for the target FPGA, the user's application runs on the target at near-native performance.

Cascade supports unsynthesizable Verilog by applying transformations to the user's programs that produce code that can trap into the Cascade runtime at the

end of the logical clock tick. As discussed in §2.1, hardware cannot be dynamically instantiated from a Verilog program. Thus, Cascade can perform a computationally tractable static analysis to name every variable in a Verilog program at compile time.

Using this analysis, Cascade identifies every input, output, and stateful variable in a module. This set of variables make up the module's state. Cascade inlines module instantiations into a single user program with a combined set of state variables. As Cascade transitions the user program from software to hardware, it sends a `get` request to read the value of each of the variables in that set in software and then sends a `set` request to write each value to the corresponding variable in hardware before continuing execution in hardware. This state transfer occurs in between logical clock ticks once the program reaches a consistent state. The Cascade runtime retains the ability to read from or write to those variables in hardware or software, and that set of variables encompasses the state for the corresponding module.

2.3 AmorphOS

AmorphOS is an FPGA runtime infrastructure that enables sharing among different applications with strong protection and isolation guarantees [31]. It also provides a compatibility layer between each application and the cloud FPGA platform it is running on. Isolation, protection, and compatibility are furnished by the AmorphOS *hull*, which virtualizes the IO interfaces exposed to the application.

Additionally, AmorphOS supports high-degrees of multi-tenancy with spatial and temporal multiplexing among *morphlets*, a new abstraction for FPGA-based execution. A morphlet is an extension of a process that forms a protection boundary and encapsulates user FPGA logic. One key feature of AmorphOS is its ability to

transparently change mappings between user logic and the FPGA's physical fabric. This ability enables AmorphOS to increase fabric utilization by reducing fragmentation. Moreover, an application running on AmorphOS can scale dynamically in response to the FPGA's load and availability. If additional fabric is available on the FPGA, a morphlet can *morph* to a more performant form that uses more resources.

An application written for AmorphOS has an FPGA component and a host component. On the FPGA side, rather than writing FPGA applications for a specific shell, developers write them for the AmorphOS hull interfaces. The hull provides access to a control register (CntrlReg) interface for managing morphlets, simple PCIe for bulk data transfer, and the AmorphOS Memory Interface (AMI) for accessing data on the on-board DRAM. The AMI also gives morphlets the perception that they have full control of the available memory. On the host side, a process can instantiate or evacuate, send a control signal to, or conduct a bulk data transfer with the corresponding morphlet by sending a request to the AmorphOS scheduler. The scheduler manages that morphlet's mapping onto FPGA resources, and AmorphOS handles communication between the two. This scheme ensures that any user code written for AmorphOS is portable to any of the platforms it supports.

The hull mediates access to on-board memory and IO. It also exposes a simple virtual memory system in the form of the AMI. Memory protection is implemented within the hull using segment-based address translation. These features are necessary to enable mutually distrustful morphlets to share the FPGA.

Demand-sharing is implemented through the AmorphOS scheduler. The scheduler first attempts to share the fabric among all of the morphlets, but if resource constraints preclude spatial sharing, then the system falls back to time-slicing execution on the FPGA. Similarly, if the FPGA is underutilized, the scheduler can

remap the running morphlets onto a configuration that takes up more of the fabric. AmorphOS hides the latency of retargeting a morphlet to another set of the FPGA's physical resources with a cache of pre-compiled bitstreams called the *registry*. Note that loading a new bitstream onto the FPGA resets its state. Accordingly, time-sharing and resource remapping require some means of state capture for the morphlets running on the currently deployed configuration. AmorphOS addresses this problem with a quiescence interface, which allows a morphlet to signal that it can be evacuated from the FPGA without losing state. While a quiescence interface is assumed in AmorphOS's design, the original system provides limited support for it.

Host-side user programs interact with the scheduler through a system call interface, which is implemented as a system service. When a client instantiates a morphlet, it receives a file descriptor. The service associates all metadata about the morphlet's state with this handle.

2.4 Synergy

Synergy is a hypervisor for FPGAs [34] that enables a hardware program to be suspended and resumed or relocated to another FPGA mid-execution. It also supports spatial and temporal multiplexing on a single FPGA. While Synergy can be thought of as Cascade with an AmorphOS backend, this combination unlocks more functionality than either system can provide alone. Synergy provides transparent state capture for hardware programs, making efficient context switching and over-subscription possible. Furthermore, the virtualized interfaces provided by Synergy effectively decouple application design from the underlying hardware. These qualities make Synergy a very attractive solution for virtualizing FPGAs in a data center context.

Synergy's front-end includes several updates to Cascade that further support virtualization. For example, along with virtualized interfaces to IO peripherals, it also provides a virtualized file IO interface as a class of unsynthesizable constructs. As with Cascade, programs compiled within Synergy's framework can execute unsynthesizable constructs, even when they are running on hardware.

Consider a single instance of Cascade. A developer uses that instance to define and instantiate several modules, which Cascade then compiles into a single user program for the target backend. Synergy serves as a hypervisor for multiple vanilla Cascade instances. It provides an indirection layer that allows those instances to share a single compiler at the hypervisor level. Each of these instances is, in turn, embedded into AmorphOS as a morphlet, enabling multitenancy with strong protection and isolation guarantees.

This composition is made possible by specifying Synergy as the backend target for each of the Cascade instances. Compilation for a Synergy backend is implemented by sending the user source code to Synergy. Synergy combines the user program associated with each of the instances into a bitstream for the specific FPGA fabric managed by the Synergy hypervisor. Since the Synergy virtualization layer nests, one hypervisor can delegate the hypervisor for a different FPGA as the backend a user program that will not fit onto its own fabric. The user interface for Synergy is the same REPL as that for Cascade.

Synergy is able to transparently capture each instance's state using the same mechanism described at the end of §2.2. This allows the Synergy scheduler to be more aggressive about reconfiguring the FPGA the AmorphOS scheduler, enabling fairer time slicing, as illustrated by the following example. Suppose the hypervisor has just finished compiling a bitstream to accommodate a Cascade instance for some

user program X. It can evacuate the state of each of programs currently running on the FPGA it manages, reconfigure the fabric, read each program's state back in, and continue execution. Once X finishes executing, this process can be repeated to reprogram the FPGA with the previous bitstream, which is available in the registry. Conversely, under an analogous setup for vanilla AmorphOS, the scheduler would have to wait for each of the resident morphlets to reach a quiescent state or risk discarding state. This condition could cause a significant increase in the turnaround time for X.

2.5 DnnWeaver

DNNs have shown impressive prediction accuracy in a variety of applications, such as computer vision and speech recognition [3], but they require substantial compute and memory capabilities. ASICs have enabled further advances in DNN technology, but those benefits come with high non-recurring engineering costs. DnnWeaver [54] is a framework that allows users to translate a high-level specification for a DNN architecture into synthesizable Verilog. FPGAs provide an appealing means of accelerating DNNs, which are very compute-intensive. Still, DNNs also have substantial memory requirements. The limited on-chip memory and off-chip bandwidth of FPGAs make generating FPGA accelerators for DNNs a non-trivial task. Another significant challenge with FPGAs is designing hardware that is both performant and energy-efficient. DnnWeaver was designed to address these challenges with a scalable and reusable acceleration framework. Additionally, the creators of DnnWeaver aimed to reduce the time and expertise required to accelerate DNNs on an FPGA, which aligns with our goals for Synergy.

The application programmer defines a DNN architecture using Berkeley

Caffe. Next, that architecture is translated into macro dataflow ISA. These instructions are then mapped onto a hardware template architecture design. At each layer, computation is partitioned into groups of operations that share and reuse data. The output of each group is called a slice. These slices are used to generate an execution schedule, and they determine how the model is laid out in memory.

The architecture is organized into several independent processing units (PUs), each made up of a set of smaller processing engines (PEs). Organizing computation into PUs enables the framework to use a Design Planner to schedule data transfers to each PU that maximize bandwidth utilization. The accelerator produced by DnnWeaver uses several additional strategies to adhere to the target FPGA's resource constraints. To reduce data communication, partial results are stored locally. Similarly, one-way communication links are added in between adjacent PEs to forward shared inputs. DnnWeaver hides the latency of pooling operation execution by overlapping it with convolution. A heuristic search algorithm generates the architecture and static execution schedule that minimize off-chip memory accesses and maximize performance for the target FPGA. Finally, memory interface code for the target platform is incorporated into the accelerator code.

Unlike previous research to accelerate specific DNN models or components of DNN computation for a specific platform, DnnWeaver is an accelerator generator. DnnWeaver differs from high level synthesis (HLS) in that the hardware implementation produced from the high-level programmer abstraction is optimized for a target FPGA. Each output is for a given architecture, FPGA pair. The accelerators generated by DnnWeaver outperform CPUs and are able to provide greater performance-per-watt compared to GPUs.

2.6 Related Work

There are many works that explore different aspects of OS-level support for FPGAs. Chen et al. [16] developed an online scheduling algorithm for sharing an FPGA with multiple application-specific instructions implemented in its fabric among multiple cores. Like Synergy, this work considers sharing portions of the FPGA's fabric among various tasks. Unlike Synergy, this scheduler must account for the additional constraints of a task's arrival time and deadline.

Many works combine several operating systems features to extend host OSes to create process-like analogues on FPGAs. In order to ease the process of HW/SW codesign, BORPH extends the Linux kernel to use hardware processes [55]. These hardware processes use the same UNIX interface and inherit the same services as software processes. They also have access to system resources like standard streams and the general file system. Communication between hardware and software is carried out using pipes, shared files, signals, and the other standard UNIX IPC mechanisms. Consequently, a hardware process is effectively the same as a traditional UNIX process, but its "program" is an FPGA application. Unlike Synergy, which constrains the hardware program to Verilog, an FPGA application developed using BORPH can use any hardware language. Other similar works include ReconOS [41], HybridOS [30] and FUSE [24].

Several works consider temporal multiplexing. Many earlier context-switching solutions rely on Xilinx's Internal Configuration Access Port (ICAP) interface to read columns of an FPGA's configuration memory, which includes current RAM contents and register values. While this method can be used to obtain the state of any application running on the FPGA, without additional refinement, it has very poor data efficiency, as a relatively small portion of the readback data comprises

state [28]. Attempts to reduce the portion of columns to be read in include [28] and [37]. Kalte et al. [28] suggest a "shutdown process" to transition a hardware task into a state where fewer registers constitute state in order to reduce the area overhead of a communication infrastructure to read and write values, i.e a quiescent state.

Like Synergy's `get/set` requests, Ullman et al. [57], Mignolet et al. [44], Huang et al. [23], and Puttegowda et al. [49] employ an interface to save and restore state registers, but they all require the hardware applications to reach an interruptible or idle state before being swapped out. Bourge et al. [11] enable context-switching using static analysis on a hardware application's finite state machine to determine "hardware checkpoints" that occur within a certain latency constraint and minimize the cost of saving and restoring state. Hardware checkpoints correspond to suitable execution points for quiescence to be asserted.

Rupnow et al. [50] consider three policies for hardware application preemption, block, drop, and rollback. For each of these policies, the hardware application is considered to be accelerating a software thread, so preemption occurs when its corresponding thread loses control of the CPU. Of the three policies, only rollback allows the portion of fabric corresponding to a stalled hardware application to be reconfigured for a running thread. This work does not consider capturing the state of the hardware application. Levinson et al. [38] identify the need to have complete clock control of the FPGA to suspend a task, and they use a client-server architecture as a proof-of-concept to allow multitasking with preemption on an FPGA.

Many systems rely on partial reconfiguration (PR) to provide spatial sharing of an FPGA among multiple applications. This technology makes it possible to reconfigure a specific portion of the fabric without affecting the state of the rest of

the FPGA. Since the set of resources corresponding to each partially configurable partition must be specified ahead of time, systems that use PR to provide sharing must section the FPGA into fixed-size slots. Along with increasing the potential for fragmentation, using fixed-size slots constrains application size to the size of the largest slot. Both of these drawbacks limit the diversity of workloads this approach can support. Moreover, the flexibility of these systems is further limited by their reliance on hardware support for PR. Traditionally, PR is difficult to use. In some cases, a cloud platform's build system may even interfere with the PR compiler flow. Consequently, most public cloud vendors do not expose PR to users.

The LEAP OS is structured around the assertion that communication is the fundamental abstraction required to provide OS-like features to FPGA programs. This assertion allows the system to build off of latency-insensitive design, presented by Carloni et al. in 2001 [13]. LEAP uses latency-insensitive channels as a primitive. These channels, which are essentially register-transfer level (RTL) FIFO queues, guarantee FIFO message delivery and at least one in-flight message at any time. LEAP leverages this programming model to allow modules to be distributed among FPGAs, manage CPU-to-FPGA communication, and virtualize physical device interfaces, making applications portable to any FPGA platform that LEAP supports.

LEAP's compiler interface allows the system to provide service libraries for which automatic resource management decisions can be made at compile time. Together, these features enable LEAP's scratchpad memory [6]. The latency-insensitive channels serve as abstract interfaces to memory, which LEAP's compiler automatically groups into a cache hierarchy made up of the platform's available SRAM, DRAM, and host virtual memory. RIFFA [26], HThreads [8], ClickNP [39], Blue-

spec [46], and CMOST [64] are other systems centered around providing resource management to FPGAs.

FOS is an operating system that aims to modularize each stage of FPGA development [58]. The authors propose abstracting each hardware and software layer into a stack of standardized APIs, allowing components to be swapped at one layer without requiring recompilation or redesign at the others. One notable feature of FOS is the decoupling of the compilation of an FPGA shell and its user logic. Spatial multiplexing is enabled through PR, and while a daemon schedules requests such that multiple applications can share a single accelerator, each request must run to completion before another application can be scheduled.

ViTAL is a recently proposed full-stack solution to provide cloud FPGA virtualization. Its compiler tool transparently partitions application logic into homogenous virtual blocks, with each block representing an identical allotment of resources, backed by a physical block. At runtime, the runtime system maps each of the application's virtual blocks onto a physical block, which may be on any of the FPGAs being managed by ViTAL. Since an application's virtual blocks may be mapped to multiple FPGAs, the system must be able to handle bandwidth mismatches and latency from communicating across FPGAs. This communication is handled by the inter-block interface, which enforces functional correctness. This interface is modeled after the latency-insensitive communications channels implemented in the LEAP operating system [22]. Along with on-chip resources, ViTAL provides peripheral virtualization.

Unlike Synergy, ViTAL does not provide a state capture mechanism, though the system's performance would benefit from adding this feature. Given the substantial latency of inter-FPGA communication as compared to communication between

blocks on the same FPGA, an ideal runtime scheduling algorithm for this system would attempt to map virtual blocks from the same application onto the physical blocks of one FPGA. Without being able to save state, an application with physical blocks on multiple FPGAs would have to run to completion in that configuration, even if enough physical blocks to accommodate it on a single FPGA become available. For a system managing many applications, this fragmentation poses a significant limitation on performance.

In representing heterogeneous resources as homogenous virtual blocks and virtualizing peripheral devices, the authors of ViTAL identify abstracting the underlying device as a key aspect of system support for FPGAs. This design philosophy aligns with Synergy. Still, one significant difference between ViTAL and Synergy is that the authors consider scale-out acceleration with multiple FPGAs as a major limiting factor to their widespread adoption. Thus, its design is optimized around this case.

Synergy was not designed around this case for several reasons. First, as the authors of the paper pointed out, FPGA capacity is growing substantially over time. This phenomenon implies that the FPGAs being equipped in data centers are also growing over time. One argument for adopting this design is that it enables large applications to be deployed in data centers that were originally equipped with older, smaller FPGAs. This is certainly a compelling argument, but ViTAL's virtual blocks rely on hardware support for partial reconfiguration. As discussed above, most public cloud providers might not expose that capability to users, as is the case with Amazon F1. Because we consider the substantial price of one FPGA to be a major factor limiting their accessibility, Synergy is designed for applications that are either smaller than or the same size as the FPGAs they run on.

ViTAL, like Synergy, can be characterized as a system that virtualizes FPGA reconfiguration. Despite considering a similar use case, ViTAL does not enable context switching of an FPGA. One recent work that targets both spatial and temporal multiplexing of an FPGA is Optimus [60].

Optimus is a hypervisor that supports shared-memory FPGA virtualization. Here, the authors distinguish a host-centric FPGA programming model from a shared-memory one. Host-centric virtualization support requires all direct memory accesses (DMAs) to be issued by the host, which then passes the data to the relevant application. For accelerators that exhibit pointer chasing, the constant CPU to FPGA communication incurs large performance costs. A shared-memory programming model allows each FPGA application to issue DMAs without going through the host while still sharing an address space with a process on the CPU. Optimus virtualizes IO using page table slicing, which is implemented as a hardware-software co-design. Oversubscription of specific accelerators is enabled with a preemption interface, similar to the quiescence interface discussed in this work, that allows virtual accelerators to write their state to or from system memory. Optimus is designed under a different use case than Synergy. Namely, Optimus assumes FPGAs that are pre-configured with a subset of the accelerators offered by the cloud provider.

Similarly to Optimus, Feniks aims to reduce communication between a host CPU and the FPGA [63]. The FPGA shell is extended to implement an OS that contains modules to enable user applications to communicate with devices such as other FPGAs, coprocessors, storage, and NICs over PCIe. Feniks virtualizes IO devices and their stacks to enable sharing.

Chapter 3

Goals

3.1 Motivation

For Synergy to support the full range of data center customers, it must give developers the flexibility to optimize for different workloads. This includes programs that reach quiescence points, where their execution state is fully encapsulated by the values of a subset of their stateful variables. In many environments, workload migration takes place on a flexible schedule, making it both possible and preferable to wait for a quiescence signal if it allows systems to avoid state-copy altogether. Additionally, many FPGA applications make use of pipelines, which may also retain excess state. Application developers may prefer to recompute values along a pipeline rather than save and restore them from system memory. Since automatic state capture is a major contributor to the overheads of using Synergy over a native application, we identified support for quiescence as an important avenue for optimization. Finally, because Synergy embeds programs into AmorphOS, the infrastructure to support quiescence in AmorphOS must also be fully implemented.

Not only does quiescence allow faster context switching, but it also creates an

opportunity for hardware applications to access the Synergy's benefits (and incur the associated overheads) intermittently. For example, imagine a complex program that executes many cycles before encountering a bug. With a unified quiescence interface between AmorphOS and Synergy, this application could execute on AmorphOS at native performance speeds before signaling quiescence. Then, that minimal state could be transferred to an implementation with Synergy's instrumentation, allowing a developer to debug the application directly using the unsynthesizable constructs supported by Cascade.

3.2 Programming Model

Similarly to both Cascade and AmorphOS, the programming model that we target is HDL over an abstract FPGA fabric. However, programming for the Cascade frontend to Synergy is substantially different from the interface exposed by the AmorphOS hull, so the models used to support quiescence at both levels differ as well.

Previously to this thesis, the functionality offered by AmorphOS and Cascade's virtualized interfaces did not fully overlap, so some programs may be easier to implement on one system over the other. Since we are also adapting our realistic workload to use both the AmorphOS and Synergy quiescence interfaces, we have the opportunity to see how the two different models affect the application programming experience as a whole.

3.2.1 AmorphOS

While AmorphOS provides virtualized interfaces that make it easier to port applications to different platforms, these interfaces preserve much of the functionality

provided by the shells that hardware programmers are familiar with. An application developed for AmorphOS communicates with the host and accesses on-board memory by programming to these interfaces, specifically CntrlReg, PCIe, and AMI. While the CntrlReg interface allows morphlets to send data in response to a request from the software client, AmorphOS does not provide hardware applications with a way to initiate a request to the software.

3.2.2 Synergy

On the other hand, the interface virtualization provided by Synergy is much farther removed from the hardware than that of AmorphOS. Rather than expose channels through which a software application can send data or control to the hardware program, the Synergy model favors building interactivity into the program directly via unsynthesizable Verilog. Rather than responding to requests from software, programs developed for Synergy tend to operate on one batch at a time. Data can be moved between the FPGA program and the system through virtualized File IO constructs. These constructs greatly simplify the process of writing applications that stream data.

3.3 Requirements

3.3.1 Quiescence Interface

For a quiescence interface to increase the utility of Synergy, it must be easy for the application programmer to use. Its implementation must also be relatively lightweight, given that its main purpose is to reduce overheads. Since Synergy workloads perform batch computations instead of responding to specific requests, a Synergy application need not implement support for refusing requests in order

to stay in a consistent state. Consequently, the Synergy quiescence interface only needs to support communication in the direction of the application to the system. This design solution would not be as suitable for an AmorphOS application because AmorphOS does not provide a way for the hardware program to send a request to the corresponding software client. Finally, the quiescence interface must actually improve the performance of applications that use it.

3.3.2 Workload Considerations

As touched on before, the workload that we adapt for Synergy must be representative of the type of hardware acceleration that a data center application would ostensibly want to use an FPGA for. Along those lines, an optional requirement would be for the application to make use of one of the higher-level abstractions that aim to make hardware programming more accessible. Additionally, we want to use this workload to demonstrate the potential for optimization posed by the quiescence interface. Thus, we want to choose an application for which quiescence is relatively easy to identify.

Chapter 4

Design

4.1 Design Space Exploration

4.1.1 Quiescence Interface

We considered three designs for providing quiescence support in AmorphOS. The first option would be to use the CntrlReg interface exposed to all morphlets but to reserve the top two addresses in each morphlet's virtual address space: one to indicate a request for quiescence, and the other to check if the morphlet has quiesced. This design has the benefit of being very lightweight, but it is more difficult for applications to use. It would require applications to complicate the logic they use for standard CntrlReg requests and responses, and could introduce communication latency for morphlets using the CntrlReg interface to communicate with the host every cycle.

A second option would be to create a small application that accepts control register requests and maps each address to a specific morphlet. This application would only be exposed to the AmorphOS runtime, and it would manage quies-

cence requests and responses for each of the morphlets on the fabric. Although this approach would consume more resources than the first option, it enables user applications to respond to CntrlReg requests and quiescence requests in the same cycle.

The third option we explored would be to attach a new interface to a PCIe Base Address Register (BAR) in a similar fashion to the CntrlReg interface. This design would likely entail less resource utilization than the second option, and it would also avoid complicating the user's logic for managing CntrlReg requests and responses. One substantial drawback to this approach is that platforms expose a limited number of PCIe BARs to their customers. On Amazon F1, three out of the four BARs available to the user application are already used by AmorphOS; using the last one to implement quiescence could limit design choices for a new interface with heavier traffic later on.

As discussed in Section 3.2.2, Synergy does not expose a natural boundary to send a request from the runtime to the user application. The virtual clock, and other virtualized IO peripherals, are made available as modules instantiated in the root; the programmer can choose to interact with them or not. One option that we had considered to send a quiescence request through Synergy would be through a quiescence module. Ultimately, along with providing an unclear benefit, this module would be a confusing addition to the system as a whole.

Synergy's compiler frontend makes using annotations a natural way for the developer to identify which variables constitute state. This still raises the question of whether an annotation should indicate that a variable's value be saved or ignored. Since we postulate that a quiescence interface will be most useful (and used) when state is relatively small, our design requires annotations on the variables to be kept

as state to reduce the work required by the programmer.

4.1.2 Workload

First, we wanted to highlight a workload for which FPGAs offer unique benefits compared to other options, such as GPUs. Three such options are genetic sequence alignment [53], high-frequency trading [35], and single-inference DNN acceleration [3]. We further narrowed the specification to applications developed using hardware-acceleration frameworks, like OpenCL, which we expect to see more of because of the ease of use. Finally, we decided to choose a workload that could be organized as a request-response style program. This structure is especially well-suited to benefit from the ability to assert quiescence because these applications may have substantial in-flight state that can be ignored in between serving requests.

4.2 Implementation

4.2.1 Quiescence Interface

Ultimately, we chose to implement quiescence design presented in option 2, a hardware application. The ability to use existing infrastructure without affecting the responsiveness of other morphlets made it an appealing choice over the first option. Similarly, our desire to retain the flexibility to implement a different feature to the exposed BAR made this design more appealing than option 3. Since the quiescence application uses AmorphOS's existing infrastructure, it has the added benefit of being immediately portable to both platforms AmorphOS supports.

Within Synergy, we implement the quiescence interface as a system task called `$yield` in a similar fashion to the other unsynthesizable constructs. An example of its usage is shown in Figure 4.1. Programs that reach a quiescent state can

```
1: module Root();
2:   (* non_volatile *) reg[31:0] x;
3:   reg[31:0] y;
4:   always @(posedge clock.val)
5:     if (...) $yield;
6:   // Additional program logic ...
7: endmodule
```

Figure 4.1: Verilog program using Synergy quiescence interface. The `$yield` task enables Synergy's quiescence interface. Stateful variables are assumed volatile by default and must be managed by the user.

assert `$yield`. When present, stateful program variables are considered *volatile* by default, and their values are ignored. Developers can override this behavior by indicating the variables that represent non-volatile state with a `(* non-volatile *)` annotation. When a program reaches a yield point in its execution, it traps back into the runtime system, which can then move the application off of the FPGA, or allow it to continue execution in hardware.

4.2.2 DnnWeaver

We chose DnnWeaver because it provides the added benefit of comparability; it was one of the benchmarks we included in the original discussion of AmorphOS [31]. We used an 8-layer LeNet [36] topology. The top level of DnnWeaver is implemented as a state machine, so we are able to assert quiescence when it completes an inference and returns to an idle state. This enables it to adapt to AmorphOS's quiescence interface easily, and ease of use was a key principle of our design.

DnnWeaver is a memory-intensive application. It writes the results of from each layer to specific addresses and then references them to perform the computations of each subsequent layer. One major challenge in adapting DnnWeaver for Synergy was that Synergy does not have a virtualized memory interface like Amor-

phOS. In order to allow the driver to interact with byte-addressable "memory", we used Synergy's file IO capabilities. We created a file to represent the application's memory in the host file system. Each address passed from the memory controller was used as the offset into that file from which to read and write the corresponding data to.

Despite being chosen as a workload because of the programmability it would offer, the Verilog output by DnnWeaver presents a significant challenge to port to Synergy. Since designs were coupled to specific FPGAs, they were also written to use vendor's specific shells. The work done by DNNs requires substantial access to memory and compute, and adapting the AXI interface protocol to use the constructs provided by Cascade made DnnWeaver an especially difficult application to virtualize with Synergy. There was no reference behavior to validate against. Moreover, the lack of documentation associated with the Verilog generated using DnnWeaver made it difficult to infer the expected behavior of some of its components.

In one example, the inference would hang because the main processing unit for the network pauses reads until the number of writes from the memory controller reaches the value associated with the current layer of the network. Each write is directly dependent on incoming data from the reads, so once the reads are throttled, data stops being pushed to the write queue. As a result, the total writes never increases to the specified number for the first layer. Still, the deadlock caused by this interaction appears to be a correct outcome as specified by the Verilog and associated parameters. Without further information about the expected behavior, it was difficult to debug. Our efforts to address these challenges were not sufficient to port DnnWeaver to Synergy in time for the publication of this work, but through this process we were able to make several improvements to Synergy's file IO interface

and address sources of unexpected behavior in Synergy that had not been discovered developing smaller applications.

Chapter 5

Experimental Evaluation

We evaluate the Quiescence Interface as implemented on AmorphOS and Synergy on Amazon F1 cloud instances. The F1 instances support multiple Xilinx UltraScale+ VU9Ps running at 250 MHz and four 16 GB DDR4 channels [7]. Synergy uses build tools adapted from the F1 toolchain and communicates with the instances' FPGA fabric over PCIe. The hardware was synthesized using 64-bit Vivado 2019.2.

5.1 Methodology

We wanted to evaluate how well the chosen implementation of the quiescence interface meets the goals laid out in Chapter 3. We established that the quiescence interface should be lightweight and easy to use, along with actually reducing performance overheads. To this end, we measured both the area overhead and performance impact of our implementations of the quiescence interface.

More specifically, lightweight means that our implementation should take up relatively few of the available FPGA resources. In order to test this quality, we wanted to compare the difference in resource utilization between AmorphOS bit-

streams built without the quiescence interface to those built with it. Bitstreams *without* quiescence support were built with the original implementation of AmorphOS and no quiescence management in the application. Bitstreams *with* quiescence support have hardware for the quiescence application as well as the wires connecting it to the rest of the system. We also tested how the implementation scaled by building bitstreams with increasing numbers of morphlets.

Similarly, we built a Synergy application with and without support for quiescence. Using the Vivado build tools provided by Xilinx, we observed how the resource allotment changed.

It was also important to determine how the quiescence interface can affect the performance overheads introduced by Synergy. These overheads can be measured using the length of a virtual clock cycle, as longer cycles point to more time spent executing in software before control returns to the FPGA. Here, we want to observe how the performance of an application degrades during a context switch, and how asserting quiescence changes that performance. Using similar guidelines to those presented in Section 3.3.2, we chose an application that has significant in-flight state between quiescence points for this experiment.

The workload used in this measurement implements a MIPS processor randomizing then performing bubble sort over a large in-memory array. This workload is typical of long-running batch computations which are coalesced to improve data center utilization. Quiescence can be asserted in between each sort, at which point only the instruction memory, register memory, and program counter need to be saved, and all data memory can be discarded. In this experiment, we transition from one physical F1 FPGA to another, continuing execution in software during the migration. Since one instruction is executed each virtual cycle, we are able to

Configuration	LUTs	FFs	RAMs
1 Morphlet	115,295	94,720	4,078,848
1 Morphlet + Quiescence	115,316	97,341	4,076,288
3 Morphlets	145,637	104,239	5,024,512
3 Morphlets + Quiescence	144,819	104,312	5,085,952
7 Morphlets	209,033	123,399	7,140,608
7 Morphlets + Quiescence	209,891	123,459	7,138,048

Table 5.1: Total AmorphOS FPGA resource utilization with and without quiescence. Here we list the total number of look-up tables (LUTs), flip-flops (FFs), and memory bits (RAMB36, RAMB18, URAM, LUTRAM) taken up by user logic by AmorphOS with and without quiescence.

use the virtual clock cycle length to track how the number of instructions executed per second varies when migrating using automatic state capture and quiescence.

5.2 Results

5.2.1 Resource Utilization

Here, we present the low-level logic, register, and memory resources taken up by implementing quiescence in AmorphOS and Synergy. Logic and register utilization are reported as LUTs and FFs, respectively. On-chip memory usage is reported as the combined total of BRAM and distributed RAM (LUTRAM) bits for each design. We report memory utilization as a bit value because each type of memory has a different size.

Table 5.1 shows the total resource utilization for AmorphOS with and without quiescence as the number of morphlets it is managing increases. Note that for each of the three values of embedded morphlets, implementing quiescence increases the number of FFs used to synthesize this design, but the percent increase differs substantially from case to case. For 1 and 7 embedded morphlets, the number of

Application	LUTs	FFs	RAMs
Quiescence with 1 morphlet	173	56	4,096
Quiescence with 3 morphlets	220	56	4,096
Quiescence with 7 morphlets	257	56	4,096
DnnWeaver	5,566	4,513	266,240

Table 5.2: FPGA Resource utilization of quiescence application and DnnWeaver broken down by resource type. Here we report the resources belonging specifically to the quiescence application to support up to 7 morphlets. We also provide the resource utilization of a single DnnWeaver morphlet to contextualize these values. All values as reported by the Vivado build tools.

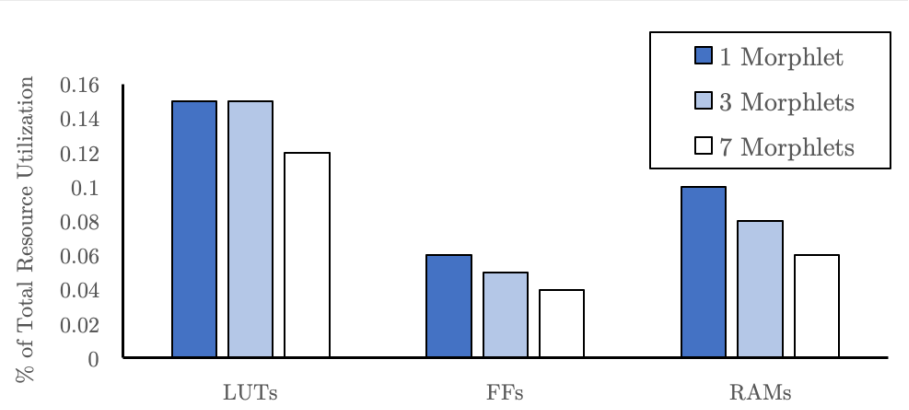

Figure 5.1: FPGA Resource utilization of quiescence application compared to total user logic. Here we report the resource utilization taken up by the quiescence application as a percentage of the total resource utilization of AmorphOS with varying numbers of DnnWeaver morphlets embedded.

LUTs to support quiescence increases, while the bits of RAM decreases. The opposite is true for 3 embedded morphlets. This discrepancy can be attributed to the fact that the same logic can be synthesized using different resources, as discussed in §2.1. These data were presented to establish that our implementation of the AmorphOS quiescence interface results in less than a 1% increase in utilization for each type of resource.

Table 5.2 shows the resources allotted specifically to the quiescence appli-

Configuration	LUTs	FFs	LUTRAMs
MIPS	14468	13976	136
MIPS + Quiescence	9612	13196	136
% Difference	31.2	5.6	0

Table 5.3: FPGA resource utilization of Synergy with MIPS workload with and without quiescence support. For both configurations, Synergy supports a single MIPS application.

cation when AmorphOS is built with varying numbers of morphlets. As the application scales to manage quiescence for additional morphlets, only the number of LUTs increases. Even when supporting 7 morphlets, the quiescence application can be synthesized with orders of magnitude fewer resources than a standard morphlet. Similarly, Figure 5.1 illustrates that as the quiescence application scales to accommodate more morphlets, it still makes up less than 1% of the total utilization for each type of resource.

Table 5.3 shows the decrease in resource utilization when quiescence is implemented in the Synergy MIPS application. Although the application requires hardware resources to implement quiescence, i.e., to check if it is at a quiesced state and to yield, the overall fabric utilization decreases. Note that in Table 5.3, memory utilization is reported as LUTRAMs instead of RAM bits because the bitstreams for the MIPS workload do not use block RAM.

5.2.2 Hardware Migration

Here we report performance measured by throughput.

Figure 5.2 plots the performance of the single-cycle MIPS bubble sort described in §5.1. The curves show the execution of two versions of the MIPS benchmark, one with no quiescence logic, and the other with quiescence. The timing of key events is synchronized to highlight the differences between the two versions. In

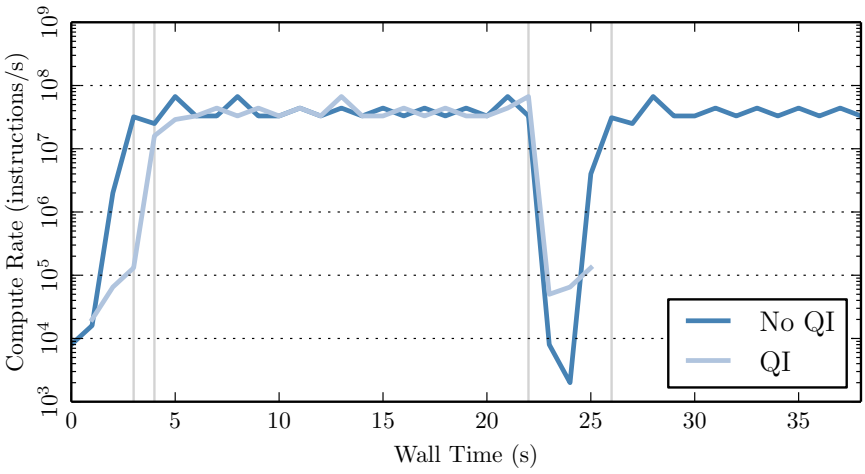

Figure 5.2: Hardware migration of MIPS benchmark with and without quiescence. A program begins execution on one target and is migrated mid-execution to another.

both cases, control begins in software, and transitions shortly thereafter to hardware ($t = 3, 4$). MIPS is able to achieve peak throughput of 10^8 instructions per second. At ($t = 21$), we emit a signal that causes both contexts to evaluate a `$save` task as the program is moved between FPGAs. Control then transitions temporarily to software as the runtime evacuates the program's state using `$get` requests. Here, the performance degradation during hardware/software transitions can be observed in the steep drop-off in throughput after retargeting.

Compared to the version with quiescence, MIPS without quiescence experiences more than 10× the performance degradation. This is a result of the large amount of state that must be managed by `get`/`set` requests. Note that when MIPS's state is saved using the automatic state capture provided by Synergy, all of its data memory must be saved as well. Thus, state transfer is reduced by 57% when Synergy waits for MIPS to quiesce.

Chapter 6

Discussion

6.1 Limitations

Currently, there is no mechanism in place for an application to send a signal to AmorphOS. Consequently, either the client software application or AmorphOS itself must poll the hardware application after requesting quiescence to determine if it is actually quiesced. This implementation can lead to increased traffic on the CntrlReg pathway, as well as additional latency between when a safe context switch becomes possible and when the system recognizes it. Furthermore, it is logically dissimilar to how quiescence is implemented in Synergy.

At the very least, for Synergy to be a suitable system for memory-intensive applications, it should provide a memory interface. Although we were not able to test the performance of the application memory interface implemented in Section 4.2.2, relying on file IO as a substitute for the FPGA's off-chip DRAM would introduce substantial latency. One option would be to expose AmorphOS's AMI to the application, but this would require all of the other backends supported by Synergy to have a similar implementation of memory.

Since the computation must start in software, one limitation of how quiescence is implemented in Synergy is that an application must run in software until it reaches a quiescent state before being transitioned to hardware.

Another limitation of this work is that we were not able to measure the performance of state restoration in Synergy when quiescence is asserted by the application. Finally, we fell short of our goal of adapting DnnWeaver to run end-to-end on Synergy.

6.2 Future Work

One extension to the AmorphOS quiescence interface would be to provide a state-saving mechanism that utilizes DMAs in the quiescence interface, similar to how the preemption interface is implemented in Optimus [42]. This feature would allow native applications to save minimal state at quiescence points more easily, and it would provide a more straightforward pathway for the hardware component of Synergy to store the state of each embedded application.

Another avenue for future work would be to provide support for native applications on the same fabric as Synergy applications. To introduce consistency to the programming models for Synergy and for AmorphOS, it may be beneficial allow hardware applications to send interrupt-like signals to indicate quiescence.

One intention with adapting the DnnWeaver generated LeNet inference accelerator was to be able to observe how its performance and resource utilization differs running on Synergy compared to AmorphOS. Since AmorphOS's publication, DnnWeaver v2.0, which addresses some correctness problems with the original framework and has an improved static scheduler, has been released [1]. In order to achieve this goal and those laid out in Chapter 3, we could adapt the LeNet applica-

tion generated by the new version of DnnWeaver for both AmorphOS and Synergy. With this application, we would also like to measure how the Synergy implementation of memory described in Section 4.2.2 affects the performance of the application. Additionally, it would be helpful to see if performance degrades substantially when DnnWeaver is multitenant with other applications that use Synergy's IO capabilities. It seems likely that contention on the IO path shared by applications would cause issues.

6.3 Conclusion

We introduce Synergy as an FPGA virtualization solution that is deployable on the FPGAs available in data centers today. The overheads from implementing automatic state capture can be reduced for applications that use a quiescence interface at the expense of latency during context switching. Providing this interface for native applications in AmorphOS increases fabric utilization by less than 1%. Implementing quiescence in a virtualized Synergy application can substantially reduce its resource utilization.

Bibliography

[1] [n. d.]. DnnWeaver v2.0: Open Source Specialized Computing Stack for Accelerating Deep Neural Networks. http://dnnweaver.org/

[2] [n. d.]. F1 Instances for Educators. https://aws.amazon.com/education/F1-instances-for-educators/

[3] [n. d.]. *FPGAs in the Emerging DNN Inference Landscape*. Technical Report. Xilinx. https://www.xilinx.com/support/documentation/white_papers/wp514-emerging-dnn.pdf

[4] [n. d.]. Intel® Arria® 10 Device Overview. ([n. d.]), 43.

[5] 2017. Microsoft unveils Project Brainwave for real-time AI. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/ Library Catalog: www.microsoft.com Section: Artifical intelligence.

[6] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11)*. Association for Computing Machinery, Monterey, CA, USA, 25–28. https://doi.org/10.1145/1950413.1950421

[7] Amazon. [n. d.]. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[8] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. 2005. hthreads: a hardware/software co-designed multithreaded RTOS kernel.

In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, Vol. 2. 8 pp.–338. https://doi.org/10.1109/ETFA.2005.1612697 ISSN: 1946-0759.

[9] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. (Oct. 2010), 20.

[10] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*. ACM Press, San Francisco, California, 1216. https://doi.org/10.1145/2228360.2228584

[11] Alban Bourge, Olivier Muller, and Frédéric Rousseau. 2015. Automatic High-Level Hardware Checkpoint Selection for Reconfigurable Systems. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 155–158. https://doi.org/10.1109/FCCM.2015.8

[12] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2015. Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. Association for Computing Machinery, Monterey, California, USA, 94–97. https://doi.org/10.1145/2684746.2689086

[13] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (Sept. 2001), 1059–1076. https://doi.org/10.1109/43.945302

[14] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA '14)*. Association for Computing Machinery, Monterey, California, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[15] Adrian M Caulfield, Eric S 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. (2016), 13.

[16] Liang Chen, Thomas Marconi, and Tulika Mitra. 2012. Online scheduling for multi-core shared reconfigurable fabric. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 582–585. https://doi.org/10.1109/DATE.2012.6176537 ISSN: 1558-1101.

[17] Eric Chung, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Jeremy Fowers, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Kalin Ovtcharov, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Michael Papamichael, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, Doug Burger, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, and Shlomi Alkalay. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (March 2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[18] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. Association for Computing Machinery, Monterey, California, USA, 105–110. https://doi.org/10.1145/2847263.2847339

[19] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. Association for

Computing Machinery, Monterey, California, USA, 217–226. https://doi.org/10.1145/3020078.3021739

[20] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (Oct. 1974), 256–268. https://doi.org/10.1109/JSSC.1974.1050511 Conference Name: IEEE Journal of Solid-State Circuits.

[21] Milan Dvořák and Jan Kořenek. 2014. Low latency book handling in FPGA for high frequency trading. In *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*. 175–178. https://doi.org/10.1109/DDECS.2014.6868785

[22] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. 2014. The LEAP FPGA operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927488 ISSN: 1946-1488.

[23] Chun-Hsian Huang, Kai-Jung Shih, Chao-Sheng Lin, Shih-Shiue Chang, and Pao-Ann Hsiung. 2007. Dynamically Swappable Hardware Design in Partially Reconfigurable Systems. In *2007 IEEE International Symposium on Circuits and Systems*. 2742–2745. https://doi.org/10.1109/ISCAS.2007.378620 ISSN: 2158-1525.

[24] Aws Ismail and Lesley Shannon. 2011. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *in Proc. 19th IEEE Symp. Field-Program. Custom Comput.* 170–177.

[25] Zsolt Istvan, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. 425–438. https://www.usenix.org/node/194955

[26] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. *ACM Transactions on Reconfigurable Technology and Systems* 8, 4 (Oct. 2015), 1–23. https://doi.org/10.1145/2815631

[27] Alexander Kaganov, Asif Lakhany, and Paul Chow. 2011. FPGA Acceleration of MultiFactor CDO Pricing. *ACM Transactions on Reconfigurable Technology and Systems* 4, 2 (May 2011), 20:1–20:17. https://doi.org/10.1145/1968502.1968511

[28] H. Kalte and M. Porrmann. 2005. Context saving and restoring for multitasking in reconfigurable systems. In *International Conference on Field Programmable Logic and Applications, 2005.* 223–228. https://doi.org/10.1109/FPL.2005.1515726 ISSN: 1946-1488.

[29] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4. https://doi.org/10.1109/FPL.2016.7577353 ISSN: 1946-1488.

[30] John H. Kelm and Steven S. Lumetta. 2008. HybridOS: runtime support for reconfigurable accelerators. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays (FPGA '08)*. Association for Computing Machinery, New York, NY, USA, 212–221. https://doi.org/10.1145/1344671.1344703

[31] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with. (2018), 22.

[32] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. 2008. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM Transactions on Design Automation of Electronic Systems* 13, 3 (July 2008), 40:1–40:27. https://doi.org/10.1145/1367045.1367049

[33] Ian Kuon and Jonathan Rose. 2010. *Quantifying and Exploring the Gap Between FPGAs and ASICs*. Springer Science & Business Media. Google-Books-ID: UIoI1Z5n_WwC.

[34] Joshua Landgraf, Eric Schkufza, William Lin, Tiffany Yang, and Christopher J. Rossbach. 2020. *Compiler-Driven FPGA Virtualization.* Technical Report. VMware Research Group.

[35] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High Frequency Trading Acceleration Using FPGAs. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL '11)*. IEEE Computer Society, USA, 317–322. https://doi.org/10.1109/FPL.2011.64

[36] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Dec. 1998), 2278–2323. https://doi.org/10.1109/5.726791 Publisher: Institute of Electrical and Electronics Engineers Inc.

[37] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. 2009. Hardware Context-Switch Methodology for Dynamically Partially Reconfigurable Systems. (2009), 17.

[38] L. Levinson, R. Manner, M. Sessler, and H. Simmler. 2000. Preemptive multitasking on FPGAs. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*. 301–302. https://doi.org/10.1109/FPGA.2000.903426

[39] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. 2016. ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference - SIGCOMM '16*. ACM Press, Florianopolis, Brazil, 1–14. https://doi.org/10.1145/2934872.2934897

[40] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 476–488. https://doi.org/10.1145/2749469.2750416 ISSN: 1063-6897.

[41] Enno Lübbers and Marco Platzner. 2009. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems* 9, 1 (Oct. 2009), 1–33. https://doi.org/10.1145/1596532.1596540

[42] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 827–844. https://doi.org/10.1145/3373376.3378482

[43] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Barcelona, Spain, 14–26. https://doi.org/10.1109/HPCA.2016.7446050

[44] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. 2003. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Automation and Test in Europe Conference and Exhibition 2003 Design*. 986–991. https://doi.org/10.1109/DATE.2003.1253733 ISSN: 1530-1591.

[45] Aaftab Munshi. 2009. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, Stanford, CA, 1–314. https://doi.org/10.1109/HOTCHIPS.2009.7478342

[46] Rishiyur S. Nikhil. 2008. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In *High-Level Synthesis: From Algorithm to Digital Circuit*, Philippe Coussy and Adam Morawiec (Eds.). Springer Netherlands, Dordrecht, 129–146. https://doi.org/10.1007/978-1-4020-8588-8_8

[47] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. Association for Computing Machinery, Monterey, California, USA, 111–117. https://doi.org/10.1145/2847263.2847337

[48] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. 2014. *A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services.* https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/

[49] Kiran Puttegowda, David I. Lehn, Jae H. Park, Peter Athanas, and Mark Jones. 2003. Context Switching in a Run-Time Reconfigurable System. *The Journal of Supercomputing* 26, 3 (Nov. 2003), 239–257. https://doi.org/10.1023/A:1025694914489

[50] Kyle Rupnow, Wenyin Fu, and Katherine Compton. 2009. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 63–70. https://doi.org/10.1109/FCCM.2009.30

[51] Sartaj Sahni and Atul Bhatt. 1980. The complexity of design automation problems. In *Proceedings of the 17th Design Automation Conference (DAC '80)*. Association for Computing Machinery, Minneapolis, Minnesota, USA, 402–411. https://doi.org/10.1145/800139.804562

[52] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, 271–286. https://doi.org/10.1145/3297858.3304010

[53] Hurmat Ali Shah, Laiq Hasan, and Nasir Ahmad. 2013. An optimized and low-cost FPGA-based DNA sequence alignment–a step towards personal genomics. *Conference proceedings: ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual Conference* 2013 (2013), 2696–2699. https://doi.org/10.1109/EMBC.2013.6610096

[54] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[55] Hayden Kwok-Hay So, Artem Tkachenko, and Robert Brodersen. 2006. A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH. (Oct. 2006), 6.

[56] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. Association for Computing Machinery, Monterey, California, USA, 16–25. https://doi.org/10.1145/2847263.2847276

[57] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. 2004. An FPGA run-time system for dynamical on-demand reconfiguration. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 135–. https://doi.org/10.1109/IPDPS.2004.1303106

[58] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A Modular FPGA Operating System for Dynamic Workloads. *arXiv:2001.09990 [cs]* (Jan. 2020). http://arxiv.org/abs/2001.09990 arXiv: 2001.09990.

[59] Dong Wang, Ke Xu, and Diankun Jiang. 2017. PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 279–282. https://doi.org/10.1109/FPT.2017.8280160

[60] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 845–858. https://doi.org/10.1145/3373376.3378491

[61] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. Association for Computing Machinery, Monterey, California, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[62] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. Association for Computing Machinery, Monterey, California, USA, 25–34. https://doi.org/10.1145/3020078.3021698

[63] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems - APSys '17*. ACM Press, Mumbai, India, 1–7. https://doi.org/10.1145/3124680.3124743

[64] Peng Zhang, Muhuan Huang, Bingjun Xiao, Hui Huang, and Jason Cong. 2015. CMOST: a system-level FPGA compilation framework. 2015 (July 2015). https://doi.org/10.1145/2744769.2744807

Vita

Tiffany Yang is from Houston, Texas. She attended high school at Choate Rosemary Hall in Wallingford, Connecticut. After earning her diploma, she began studying English literature and mechanical engineering at the University of Texas at Austin (UT Austin) in 2012, where she received her Bachelor of Science in Mechanical Engineering in 2016. In the fall of 2017, she enrolled in the masters program in the Department of Computer Science at UT Austin. Her master's research is focused on heterogenous systems, conducted under the supervision of Prof. Christopher J. Rossbach. Following graduation, she will work as a software engineer at VMware in Palo Alto, California.

Email Address: tiffanyyang at utexas dot edu

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.