Fluid:

Raising the Level of Abstraction for FPGA Accelerator Development Without Compromising Performance

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Joseph G. Melber

B.S., Electrical Engineering, University at Buffalo
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August 2021

Acknowledgements

Completing my PhD has been an incredibly meaningful and enriching process. First and foremost, I thank Prof. James C. Hoe for supporting me, guiding me towards interesting research and mentoring me throughout my PhD. James taught me how to think, find the essence of every question and develop an instinct for the truly interesting parts of a research project. I have grown as a researcher and most importantly as a person as a result of your guidance. Thank you.

I want to thank my committee members Prof. Tze Meng Low, Prof. Nathan Beckmann, and Dr. Kermin Elliott Fleming. Their valuable insights, constructive feedback and excellent advice helped me to refine my thesis. All of your perspectives impacted my thesis work and I am very grateful.

I thank my friends and collaborators at CMU. Thank you Siddharth Sahay, Shashank Obla and Zhipeng Zhao for your immeasurable help, research discussions, and friendship. I thank all of James' students that I had the pleasure to work and collaborate with: Gabriel Weisz, Michael Papamichael, Yu Wang, Guanglin Xu and Marie Nguyen. Thank you to all denizens of a-level for making it a great place to work, have lunch, and for your friendships. Thank you Thom Doru Popovici, Anuva Kulkarni, Milda Zizyte, Daniele Spampinato, John Filleau, Mark Blanco, Emily Ruppel, Elliott Binder, Upasana Sridhar and Bari Guzikowski. I have many fond memories of my time at CMU thanks to all of you. I truly appreciate and will miss all of the discussions, lunches and happy hours we had in Pittsburgh. Outside of CMU, I am very thankful for my friends from Buffalo and Pittsburgh for their support and for making graduate school fun.

Thank you to everyone in my family. Graduate school is a lengthy journey and your encouragement has been invaluable to the process. I would like to thank my parents for teaching me the importance of education and hard work, and for their love and support throughout my life. I also thank my in-laws, siblings, grandparents and extended family who have celebrated the highs and motivated me through the lows. I would like to thank my dog and office mate during the COVID-19 pandemic, Moose, who brings me constant

joy and companionship. Lastly, I am truly thankful for my wife, Margaret, for her unconditional love, encouragement and unending support. Without you I would not be who I am today, I love you.

Finally, I would like to thank Intel Corporation for their generous financial and equipment donations in the ISRA grant. I would also like to thank Intel Corporation and VMware Inc for their generous financial donation to and lengthy discussions within the Crossroads 3D FPGA-Academic Research Center.

Abstract

The unrestricted freedom presented to hardware accelerator developers by the inherently reconfigurable fabric of Field Programmable Gate Arrays (FPGAs) gives rise to highly optimized and efficient architectures. However, correctly navigating this same vast freedom is what makes hardware design difficult. Adding to this challenge, modern FPGAs continue to grow in logic capacity, and offer rich and varied memory and communication interfaces. FPGA accelerator developers need the same ease-of-development support software programmers have come to expect—through abstraction—to manage this complexity.

In current FPGA development flows, the lack of abstractions and proper tooling encouraging composition, encapsulation and code reuse leads to unnecessarily long development cycles. Hardware developers must manually adapt their accelerators for new platforms or, at worst, rewrite them. To address those challenges, so far, FPGA abstractions give disproportionate emphasis to reducing the design effort for algorithmic processing kernels rather than to the memory and communication architecture side of the design task. Designers are asked to build all of the datapaths for buffering, data movements, and external interfaces, as well as the state machines to coordinate these datapath activities. Bus-level integration and design abstractions remain the state-of-the-art for designing memory and communication infrastructure. Furthermore, such datapaths are often ad-hoc efforts and are not generally reusable.

This thesis presents Fluid, an FPGA-aware modular design methodology that enables higher levels of abstraction for memory, communication and system architecture. Fluid draws inspiration from service-oriented architecture concepts developed for cloud computing to separate functionality from implementation in hardware design. The Fluid design methodology encourages a decoupled design paradigm where accelerator developers are relieved of memory and communication logic within their modules. Fluid's service abstraction begins at a module's boundary. It defines a clean interface that logically decouples components in a design and enables flexible infrastructure implementations outside of the module's scope. This design methodology reduces FPGA hardware development cost

by removing infrastructure from a designer's modules and enabling them to architect their accelerator at a high level without RTL redevelopment or compromising performance.

Fluid encapsulates high level memory abstractions as services that provide designers starting points higher than bus interfaces and standard DMA IPs. Fluid also raises the level of abstraction for communication; the designer's modules communicate with service modules through logically point-to-point channels and a message passing standard. Fluid's design methodology enables layers of abstraction for accelerator architects, service library designers, and infrastructure designers. Fluid (1) absorbs the complexity of memory and communication infrastructure from designers' modules without increasing overhead or compromising performance, (2) enables higher levels of abstraction for hardware design by decoupling functionality from implementation, and (3) supports designers to flexibly compose accelerator systems on FPGAs at a high level with transparent control over implementation details.

This thesis creates a working Fluid design framework equipped with a catalog of services. These services are available as an extensible library of parameterizable, composable hardware modules implementing high-level and primitive building-block functionalities. Fluid's design framework provides a communication infrastructure library to flexibly compose, connect and place services across memory and compute devices. Designers build complete accelerators with a high level design framework that enables maintainable and expressive descriptions of their design. Fluid enables designers to abstractly specify their designs at a high level maintaining control over the implementation details. Designers use Fluid's programming interface to compose a service-oriented design and generate the necessary infrastructure between modules. Using this framework, we complete an evaluation of Fluid's merits through the design and implementation of a series of applications. The evaluation demonstrates the programmability benefits Fluid facilitates by separating specification from implementation for FPGA hardware development. Fluid's methodology increases design flexibility and convenience without increasing resource utilization or reducing performance—even for large, high-performance applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 FPGAs for Computing

For decades, general-purpose CPUs have been the staple computing architecture. Their ubiquity is due, in part, to their convenient programming model. Furthermore, CPUs have steadily increased their performance capability due to transistor scaling. However, in recent years their performance gains have severely slowed due to power limitations [28]. Now, computer architects turn to hardware specialization through dedicated compute accelerators to increase efficiency while continuing to add capabilities with additional available transistors [23]. Over the last decade field-programmable gate arrays (FPGAs) have rapidly evolved from their traditional use as glue-logic or verification technologies for CPUs and ASICs into fully capable computing devices [37, 66, 67]. This evolution is spurred by the computing industry's focus on hardware accelerators to increase compute power through efficient specialized circuits, while balancing a need for more flexibility than ASICs offer.

FPGAs occupy a unique position in the computing landscape. Their field-programmable nature allows accelerator designers to achieve high performance and efficiency through specialization while maintaining flexibility to change over time. FPGAs have demonstrated capability as compute devices accelerating many workloads with highly tuned compute kernels [17, 55, 85]. The promise of performance has even led to tighter integration with general-purpose CPUs, and in some cases cache coherent access to main memory [69]. Large-scale cloud networks of FPGAs have also been successfully deployed [30, 33, 76]. Furthermore, FPGAs are growing in logic and compute capacity with additional hardened compute macro blocks [35, 53]; large on-chip and in-package high-bandwidth memories appear inside FPGA packages to feed these additional compute capabilities as well [45]. All of these additional capabilities are coupled with extremely large logic fabrics to support larger accelerators on a single FPGA device. As FPGA

devices grow, distributing data across a large logic fabric requires new design patterns as long wires are no longer effective or efficient [2, 39]. Designers must adapt their designs to face this new reality by targeting recently available resources like hardened network-on-chips for data distribution.

While FPGAs themselves are capable and available, their programming model limits their widespread adoption—today's FPGAs are too large and complex for a single designer. Hardware designers harness the flexible potential of FPGAs controlling every bit at each cycle. While this level of control is effective for compute kernel design, it becomes tedious for application composition; namely implementing support structures and datapath state machines. Programming at the bit and cycle level limits portability and scalability when support infrastructure requires redevelopment for each additional platform. Designers must account for more heterogeneity and possibilities than ever before on today's modern FPGAs. Modern FPGA devices and platforms are rapidly evolving to meet the growing needs of the compute industry compounding the complexity of redevelopment to harness the additional compute, communication, and memory capabilities available to each new device [3, 21, 47]. Recently, several tools and abstractions have been introduced to simplify FPGA development which are discussed in detail in Chapter 2. Commercially available high-level synthesis tools and IP generators greatly simplify compute kernel design—and recent advances in programming abstractions facilitate simple data transfers and module connections [5, 41, 42, 95]. However, their focus remains on low-level load-store data transfers and structural connections through standard buses. While current tools assist in developing straightforward plumbing, their programming models are often tedious and error prone. FPGA designers are still responsible for more complex data access patterns and the semantics of the data transferred on module connections. Current FPGA tools and programming abstractions stop short at raising the level of abstraction for memory, communication infrastructure and design composition.

Many layers of abstraction have been built over time for general purpose processors—applying this same approach, abstraction can simplify FPGA application development. Memory is the fundamental general-purpose computing architecture, where further abstractions such as operating systems, and software libraries make software design efficient, portable, and flexible. However, not all CPU abstractions are a good fit for FPGAs, and abstractions have a different cost in hardware. Abstraction cost is incurred in logic capacity on the FPGA fabric, not time. FPGAs unique advantage as a programmable hardware technology should extend into an abstraction that increases their programmability. Hardware designers, through abstraction, should be able to start at design points higher than simple data transfers on busses. FPGA designers should be supported by specialized functionalities, available in a library, that reduce overall development time and facilitate access to data.

1.2 Increasing FPGA Programmability

Challenges. Though abstraction is promising to increase programmability, many abstractions are insufficient or complicated for widespread adoption. Developing hardware modules is complex and time consuming, especially following best practices for reusability and flexibility [71]. However, abstractions have been demonstrated for decades in software development to increase programmability. Rather than paying for abstraction in cycles that reduce performance as in software, hardware abstraction performance overhead is low, where functionalities are spatially distributed and would ordinarily be contained within the designer's kernel modules. Much of the design complexity in hardware accelerators arises from memory and infrastructure. FPGA designers need more support from tools and abstractions as a way to digest, understand, compose, and integrate modules correctly to increase programmability. Going forward, hardware designers require starting points higher than loads and stores for accessing data in memory, while retaining the flexibility required to produce efficient hardware designs.

An abstraction that balances productivity, efficiency and expressivity to decouple functionality from implementation meets this requirement. Modules rely on functionality exposed through the abstraction and implementation choices only change performance, power, and/or area rather than correctness. Developing abstractions that support designers and promote reusability is a difficult task by itself. Software abstractions typically employ a layered approach where higher level support is built in a stack. Following a similar design pattern, high-level hardware abstractions can be built hierarchically layering support on lower-level IPs and physical devices. This reduces development efforts by virtualizing platform details and providing portability.

Integrating IPs correctly and intelligently is essential to build real systems from abstract specifications. Managing integration complexity is required to factor memory and datapath infrastructure outside of the designer's compute modules. Currently, connections are purely mechanical in FPGA design processes. Designers must wire busses to their IPs and correctly implement the protocol. Abstraction can simplify module integration and system architecture as well. A proper FPGA design interface should raise the level of abstraction, above RTL, enabling designers to conveniently and efficiently architect their accelerators and provide transparent control over the implementation details. Simply describing physical connections isn't sufficient for designers and tools to correctly build designs. It is necessary to understand the semantics and structure of the data sent between modules in order to verify that modules are correctly connected. The design abstraction must be able to provide guarantees that the required modules are instantiated and connected correctly to implement the designer's high level specification.

Goals. The key question in this thesis is: *what abstractions fit for FPGA hardware development?* An answer

requires (1) an identification of the boundaries and functionalities where abstraction benefits the designer, (2) a good design methodology and expressive interface to effectively and efficiently support designers, and (3) an evaluation of this methodology that demonstrates it's ability to increase programmability without negatively impacting performance.

Fluid's goal is to reduce the barrier to entry for FPGA accelerator development, without compromising performance, and enable hardware developers to conveniently and efficiently develop their ideas in hardware. Fluid develops a design methodology that supports a hierarchy of abstraction layers that enable service-oriented hardware design, and decouple memory and communication infrastructure from a hardware developer's modules. It borrows concepts from service-oriented architecture to abstract memory and communication to simplify FPGA hardware development, both for module development and design integration. This service-oriented abstraction breaks hardware into digestible units with structured interfaces where the details of communication and memory infrastructure are relieved from the module designer's concern.

Flexibility is a subtle strength inherent to FPGAs. The abstractions Fluid initiates harness and celebrate flexibility through structure rather than uninhibited freedom. Fluid demonstrates that structure reduces complexity and encourages reusability, portability and composability. It raises the level of abstraction for memory from low level primitives, such as loads and stores, to high-level semantic rich operations like graph traversals. However, it is not enough to enable high level abstractions for accessing data in memory, a communication abstraction provides the necessary facilities to integrate modules into a larger design. Raising the level of abstraction for communication and memory allows the designer to utilize high-level memory operations and flexibly compose systems increasing programmability in terms of correctness, productivity, and performance.

The support Fluid provides, through service-oriented design practices, requires a shift in the designer's thought process from a flat module hierarchy to abstract distributed services, connected logically, outside their typical RTL design flow. Providing abstractions in the right places, namely limiting abstraction to the module's boundary and outside infrastructure, increases flexibility without reducing efficiency and enables designers to focus on their contributions. Additionally, Fluid's modular design methodology enables designers to describe their accelerators at a high level without ceding all of their control over implementation details to the abstraction. Fluid aims to support RTL designers, absorbing the development responsibilities around accessing data in memory and communication infrastructure, enabling designers to integrate their modules into a larger accelerator design through a high level description.

Figure 1.1: Service module abstraction.

1.3 A Methodology for Increasing Programmability

Fluid develops abstractions for FPGA memory and communication by absorbing the complexity of each from hardware developer's modules to increase programmability. Fluid also provides a design framework to integrate modules adhering to these abstractions. This thesis demonstrates that Fluid's approach increases flexibility and does not increase resource utilization nor negatively impact performance. Fluid's abstractions factor memory and communication logic out of a hardware designer's kernel module. Fluid develops a service-oriented design paradigm where modules are abstracted as services as shown in Figure 1.1. The hardware designer is relieved of the tedious and error prone responsibility of developing infrastructure for accessing and communicating data in their kernel module designs. Fluid simplifies RTL modules as its primary goal, and provides a library of high-level specialized services for accessing data as starting points higher than generic load/store interfaces. The provides-requires relationship shown in Figure 1.1 develops a framework so that modules can rely on functionalities outside of their scope and design changes do not impact functional correctness. Fluid's service design methodology defines a structured module interface so that modules and connections can easily be generated from a high level description. Furthermore, the provide-requires relationship enables the design generation framework to understand the semantics of connections beyond their physical properties to ensure correctness and support composability.

A designer's modules, as clients, request and consume services through a message passing protocol that flexibly supports single transfers or streams of data. The shift in thinking required for designers to rely on services naturally extends to communication. This thesis provides a communication abstraction where module connections are purely logical, just as services abstractly provide high-level operations to access and process data. Connections between a designer's kernel modules and service modules follow latency-insensitive design practices that separate the connections implementation and performance from functionality [14, 15]. As far as the designer is concerned connections are purely logical, this makes communication trivial regardless of the IP implementation language, architecture, device or board.

While Fluid's service abstraction is fundamentally influenced by FPGA design practices and FPGA's

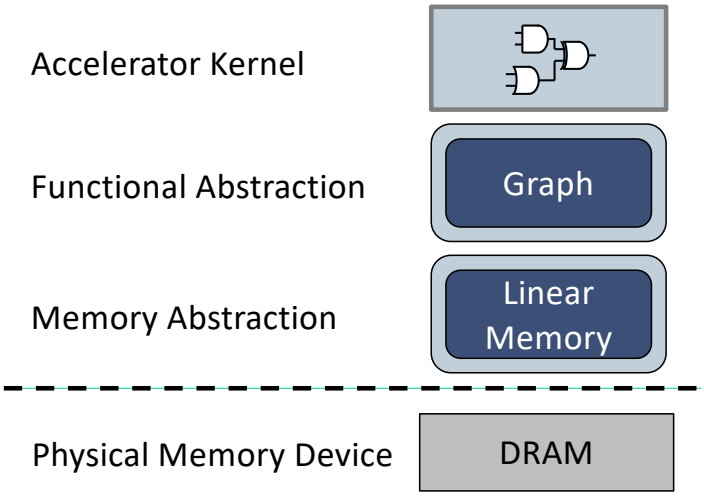


Figure 1.2: Memory as a service layered system architecture.

inherent flexibility to change over time, however it can apply to application-specific integrated circuit (ASIC) development. FPGA designs lend greater tolerance for abstraction compared to ASICs which typically have different design goals. The relatively longer ASIC development timelines leans towards extreme efficiency optimizing for every bit and cycle. However, as ASIC designs grow, module integration time increases; this is where the methodologies developed in this thesis can be applied. Fluid develops a methodology for modular design practices and abstraction where module connections carry semantic significance beyond structure and protocol. This allows tools to reduce the accidental complexities encountered in integrating modules in disaggregated designs. ASIC designers could also benefit from Fluid's high level design interface that would allow them to write expressive and maintainable descriptions of their infrastructure and accelerator's composition, and maintain them as IPs.

1.3.1 Memory as a Service

Memory Abstraction. Any abstraction can crumble without a sound foundation. Today's FPGAs are afforded a rich landscape of memory devices from monolithic on-chip memory to in-package high bandwidth memory to large capacity persistent memory. Fluid's memory abstraction supports the portability of higher-level services across memory devices without impacting functionality. Each memory type comes with its own set of properties and protocols in addition to its physical interface to the FPGA device. In order to build support for higher-level services and ensure portability, the lowest level memory services abstract platform details. The lowest level memory services provide familiar read and write primitives to access data in linear address spaces. Read and write services support a uniform abstraction that underpins

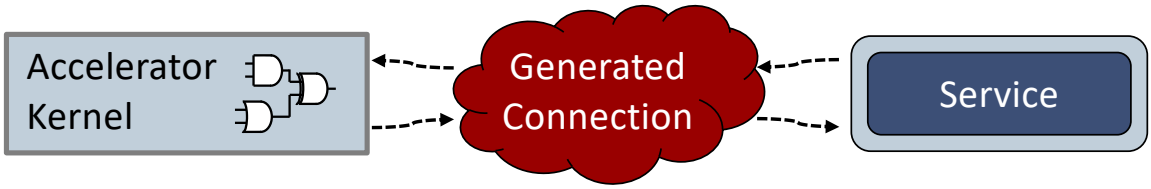


Figure 1.3: Communication as a service abstraction with generated connection implementations.

the memory as a service model by virtualizing the details of platforms and memory devices. Multiple read and write services can share a single memory device and support multiple virtual scratchpads. Additionally, read and write services are sufficiently parameterizable so that designers are unrestrained and can leverage a device's natural properties, control performance, or choose a simplified protocol. The currently available read and write services support common commercially available bus interfaces for on chip memory and off chip memory controllers: AXI, Avalon, and CCIP. Additionally, parameterizable caches are available as a performance enhancement layered on read and write services.

Functional Abstraction. To increase programmability and provide the same ease of use that software designers have come to expect, Fluid extends service-oriented design paradigm to provide abstract, high-level functionalities to hardware designers. Fluid's memory as a service paradigm raises the level of abstraction for hardware designers to services. The service abstraction encourages designers to disaggregate their design into reusable composable service modules to build more complex services in a hierarchy as shown in Figure 1.2. Even services are encouraged to be hierarchically composed of services. Services provide specialized operations to access and process data that live outside of the accelerator designer's module scope. Fluid provides a structured, uniform interface for communicating with services and a framework for linking clients' interfaces to service modules. Services are available in a library with high-level descriptions that form a catalog. Services absorb complex operations from the designer's kernel modules such as the logic for sequencing atomic operations in memory or managing data structure interactions. A service describes an abstract functionality, and multiple implementations of a service can exist in the library for example: designs encapsulating logic for different underlying data layouts and/or with various power, performance, and area characteristics.

1.3.2 Communication as a Service

Communication Abstraction. It is necessary to include communication as a first-class concern in any FPGA abstraction. It is insufficient to raise the level of abstraction only for memory—moving data both on- and off-chip is a major part of any non-trivial design and paramount for performance. The communication abstraction developed by Fluid defines a channel interface that relies on latency-insensitive design

practices that guarantees in-order delivery of messages but does not specify exact timing details. Relying on abstract communication between modules increases implementation flexibility, design scalability, and introduces the ability to transparently place modules across chip boundaries. Latency-insensitive communication is well studied for hardware design and inspired Fluid's communication abstraction [15,31]. Modules send request messages, through logically point-to-point channels, to services that will satisfy the request with an appropriate response. All necessary communication infrastructure to implement the connection the channel specifies lives outside of the module's scope. Figure 1.3 shows the communication abstraction where channel connections are implemented later on by a code generator. Generated connections are expressed abstractly; substituting one connection type for another doesn't impact the functionality of the overall design. This allows designers to flexibly compose modules at a high level and incrementally support new communication substrates for connections, without modifying their existing modules. Fluid provides a library of connection types including directly connecting modules with wires to more exotic connection types including network on chip and Ethernet connections.

Hardware-Software Abstraction. Fluid abstracts away the implementation details of both the service and communication infrastructure. Consequently, this makes hardware-software communication seamless between components that adhere to the abstraction. Fluid provides libraries and infrastructure for both hardware and software so that the channel and message passing abstraction is upheld across both architectures. Services can have software and hardware implementations transparent to a client module. Software threads on a host processor allow designers to quickly prototype services and even involve software in designs for complex operations. Fluid provides a couple channel connection implementations for hardware-software communication relying either on circular buffers in the host's main memory or memory mapped input-output transactions pushed by the host. This thesis develops an understanding of the capabilities of software services and demonstrates that software services can be used to simplify hardware design efforts in testing and integration.

1.3.3 Abstracting High Level System Design

Raising the level of abstraction for design composition is essential to unlock productivity and programmability gains for hardware designers using the abstractions developed by Fluid. Fluid's design composition framework provides a high level design interface that allows designers compose their systems in Python rather than RTL. The framework is transparent and flexible so that designers can logically compose their accelerator while maintaining control over implementation details. This is enabled by its expressive and extensible interface that exposes relevant design details as parameters. Fluid's design tool generates

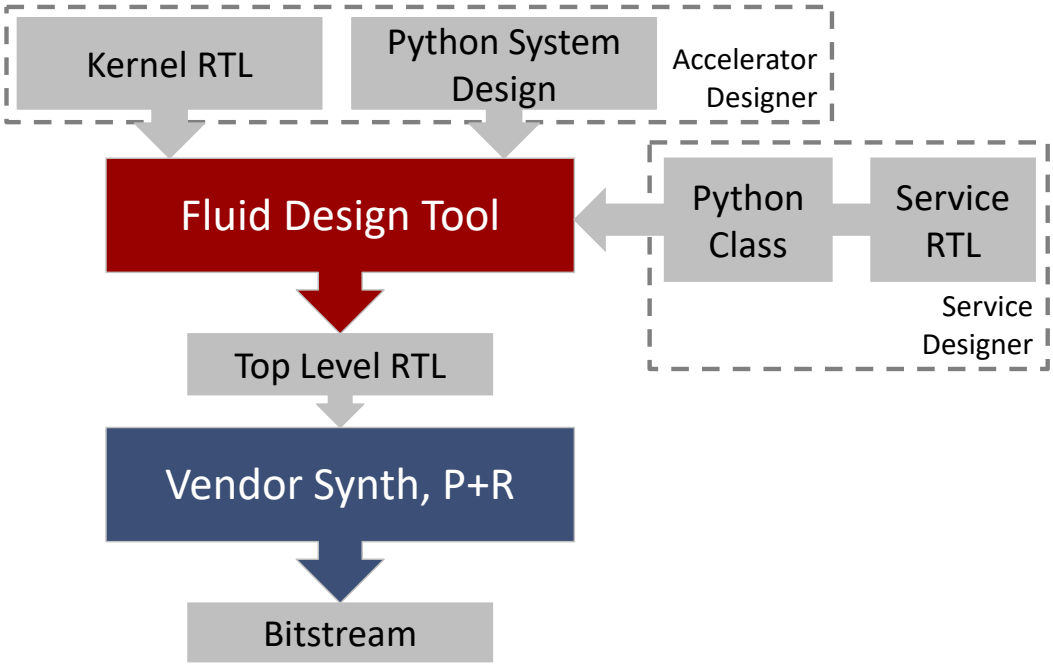


Figure 1.4: Fluid code and design flow.

the specified infrastructure to build connections between modules without completely removing the designer's influence. Additional abstraction layers and convenient functions can be built within the high level design interface to simplify the design process. Highlighting its extensibility, as FPGA devices incorporate emerging memory devices and/or hardened network on chips, designers can incrementally adopt these resources once they are introduced into the framework. Fluid encourages designers to capture the available performance and/or flexibility by regenerating their design from a high-level description now targeting the additional memories or communication resources with minimal additional effort.

A description of the design flow is shown in Figure 1.4. The accelerator designer uses a high level Python description to compose their service-oriented system design and writes their kernel modules in RTL. Designers instantiate services in their high level system design and specify connections between each module's channel interfaces. Every service has a corresponding class description in the service catalog and RTL implementation provided by the service's designer. The class describes the service, its channels, the services it provides and requires on those channels, as well as any additional wires and registers in its interface. Fluid's code generation framework takes the designer's Python description of their system and generates SystemVerilog code that instantiates modules, connections, and external interfaces. Design choices and exploration are simplified as the designer works with a high level description rather than RTL. Furthermore, the framework ensures channel connections are correctly elaborated and satisfy the provides-requires service contract.

1.4 Working with Fluid in Practice

Fluid's design methodology introduces abstraction layers for memory, communication and system design. The methodology decouples the hardware development process separating module and system design. This separation of concerns simplifies the development process and enables layers of abstraction to be built that increase productivity. Fluid's design framework provides a high level interface to build service-oriented accelerator designs; the details and features of which are described and evaluated in this thesis. The memory and communication abstractions developed by Fluid address FPGA design complexity by reliving the designer of the responsibility of implementing this logic in their modules. Support for these abstractions and relevant accelerator examples are realized and evaluated as thesis contributions. A library of services is designed and implemented as a proof of concept set in this thesis. This initial set of services covers a range of useful and complex memory functionalities that demonstrate the types of operations that fit the service-oriented design paradigm. This thesis also provides a library of supporting IPs for memory and infrastructure that designers leverage explicitly and implicitly to build accelerators. Specifically, some IPs are instantiated behind the scenes to adapt and share external interfaces on the FPGA device, whereas other IPs adapt and integrate network on chips and inter-FPGA communication devices.

1.4.1 Fluid Benefits Evaluation

An evaluation of Fluid's abstractions and design framework reveals the flexibility enabled by separating functionality from implementation. The evaluation also demonstrates that structured interfaces support the abstraction for designers to produce correct and efficient designs from abstract functionalities as services. Fluid is evaluated (1) in a study of the communication abstraction and library of connection types, (2) case study of a graph accelerator for the breadth-first search algorithm, (3) design and implementation of a sparse-matrix vector multiplication accelerator, and (4) through a high-performance intrusion prevention and detection accelerator. Notably, the evaluation demonstrates that Fluid raises the level of abstraction, in doing so increases programmability without impacting performance or resource utilization.

The communication abstraction is evaluated across the set of connection implementations designed and provided in the framework. The study evaluates data transfers of relevant sizes between hardware services and develops recommendations for design choices. The communication abstraction study further evaluates connection types between hardware and software services and offers a novel use-case for software services in the hardware developer's design process. These studies show that abstracting communication relieves hardware developers from the tedious task of plumbing for data movement and introduces additional flexibility to incrementally adopt new communication technologies and module placements.

The breadth-first search (BFS) accelerator was selected as a challenging application which requires significant complexity in order to achieve performance. The advantages of a service-oriented design are evident where services allow the designer's modules to appear close to the algorithm pseudocode and to flexibly modify the design at a high level targeting various performance outcomes. BFS is a key algorithm in graph analytics and in recent years, as data sets have exploded in size, become a target for hardware accelerators. BFS is characterized by irregular memory access patterns that are challenging for any architecture. Furthermore, sparse representations of the input graph are often used to increase efficiency at the trade-off of added complexity. A BFS accelerator designed by an expert hardware developer serves as a baseline [90]; an evolution of this design, relying on services, absorbs the complexity around the irregular memory access patterns and data structures. Furthermore, portability and flexibility are studied where services transparently access data in a variety of memory devices and across several network on chip implementations.

Sparse-matrix vector multiplication (SpMV) is a fundamental computational kernel in many applications. A design and implementation of a SpMV accelerator generalizes and extends the BFS case study. SpMV relies on sparse data structures to represent the input matrix in memory reducing its footprint and increasing performance. The SpMV design reuses the graph service from the BFS design. Services emulated in software, transparent to the hardware services on the FPGA, were used in the accelerator development process to simplify debugging and testing of the additional services necessary to complete the SpMV accelerator.

The final application evaluated is a service-oriented adaptation of Pigasus, an expertly crafted intrusion prevention and detection accelerator that achieves 100Gbps throughput [101]. This demonstration abstracts the accelerator's modules as services and further abstracts high-level functionalities as reusable, portable services. Impressively, the final service-oriented design achieves identical performance, without appreciably increasing resource utilization. Furthermore, the service-oriented Pigasus design can be generated through a high-level design specification reducing design effort to produce design variants including multi-FPGA implementations.

1.5 Thesis Contributions

Fluid is an FPGA design methodology that increases programmability by encouraging a modular design style, requiring structured interfaces. Fluid enables higher levels of abstraction to be built for memory, communication and design description. This thesis demonstrates that Fluid raises the level of abstraction for FPGA accelerator design without compromising performance, and makes the following contributions:

- The Fluid modular design methodology which defines a service abstraction for memory and communication that simplifies a module's scope—the designer is only tasked with a module's internal logic rather than infrastructure. The design methodology enables a designer to logically integrate functionalities *as a service* outside of a module internal hierarchy creating a layered abstraction that insulates designers from low level details.

- An accelerator design framework supporting the modular service-oriented design methodology, which hierarchically composes abstractions for FPGA hardware design. The design framework enables designers to efficiently and flexibly integrate modules into a system through a high level programming interface retaining transparent control over the implementation details. It further supports designers to encapsulate and reuse their infrastructure composition patterns as intellectual property.

- Development of a working infrastructural library supporting the communication abstraction and initial library of services that provide essential building blocks for memory operations and demonstrate that higher level memory abstractions, such as graph operations, can be encapsulated by the Fluid design methodology. Designers can rely on and contribute to the library of high-level memory services as starting points at an abstraction level higher than loads, stores, and addresses.

- Demonstration and evaluation that the Fluid design methodology increases FPGA programmability by raising the level of abstraction for FPGA accelerator design, and simplifies module development and integration without increasing resource utilization or decreasing performance.

1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information on FPGAs as computing devices and current research on FPGA abstractions and programming models. Chapter 3 describes the Fluid memory as a service abstraction in detail. Chapter 4 describes the communication as a service abstraction developed by Fluid. Chapter 5 details and discusses the high level design framework designers use to compose service-oriented accelerators. Chapter 6 presents the catalog for the current library of services and their implementation details. Chapter 7 discusses the implementation details of the connection types supported by the communication abstraction. Chapter 8 details the evaluation studies of the Fluid design methodology, its abstractions and design framework. Chapter 9 concludes and proposes ongoing and future work inspired by Fluid.

Chapter 2

Background

This chapter presents background material that informs the reasoning behind Fluid's design methodology and abstractions. Particularly, this section will cover motivations for FPGA computing, prior FPGA abstractions, FPGA computing architectures and infrastructures, FPGA communication architectures, and concepts that influenced the work in this thesis.

2.1 Computing with FPGAs

FPGAs have been deployed in various settings leveraging the efficiency and flexibility of the reconfigurable logic. Typically, FPGAs have been used as prototyping devices for ASICs or glue logic in embedded systems. However, FPGAs have evolved and increasingly gained capability as compute devices [21, 35, 53]. Modern FPGA devices provide a large logic fabric featuring large memory capacity and compute capabilities. This evolution comes as general purpose processors face diminishing performance returns in the face of power constraints [28]. FPGAs provide a promising compute solution enabling hardware specialization for power-efficient computing.

Historically, FPGAs have been integrated into computing systems as external stand-alone boards or peripheral cards. In this model a host processor offloads computation to these boards over low-bandwidth IO buses or serial ports. As IO bus speeds have continued to improve, FPGA cards have become more fully featured. Modern FPGAs are available in standardized form factors, some with vendor-provided drivers that enable the FPGA fabric to access host memory, without host intervention, over the PCIe bus. These cards typically feature localized on-board and sometimes in-package high bandwidth memories. While the offload model has proven to be effective, there is recent interest in tighter integration between FPGAs and host processors. Newer systems integrate FPGAs on the primary memory bus, and in some cases with cache-coherent access to main memory to share work at a finer granularity with processors [69, 75].

FPGAs are also becoming a part of datacenter infrastructure, lending their compute capabilities as data flows through Ethernet networks [30, 63]. Commercial deployments with large networks of FPGAs have also been demonstrated, moving FPGAs closer to the data, supporting networked functions [101], and also large scale datacenter applications [33, 76]. These new deployment and integration schemes have led to a new era for FPGAs, where specialization can be applied to an increasing set of evolving workloads. Most recently FPGAs have exploded in popularity as compute devices demonstrating capability in machine learning and data analytics [6, 67, 89].

While FPGAs have demonstrated capability and are increasingly used as compute devices in commercial settings, FPGAs currently only account for a small fraction of compute devices in deployment [25]. FPGAs lack the same widespread adoption of general purpose CPUs due to their cumbersome programming model. FPGAs still most commonly programmed at the bit and cycle level. This high level of control is necessary for efficiency, however it has proven to be cumbersome as devices and systems scale. FPGAs lack the same level of support from abstraction that software programmers rely on and enjoy. Abstraction increases programmability by hiding complexity when high levels of control aren't necessary. The overwhelming focus in commercial development is on increasing programmability through new programming models and languages as FPGAs mature into compute devices [97].

Even with this increased focus on programmability, FPGA accelerator designers efforts are often ad-hoc and single-purpose. Engineering efforts are often repeated each design even on the same device and platform. Providing structure and portability through abstraction in the right places increases programmability and allows designers to focus their efforts on tuning their computation kernels rather than on infrastructure.

2.2 FPGA Programming and Control Abstractions

While FPGA devices have been promoted to first-class computing devices, their programming model is their Achilles's heel. The programming model for FPGA's has not evolved as fast as the hardware technology has. FPGAs are often programmed with RTL languages like Verilog and VHDL. These are well known hardware description languages (HDL), however they are notoriously tedious to use and difficult to debug, much like assembly language in general purpose processors. They provide absolute control over every bit each cycle, and this leads to long development cycles and complexity. HDL code further lacks software's portability as the tools and abstractions provided to HDL developers are comparatively underdeveloped. Consequently, HDL design-time burdens developers when reimplementing interface logic for new FPGA platforms. Therefore, most deployments fail to take advantage of FPGA's flexibility—

in the wild they are often viewed as static, single-task accelerators.

Several attempts have been made to replace HDLs with higher level languages. Some attempts borrow from functional programming languages and make use of compilers to profile, elaborate, and generate code. Examples of such languages are Blusepec [10], Chisel [7], SpinalHDL [82], and MyHDL [64]. These tools have been used for many academic and commercial projects, including implementations of entire processors like RISC-V. These languages focus on the description of the algorithm datapath, abstracting timing and scheduling details. This simplifies some of the design task but does not apply any higher-level abstraction to compose systems. Designers must still rely on separate integration tools that are mostly manual and bus based. Fluid's design methodology is unconcerned with a module's internal logic, a module is an abstract black-box functionality that can be implemented in any hardware description language. Fluid provides a framework and design interface that simplifies integration and enables the designer to descriptively specify infrastructure at a high level.

Another major development in FPGA programming has been in high-level synthesis (HLS). In this model, designers specify hardware using a stylized C language, and a compiler maps this code to hardware. Numerous HLS tools have been developed including, Vivado HLS/Vitis [93], Intel HLS (Altera A++) [42], Catapult-C [60], Synphony-C [83], and LegUp HLS [13]. HLS compilers specialize in generating deep parallel pipelines, but heavily rely on users for compiler hints to capture parallelism and data reuse. HLS tools often fail when describing concurrent interactions between function calls and scheduling code around long latency DRAM accesses. This requires designers to develop their own logic to handle kernel-to-kernel and kernel-to-memory interactions or rely on embedded processors for high level control. Similar to the languages discussed in the previous paragraph, HLS remains focused on describing algorithms in hardware. It does not provide higher-level abstractions for integrating HLS generated modules into larger accelerator designs.

Recent work has extended HLS compilers with additional support for task parallel programs [20]. In this work, the authors employ a channel interface to capture spatial parallelism in their designs. This allows the compiler to decompose functions by these channel interfaces and support parallelism more naturally in the inherently sequential C language. Other work extends HLS with a similar channel abstraction for hardware-software communication [61]. This work enables transparent migration of functionalities between hardware and software using HLS and standard C compilers respectively. These extensions to HLS attempt to tackle the deficiencies of HLS to capture task parallelism while adding some additional capabilities in hardware-software co-design. These tool extensions could be useful in developing modules within Fluid, however software abstractions remain still fail to capture the true concurrency of hardware design and don't attempt to manage the complexity of off-chip data access. Another orthogonal approach to

augment HLS compilers is by combining the concept of decoupled-access-execute to support specialized prefetching and irregular memory access patterns [18, 19, 32]. This approach decouples memory access from the computation in HLS programs and demonstrates that by doing so designs can better tolerate long latency of irregular memory access patterns. This decoupled approach and additional latency tolerance is common to Fluid's methodology. Fluid services demonstrate that by decoupling the logic responsible for irregular memory accesses, the user-level design is greatly simplified. Furthermore, Fluid demonstrates that higher-level memory operations, including irregular memory accesses, can be encapsulated into reusable modules accessed via latency-insensitive interfaces.

High-level synthesis has been further extended to compile complete programs for FPGA acceleration. These compilers profile and analyze complete programs to find targets as acceleration candidates. After generating FPGA kernels for these targeted code sections, tools build the surrounding infrastructure to support data transfer, and kernel invocation. The entire process appears seamless to programmers; software code remains untouched, however, now portions have migrated to FPGA accelerators. Altera's OpenCL is an example of this type of compiler [41]. The OpenCL framework compiles C code for computing systems with a host-processor and accelerator processing-elements. As a general framework, OpenCL can target both FPGAs and GPUs as processing-elements, therefore code with SIMD-pipeline structure and parallelism produce high quality results. LegUp is another similar HLS tool [13]. Rather than target a host-accelerator platform, the LegUp compiler generates soft processors on the FPGA for sequential code sections. Xilinx SDSoC [94] functions much like LegUp, where sequential code targets hardened ARM processor cores. While these tools abstract more of the complexity away, at their core they face many of the problems of traditional HLS tools. Scheduling around DRAM latency, and irregular parallelism rely on the application designer at best, and are impossible at worst.

Domain specific programming languages (DSLs) represent another important class of programming abstractions for describing FPGA accelerators [49]. DSLs provide higher levels of abstraction targeted to the domain of an application expert. Experts express their computations in a manner natural to their domain relying on compilers and tools to implement their algorithms. This approach has demonstrated success to design and as an interface to custom hardware accelerators in graph analytics [68], machine learning [1, 73], network functions [12, 40, 62, 81, 101], and digital signal processing [34, 65]. However, each language is limited in scope to its domain and the designer has minimal control over implementation details relying on the compiler to efficiently implement their intended design. Fluid's design methodology is a flexible approach allowing designers to build highly tuned hardware modules without restriction to any language, architect their accelerator at a high level, and generate the infrastructure between modules. This provides an opportunity for intersection with DSLs where service modules can be developed in a

DSL and integrated into a design by Fluid's toolset. Moreover, DSL compilers often implement their computations from a library of common building blocks. This is similar to Fluid's approach; it is conceivable that Fluid's toolset and service module library could provide building blocks to support DSL compilers on FPGAs.

2.3 FPGA Infrastructures and Memory Architectures

FPGA vendors commonly support their compute-kernel design tools with infrastructure and system design tools [5, 95]. These tools present a graphical interface to designers to connect components through buses, such as AXI and Avalon [43, 96]. Designers may also compose kernels and application components with the help of intellectual property (IP) libraries. These infrastructures provide the necessary support for application construction. However, with new versions of the tools, buses and IPs change; this can require redevelopment costs for individual kernels and entire applications. Furthermore, buses, while simple, lead to scalability issues when many components contend to drive the bus. This has led to developing interest in network-on-chip (NOC) IP-generation tools [72] and some FPGA vendors are experimenting with hardened NOCs on FPGAs [2, 3, 35]. However, vendors stop short simply providing additional infrastructure without convenient tools and programming interfaces. Fluid's design framework provides a descriptive interface for designers to specify and utilize infrastructure from a high level description of their accelerator design.

System on Chip (SoC) design tools have been researched to tackle the challenges of integrating heterogeneous IPs within a system on a single piece of silicon. FuseSoC is a package manager and build tool that enables scalable HDL design through modularity, code reuse, and generation [50]. This tool could be used to support the ideas developed in this thesis and support the design framework for generating Fluid designs. The Embedded Scalable Platforms tool proposes that regularity and constraints mitigate integration complexity [58]. Chipyard similarly simplifies the SoC design process with a convenient design framework. These tools target ASIC designers that must integrate processors, accelerators, and caches in their SoC. Constraining the IPs to a regular set of interfaces allows the tool to generate systems with these IPs. While these tools are successful for IP integration, they are mostly manual and based on busses. The bus abstraction has plenty of legacy however it is limiting as SoCs and accelerators continue to scale. These tools also stop short in their design description capabilities to raise the level of abstraction for system composition. Fluid's design methodology abstracts infrastructure used to integrate modules allowing designers to logically architect their systems through its design framework. This descriptive design framework empowers designers with transparent control over the implementation of

their system and infrastructure described at a high level.

RCMW is an FPGA infrastructure abstraction that targets platform portability by presenting a standard set of interfaces to user logic [51]. The interfaces presented are tailored for both burst-streaming access patterns and memory mapped patterns. Furthermore, RCMW provides a set of infrastructural components for data transfer and control, to and from the FPGA. This provides platform virtualization from both the host-processor and FPGA, however RCMW does not include virtualization support for internal FPGA resources and continues to promote a low-level load-store memory model.

FPGA "architectures" seek to conquer the programming problem by replicating the support that operating systems, and instruction set architectures bring to general-purpose processors. The LEAP architecture seeks to simplify FPGA application programming through memory and communication abstractions [4, 98]. LEAP supports a latency insensitive channel as its communication primitive with abstract "named" send/receive endpoints. LEAP scratchpads present a virtual memory interface to FPGA designers with support for automatic cache hierarchy generation that includes the host processor's DRAM. LEAP further provides mechanisms for coherency among scratchpads. Both LEAP communication and memory primitives are low-level abstractions that increase portability. LEAP also provides support libraries as operating system services to aid in accelerator design for printouts, debugging, and synchronization [31].

LEAP was a key influence for the work in this thesis. LEAP scratchpad, coherent scratchpad memory and cache support is comprehensive; it could be used to underpin Fluid's read/write memory services. Working within LEAP, designers could have implemented higher level memory operations on top of their scratchpad memories, however LEAP did not explore this direction in their work. Fluid's methodology provides structure and semantics to encapsulate higher level memory abstractions and make them available to designers within the FPGA fabric. For example, Fluid encapsulates support for graph operations which are memory intensive and stress the memory infrastructure. LEAP's primary abstraction is communication and latency-insensitive channels. Latency-insensitive channels also could be used within Fluid's communication infrastructure. However, the key difference between Fluid and LEAP is in their design interface. Their interface, AWB, abstracts the channel and scratchpad implementation details, limiting the designer's input to platform choice. However, Fluid's design interface gives the designer transparent control over their implementation in the designer's high-level accelerator description. Fluid's design interface is much more descriptive and extensible compared to LEAP's AWB tool, owing its flexibility to Fluid's structured abstractions that allow hardware to easily be described in Python. AWB allowed the programmer to substitute implementations of functionalities such as platform interfaces, but it was very limited in its description capability. For example, Fluid's design tool enables the designer to transparently specify individual connection types and even build clean functions to abstract the construction of more

complex infrastructure.

CoRAM simplifies FPGA programming by decoupling compute and control through a data management API [22, 24]. The CoRAM architecture supports a data management API defined by SRAM blocks where software-defined control threads handle all transfers to/from SRAMs from/to memory. Compute kernels access these SRAMs with a memory-mapped interface. The CoRAM++ architecture extends the API to data-structure-specific commands and a FIFO interface to kernels [91]. The CoRAM++ architecture also enables supporting modules called agents that accelerate data-structure-specific memory commands. Fluid's methodology derives inspiration from CoRAM in decoupling compute kernels from memory and control. However, CoRAM lacks a proper communication abstraction forcing the designer to synchronize kernels through its software abstraction and explicit data transfers. This forces designers to implement kernels that handle communication internally, or communicate through memory similar to general-purpose processors. While memory is arguably the most important part of application design, it is only part of a bigger picture. Data movement is a large concern for efficient FPGA applications, and should be a principal focus of any abstraction. Fluid was influenced by CoRAM++ and preserves its goal of facilitating higher levels of abstraction for memory on the FPGA fabric. Fluid's design methodology supports designers to hierarchically develop and layer memory abstractions for FPGA computing. These abstractions include data-structure-specific memory access patterns—like CoRAM++—but also objects and containers, such as graphs and queues, abstracting their behavior and state. Fluid also enables processing within the memory system and provides support for communication between kernels outside of memory.

Researchers have also adapted operating system concepts to the FPGA design process [31,52,99]. Operating system abstractions provide a common layer of support across various FPGA devices and platforms. Memory support typically is the first level of concern in FPGA operating systems with configurations for virtual and cache coherent memory. This is important for heterogeneous processor-FPGA systems and FPGAs in the data center. While it is important and challenging to abstract memory interfaces and increase portability, FPGA operating systems remain low-level supporting the familiar load/store paradigm. Fluid's service-oriented abstraction encourages composability, encapsulation, dynamism, and high-level memory operations, richer than loads and stores, in addition to memory portability.

2.4 FPGA Communication Architectures

Messaging frameworks have been developed with the goal of providing a standard of communication for FPGA computing platforms. Previously mentioned as an FPGA memory architecture, LEAP also

provides communication interfaces called soft connections [74]. Soft connections are used to decouple components and communicate data among them in an asynchronous, latency-insensitive model. Elastic silicon interconnects (ESI) is an abstraction that is also based on latency-insensitive communication similar to LEAP [27]. ESI is an interconnect generator that supports communication with software code and hypothesizes supporting their abstraction with standardized board support modules. Latency-insensitive connections allow flexible communication between modules and are a central tenet of the work in this thesis. Fluid is similarly based on abstract communication through latency-insensitive channels and abstracts platform interfaces with standardized interfaces. However, Fluid ascribes a functionality to each abstract channel and verifies that the correct modules providing and requesting this functionality are constructed. Connections between modules in the Fluid communication abstraction carry more semantic meaning beyond simply data-type compatibility.

Direct Universal Access is another FPGA communication architecture that provides a uniform interface for FPGAs to access resources for memory and communication in the data center [80]. Megh computing developed their Arka runtime API which allows designers to build streaming data pipelines across FPGAs and CPUs [9]. This framework enables abstract heterogeneous communication between pipeline stages and for designers to place these stages transparently across devices and architectures. Their support and capabilities are limited to streaming pipelines; Fluid's message passing interface supports streaming as a subset of its capabilities. Another messaging framework, HGum, specifically targets distributed networks of FPGAs [100]. HGum provides support for host-processor to FPGA communication with smart serialization and deserialization of large messages. HGum focuses on data movement through message transmission and does not consider how their abstraction or framework could be applied to access data in memory. TMD-MPI extends the MPI API to make FPGAs accessible to software programmers. [78, 79]. Galapagos similarly extends the MPI API and provides a heterogeneous deployment stack that enables CPU-FPGA communication [29]. These works demonstrate a heterogeneous programming and communication model, however they focus on large transfer sizes and support for standardized software-focused interfaces. Lastly, GAScore with its programmable active message sequencer (PAMS) presents an active message interface and infrastructure to virtualize FPGA functions to software and each other [92].

These works support the claim that message passing is an appropriate abstraction for FPGA communication. Fluid adopts this model in its communication abstraction. Fluid provides a framework and methodology to abstract semantic rich operations on data across memory devices and through a variety of physical communication substrates. Many of the concepts proposed and implemented in these prior works could support and implement the communication infrastructure in Fluid designs. This thesis provides an abstraction to hide these details from the developer and develops guidance on the types of

communication substrates necessary to build efficient FPGA accelerators.

2.5 Near Data Computing

The concept of "Big data" describes the exponentially increasing volume of data generated and stored by organizations interested in extracting information through an increasingly large and diverse set of applications. Moving data dominates compute costs in these applications and research has been done to identify opportunities to move computation closer to data to increase efficiency [36]. This concept is referred to as Processing in Memory (PIM). Beyond identifying computation to move near or inside memory, additional work has been developed to understand and organize the types of operations within the PIM field [57].

Moving computation throughout the memory hierarchy in traditional processing units has shown promise, in particular when computations are implemented in flexible programmable-logic for power efficiency [56]. Fluid's design methodology encourages processing within the memory system to move computation near data in memory or as data flows through the accelerator. Fluid's abstractions begin at a module's boundary and providing a paradigm where designers access functionalities logically, including computation, as if it is a part of the infrastructure. This bounded field of view enables designers to reduce their module's internal complexity and rely on functionalities outside of their design scope. Fluid enables designers to conveniently design and build FPGA accelerators, and move computation closer to data.

2.6 Active Messages

Active messages are a communication primitive for integrating communication and computation. Software messaging interfaces simplify development efforts by encouraging portable and reusable design. Software messaging APIs, such as RabitMQ and ActiveMQ [48], define a uniform interface and provide a communication infrastructure to support their abstraction. In traditional message-passing programming interfaces, messages transfer data and implicitly transfer control. By contrast, active messages pass instructions relevant to processing the data contained in the message [70, 86, 87]. An active message system incorporates three elements: requests, message handlers, and receivers. Receivers and handlers use the arguments in an active message as instructions for processing data. Active messages provide a light-weight, flexible primitive for constructing a rich set of communications between agents in a computing system. Each active message typically carries data in addition to instructions, and the receiver uses both to produce the desired response. This may seem similar to the concept of remote procedure calls (RPC). RPCs abstract away communication performing operations as a function call. Active messages could be used

to as a primitive to underpin RPC communications. Fluid was inspired by and proposes active messages in its communication abstraction. Active messages carry requests and responses to/from services with arguments that contain instructions and data relevant to its operation.

2.7 Service-Oriented Architecture

Cloud computing is the pervasive model today supporting on-demand access to compute and data resources through APIs over the internet rather than on-site physical infrastructure. Cloud computing applications are delivered over the internet often as an abstract "service" [77]. The term software as a service (SaaS) commonly refers to this generalized model. Cloud computing has ballooned to include everything as a service in the XaaS model encompassing platforms, infrastructure, data, and more. Even FPGAs have been deployed in the XaaS model in the Microsoft cloud at a multi-datacenter scale [16]. These FPGAs can be used individually or together to accelerate large data center services and applications in a"Hardware as a Service" model. Additionally, embedded heterogeneous processing platforms that include accelerators and software have been virtualized as services [88]. This framework targets hardware system on a chip (SoC) design and prototyping. Both of these concepts virtualize and manage processing functions whether at datacenter or SoC scale. The data as a service (DaaS) was introduced to abstract data access [84]. DaaS provides data as a product to users regardless of its location or the processing required to acquire it. Fluid derives inspiration from service-oriented architecture pioneered in the data center and applies it to abstract and virtualize FPGA functionalities, particularly operations around accessing data in memory and communicating data in FPGA accelerators. The service model is a good fit for FPGA hardware as it decouples specification from implementation. Furthermore, applying the service abstraction to hardware provides a well defined interface that encourages modular design and reusability. The service abstraction provides a clean interface that allows designers to efficiently describe their accelerators at a high level with transparent control over the implementation details.

Chapter 3

Memory as a Service

After introducing Fluid's design methodology, this chapter presents the application of service-oriented design to FPGA memory. The previous chapter provided an overview of current FPGA design methodologies and memory abstractions. In current development flows, FPGA designers lack first order support for memory abstractions higher than bus-level device interfaces or processor-like load-store hierarchies. This chapter defines a memory as a service abstraction for FPGA computing. This service-oriented memory abstraction reimagines how FPGA accelerator developers view and access data stored in memory. It abstracts and encapsulates logic responsible for memory functionalities as high-level services, factoring this logic outside of an accelerator designer's kernel modules. Memory services increase portability, simplify user-level code, and maintain the customizability inherent to reconfigurable computing's success.

3.1 Motivating Another FPGA Memory Architecture

FPGAs possess an enormous potential for efficient computations with large reconfigurable logic fabrics and hardened processing units. Their extreme customizability allows hardware designers to efficiently target rapidly-evolving, highly irregular applications that are difficult for more general architectures, such as CPUs and GPUs, to handle efficiently. Modern FPGA platforms are increasingly capable; they integrate large FPGA devices with a rich hierarchy of high-performance memory devices. With this increased capability, the complexity and cost of developing FPGA accelerators are growing faster than ever before.

Achieving maximum efficiency and performance from any computing platform is difficult. The experience is similar whether a programmer is tuning software kernels in assembly or designing optimal hardware kernels at the bit and cycle level. However, software programmers rely on many layers of abstraction and libraries that provide essential support to their applications. For FPGAs to reach the same level of adoption as CPUs or even GPUs the development effort required to build an accelerator needs a

similar reduction. FPGA designers need abstractions and tools that increase programmability and help manage the complexity of FPGA accelerator design. In particular, abstractions that encourage modular design practices where expertly crafted modules can be reused and efficiently integrated into accelerator designs.

Abstractions and tools that balance generality, specialization, flexibility, and productivity are required to bring FPGAs to the mainstream. Current FPGA abstractions give unbalanced emphasis to simplifying processing kernel design effort disregarding the memory access side of the design task. However, several recent research projects have developed abstractions for FPGA memory. These projects successfully address the complexities in design portability by abstracting platform details of specific memory devices. Additionally, they provide support for virtualization, coherency, and building memory hierarchies. However, these memory abstractions remain low-level, limiting their support to generic load-store primitives. FPGAs garner much of their capability through highly specialized logic designs; specialization should be extended and featured within memory systems and abstractions for FPGA designers. For example, specialized memory functionalities at abstraction layers higher than standard DMA transfers such as operations on sparse matrices.

The memory as a service abstraction raises the level of abstraction for FPGA memory to provide designers with starting points higher than standard bus interfaces and DMA IPs, or even explicit notions of memory as an array of locations [59]. Memory services are implemented as soft-logic service modules that abstract platform interfaces and data layout, and encapsulate high-level semantic-rich operations on data in memory. Services are portable hierarchically-composed functionalities, only the lowest level services are required to know the details of the memory device where the data resides. Individual services are implemented as separate components within a system—each service can be accessed by multiple clients and share resources such as memory devices. Furthermore, complex high-level memory services are generally composed of simpler services.

Specialized memory operations available as convenient services reduce the development burden on compute accelerator designers. Good designers do practice modular design, but as ad hoc efforts that are not generally reusable. Imposing a service abstraction for memory can help force those efforts into a reusable and portable form. The memory as a service model balances abstraction and customization proficiently to encapsulate in-memory data objects behind high-level, intelligent, access and modification operations. The lowest level services enable processing kernels and other higher-level service portability by abstracting the differences of diverse platforms and memory technologies.

Decoupling memory operations as services simplifies the logic of processing-focused kernels. Services range from simple memory engines supporting atomic read-modify-writes, to traversing and modify-

ing large pointer-based data structures. Furthermore, abstraction overhead in hardware is low—without services, the kernel modules would have to implement these operations internally if not provided as a service. The memory as a service abstraction can improve design results as high-quality, expert-designed service modules are available. Below this abstraction, designers can even selectively instantiate performance improvements through parameters.

3.2 Memory as a Service: Scope and Assumptions

This thesis rooted in an FPGA-first point of view, where the FPGA is the main driver of the application. An accelerator exists on the FPGA fabric that performs some useful operation on data. Any non-trivial accelerator requires data that must be supplied to compute units at a rate appropriate to their performance. It is assumed that the accelerator will access data either in memory or consume data "on-the-move" flowing from external interfaces. Fluid's service-oriented memory architecture makes no particular assumptions about system architecture or specific memory devices attached to the FPGA fabric. Fluid provides an abstraction for memory where designers request high-level operations on data from services outside of the scope of their accelerator modules. The origins of the data managed by these services may be static stored in memory devices, or acquired on-the-move from communication interfaces, or other computation units.

Accessing data is the primary focus of Fluid's memory as a service abstraction. The lowest-level memory services abstract and virtualize these memory devices as read-write services. It is assumed that the FPGA platform has access to some memory device. Memory devices may be as simple as on-chip SRAMs or range from off-chip, on-board DRAM to a host processor's main memory. The specific interface protocol and physical structure of each memory device are implementation details hidden beneath the service abstraction. Memory interfaces may include hardened or soft-logic memory controllers, generic bus interfaces, or cache-coherent interfaces to the processor's memory hierarchy.

Fluid's service architecture is been designed to support FPGA accelerator design efforts. However, the ideas presented in this thesis extend to other hardware development models and even ASIC design, but specifically aim to reduce complexity and decrease FPGA development time. The service abstraction enables designers to rely on previously developed hardware modules in their kernel designs, and integrate their designs at a high-level in Python. This model reduces development time for designers enabling them to describe and generate designs more efficiently than standard RTL development flows.

Designers are not restricted to any particular language by Fluid, however it targets RTL designers and their development process. Service interfaces and Fluid's programming model are intentionally com-

patible with familiar RTL development flows. The designer's kernel modules access services through interfaces at their module's boundary, declared in their port list. Fluid is a structured design methodology that enables designers to encapsulate logic into reusable building blocks and access functionalities outside of their module. Fluid designs are built from logically decoupled modules expressed at a high level. Fluid designs are generated as RTL and can be used in standard synthesis flows. The logic within a module can be developed in any hardware description language as long as the designer commits to Fluid's methodology. SystemVerilog is the primary language supported by and used in Fluid's implementation. However, much of the infrastructure has been created in the Bluespec System Verilog language [10]. Some of the service modules in the library developed for this thesis also have been implemented in Bluespec. It is feasible to consider that designers could develop their kernels and services using high-level synthesis (HLS) tools while adhering to the service abstraction; however support for HLS is outside of the scope of this thesis.

3.3 The Principles of Memory Services

Fluid's memory service architecture was created to answer the question: what should memory look like for FPGA accelerator developers? This question explores the necessary abstractions for FPGA memory that increase programmability and encourage the widespread adoption of FPGAs as compute devices. The FPGA fabric is inherently reconfigurable and garners much of its advantage over general purpose processors through specialization. The service abstraction for FPGA memory is lightweight in order to avoid over-constraining the designer and reducing performance. Services provide high-level semantic-rich memory operations accessible to a designer's accelerator that raise the level of abstraction above generic loads and stores. The memory abstraction layers arise naturally as services are built from decoupled hierarchically composed modules. Fluid's memory as a service abstraction is built on the following set of core principles:

- **Memory Abstraction** Services provide a standardized interface for memory devices independent of that device's physical interface or properties. This principle is essential to support service portability across the increasingly diverse landscape of memory devices. Both service and accelerator developers depend on a standardized memory abstraction to focus their efforts on contributions to their modules rather than infrastructure development. Modern FPGA devices are often connected to multiple memory devices with diverse properties that range from the structure of their physical interface to operational semantics like host-coherency and memory ordering. Standardized read and write services abstract these differences providing a simplified scratchpad interface. However, the

designer can choose to constrain and specialize read-write services through configuration parameters. This approach retains the flexibility of the FPGA logic fabric while supporting designers with simplified access to data in memory.

- **Functional Abstraction** Services act as black boxes that provide some functionality where the logical and structural complexity is hidden from the service's client. Services provide starting points, on the FPGA fabric, higher than loads and stores. For example, services encapsulate logic to access a pointer-based data structure, like a liked-list, in memory and/or the structural differences between the memory device interface and access granularity. Typically, a service developer designs their module adhering to the service architecture and defines its functional contract incorporating it into the library. Services are also built by encapsulating generally reusable logic and/or modules that are architecturally significant to a larger design. Encapsulating functionalities as services increases flexibility and allows the designer to integrate these components at an abstraction level higher than RTL. Additionally, developers may choose to optimize their services underneath the abstraction. A service may have multiple implementations of the same functionality; this is possible as long as the service maintains the same basic functionality.

Through these principles, the service architecture increases programmability and raises the level of abstraction for FPGA memory without reducing performance.

3.4 Why Choose Services as an Abstraction?

The choice of a service abstraction for memory did not come about without careful consideration. The service model is inspired by service-oriented architecture in software design. In service-oriented architecture, software designers abstract the functionalities implemented by their software code as building blocks for larger programs. Software designers call these abstract functionalities services. Each service decouples functionality from implementation—clients consume a service's functionality abstractly, unaware of its physical location or internal complexity. Services are loosely-coupled components that interact with clients through a defined protocol. Multiple services can be composed to create larger and more complex functionalities within a system.

The central question of this thesis is: *What should memory look like for FPGA accelerator developers?* This thesis asserts a change in thinking about memory from unnecessarily low-level read and write primitives to semantic-rich high-level memory operations. This is achieved by a service-oriented architecture that supports specialization and composability within a memory system for FPGA accelerators. Fluid's mem-

ory service architecture allows accelerator designers to flexibly configure and consume high-level memory operations in their accelerator designs. A service abstraction helps accomplish this goal and draws from design patterns pioneered by software developers.

Why does a service abstraction fit for programmable hardware? With the goal of raising the level of abstraction for FPGA memory, the memory as a service model provides many advantages to hardware developers. Services first decouple functionality from implementation, specifically each service is an abstract functionality that has a corresponding implementation in a soft-logic service module. This allows for multiple implementations of a service, encouraging flexibility to fit the designer's goals. The user-level logic requesting a service's operation doesn't change when a different implementation is substituted. Any implementation change that would break the operational contract between a service and client is expressly disallowed. However, there is freedom for different implementations of services with different power, performance, and area characteristics. Abstraction overhead in hardware is low compared to similar design patterns in software designs. Including services in an accelerator design doesn't increase overhead as this logic would have had to be included in the user-logic of the design originally. However, the logic implementing the required memory operation is now encapsulated into a service module. As services are developed and absorb complexity from the user-logic, they are made available in a reusable library for future designs.

The memory as a service abstraction unlocks additional benefits that increase development convenience for service designers as well. Just as an accelerator designer's user-logic relies on services, service designers benefit from services in their designs. The service abstraction encourages composability, where services decouple functionalities that are reusable and shareable by many clients including other services. At the most basic level, higher-level services rely on read/write services, removing the intricacies of whatever memory device stores the data accessed and served by their memory operations. Both accelerator and service designers benefit from the service abstraction and can focus their efforts on their logic rather than on infrastructure and complexity of memory operations already available as services.

3.5 Service-Oriented Memory Architecture

This section describes the architecture defined by Fluid to support service-oriented design practices in FPGA hardware design. FPGA developers require abstraction to increase programmability to construct accelerators that make use of FPGA's enormous processing capacity without exploding development time. The service-oriented memory architecture defines an abstraction and development paradigm for FPGA computing with services as its architectural primitive. By definition, a service *provides* a functionality

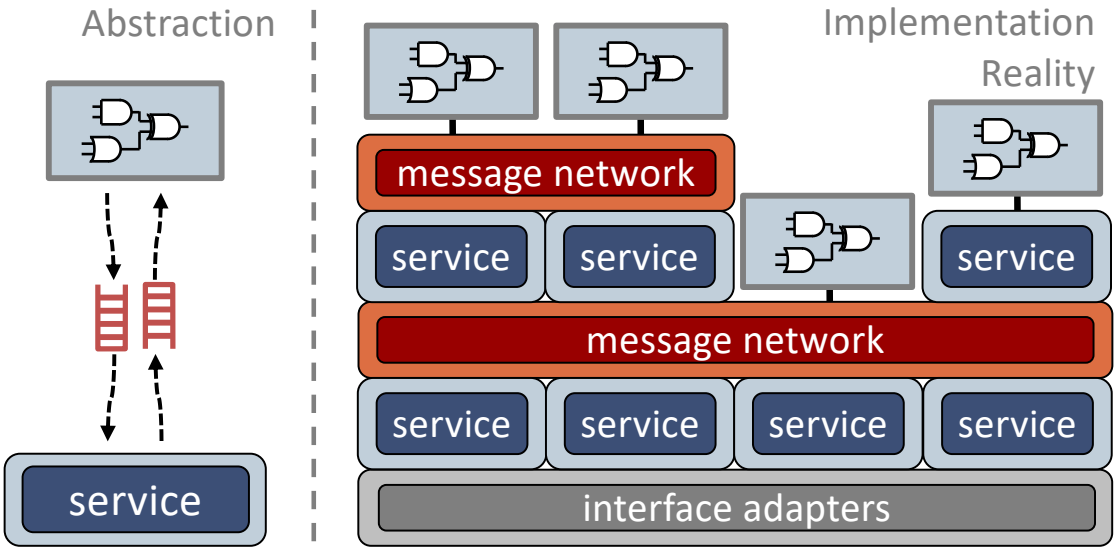


Figure 3.1: Fluid's service abstraction hides the implementation complexity of service modules and required infrastructure.

that a client *requires*. This provides-requires relationship is fundamental to build the operational contract between clients and services. Figure 3.1 shows this abstract relationship on the left. The kernel module at the top relies on an abstract service at the bottom as a client. The service exists outside of the modules hierarchy and is instantiated outside of the typical RTL flow. The service module is a soft-logic implementation of the abstract operation a service provides. In many cases higher-level services are composed hierarchically with other services—this implementation reality is hidden below the abstraction as shown on the right of Figure 3.1.

Disaggregating accelerators through services provides additional flexibility for system construction. Fluid memory services and clients incorporate a latency insensitive methodology for communication [14]. Clients make requests to services synchronously, however the communication and processing delay is variable. This model is essential to support the memory service abstraction and simplify module integration. Latency insensitivity has been demonstrated as a favorable design pattern in hardware designs, and we argue it is essential to future FPGA computing platforms. This effect is discussed further in Chapter 4 which discusses Fluid's communication abstraction. In brief, services are accessible to client modules through well defined interfaces, called channels. A client module declares channel interfaces as a port.

Creating order in the FPGA development process through an abstraction for FPGA memory is not a trivial task. Structure can unnecessarily harm performance and increase complexity if it is arbitrarily defined. The service abstraction was conceived to remain lightweight and flexible, yet encourage a core set of principles to increase FPGA programmability. Increased programmability is provided when applications

make use of services that provide high-level operations and convenient control over memory. Abstraction layers in software often come at a cost, reducing performance with additional instructions. In hardware design, abstraction should be considered differently. The FPGA fabric is a spatial architecture where the cost of abstraction is incurred in logic capacity not time. This trade-off of space for abstraction does not impact compute performance. This impact is especially apparent in the case that accelerator designers must include the logic and infrastructure to access data in memory within their accelerator modules. The methodology Fluid develops to factor this logic out of their designs and into service modules does not increase performance or area overheads. Abstraction overhead in hardware is low—without services, accelerator modules would have to implement these functionalities internally.

The basic support for the service abstraction, and much of its initial motivation, stems from the rich and diverse set of memory devices available on current FPGA platforms. Each device comes with its own control interface and bus protocol. The service abstraction mitigates this heterogeneity with structure. The primitive memory services enable read and write functionality across memory devices. Read and write services are the foundation of all higher-level services and enable a fundamental portability layer. Fluid's service-oriented memory architecture not only simplifies accelerator development efforts but also abstracts the underlying platform interface details. Decoupling this architecture from platform details enables portability across complex interfaces such as cache-coherent interfaces to host-shared memory and can extend support to future FPGA platforms with relative ease.

The memory service abstraction extends beyond increasing portability, it enables a virtualized model where multiple independent read-write services share a single physical memory device. This is a key component to the memory architecture that is important for customizability and parallelism. Supporting virtually independent memory channels through services encourages designers to specialize their memory infrastructure decoupling independent memory traffic. Designers can choose a higher priority for services sharing a memory device or selectively instantiate performance enhancements. For example, accessing data in a pointer based-data structure would best be served by a read service with high priority close to the memory interface for the lowest possible latency. Additionally, a small cache could increase performance if the access pattern exhibits locality. A client with a read-modify-write access pattern would also benefit from a cache to capitalize on the temporal locality. Bandwidth hungry clients would be assigned to lower priority services sharing the same memory device containing the pointer-based data structure.

Arguably, requests for data have always been a communication with a simple service, memory. However, services define an architecture that enable higher-level memory operations inside an accelerator's memory system. Simple services support basic data transfers such as a block copy or linear streams from an array. More complex services manage synchronization and order of memory operations requested by a

number of clients. Services even abstract objects in memory allowing hardware modules to interact with queues in memory as if they were on-chip FIFOs. Services hide the implementation details of complex data movements and representations from accelerator developers—this greatly simplifies user-level code. This type of support is typically only possible in software, however Fluid provides a methodology to encapsulate these functionalities and provide them through a library of services. The memory as a service abstraction does nor force or limit the types of functionalities that can be encapsulated as services, however this thesis does offer guidance on organizing services and decoupling functionalities encouraging modular composable design patterns.

The service abstraction also encourages specialization and optimization of a service module's implementation. Raising the level of abstraction for FPGA memory requires a delicate approach to enable specialization without reducing efficiency. A given service may have multiple implementations of the same functionality with varying degrees of performance. The abstraction requires that each service module must implement the same logical functionality for the service. Fluid's design framework allows accelerator designers to choose the service implementation tailored to their performance and resource constraints. Therefore, the service abstraction does not limit the flexibility of the accelerator design as design constraints change.

3.6 Composing Services Hierarchically

It is important to understand the possibilities unlocked by Fluid's service-oriented memory abstraction. In order to make sense of and build service-oriented systems, this section describes a hierarchy for services. The hierarchy develops a framework for service developers for disaggregating their service designs, encouraging reusability and composability. Services should be developed hierarchically, increasing the abstraction level at each layer. This simplifies the development process as higher-level services can reuse services from lower abstraction levels. Service-oriented architecture has demonstrated capability in reducing complexity and increasing development convenience for software developers. Enforcing a structured interface architecture for service modules in FPGA design brings the advantages of service-oriented architecture to accelerator development. Services provided to FPGA application developers absorb complexity and simplify the development efforts. Similarly, composing service operations by relying on the hierarchy of services simplifies the development process for service designers.

The memory as a service abstraction is intentionally structured to simplify integration, encourage modular design, and code reuse. Modular design is considered a best practice in hardware development however most designs are ad-hoc highly-specialized efforts that are not generally reusable. Every module

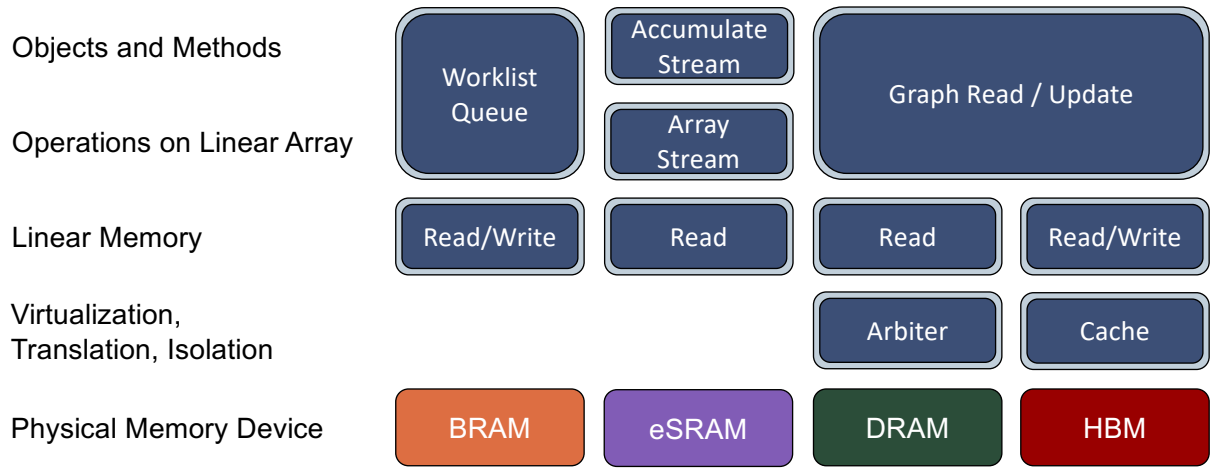


Figure 3.2: The hierarchy of composable Fluid service modules

in a Fluid design must adhere to a structured interface specification for inter-module communication within the abstraction. The structured interface specification is called a channel, declared in a module's port-list, and relies on a latency-insensitive design style. Service modules receive and respond to requests for their services through channel interfaces. A channel is tied to a specific operation type in the service module's specification. A service developer may choose to provide multiple services by their module, either on multiple channels or on a single channel. If multiple services are provided on a single channel, the request's arguments are used to identify and control the operation that is performed. Below the abstraction, a service may require support from additional services to satisfy a service request. The service module must declare additional channels to send requests to other services providing these functionalities.

As an example, a block copy service module has three channels in its port-list: command, read and write. The block copy service is provided on the command channel, the service receives requests to perform the block copy operation on this channel and responds to signal completion. The read and write channels require read and write services that provide the operations recognizable by the same name. This design pattern is intentional where each service can layer increasingly complex and semantically rich operations upon one another. Composable service operations are essential to absorb complexity from higher-level services, simplify development, and encourage general reusability and portability.

The composable service paradigm creates a hierarchy of functionality that service users and developers can rely on. In order to organize and direct this design pattern this thesis describes a hierarchy that provides a roadmap for service interactions, organizes services, and directs future service development. Figure 3.2 shows the service hierarchy including example services to illustrate the layer boundaries. Each vertical column is a hierarchically composed stack of services. The stack consists of four abstraction

levels on top of physical memory device interfaces. The layers include physical memory devices, virtual translation and isolation, linear memory, operations on linear memory, and objects and methods. Each layer is increasingly abstract as it moves away from the physical details of the memory device. Within a hierarchically composed stack services communicate to satisfy a request providing the desired response. This can include communication to access or process data, or orchestration communication that pass commands and metadata across layers.

Physical memory devices make up the the lowest level of the hierarchy. In order to simplify the explanation, the hierarchy describes the stack in the scope of memory. However, the layered abstraction approach extends to data on-the-move between accelerator modules and from external communication interfaces. This physical layer is not considered part of the service abstraction it defines the means of storing data accessed by the FPGA accelerator. Each physical device memory is accessible by own physical bus interface such as AXI or Avalon. Bus interfaces are similar in operation but often vary in protocol and construction.

The virtualization, translation and isolation (VTI) layer consists of soft-logic infrastructure to adapt vendor bus interfaces to a common interface. At this layer of the hierarchy designers specify the kind of physical memory device and how these devices are shared. This layer of abstraction enables virtualization for memory device types, address mapping, and performance enhancements. As services must adhere to a functional contract these changes transparently effect performance without impacting functionality. Additional support to share physical interfaces is provided at this level through arbitration infrastructure. This allows multiple services at the next layer of the hierarchy to share a single memory interface. Again as each service must maintain functional correctness, so this appears transparent to higher level services. As a part of the memory infrastructure designers may specify caches that are instantiated to improve performance.

The next layer in the hierarchy is the linear memory level. This layer provides the memory service primitives essential to the service memory architecture namely read and write services. These are the basic building blocks that all higher-level memory services rely on. Their operation defines a request-response protocol for read and write operations to a linear address space in memory. The VTI layer allows each read and write service to adapt to different physical memory devices. Each read-write service provides a virtual scratchpad in memory. The arbitration infrastructure in the VTI layer enables each read and write service to provide an independent scratchpad for a region of memory. As a common design practice, separate read and write services are used to access independent data structures or parts of a data structure. For example, in a design with a linked-list that stores its payload separately accessed via a pointer, one service could be responsible for the list and another service accesses the payload. Separating

data structures with independent services enables the familiar transparent separation software developers have come to expect through instantiating independent objects in their code. While this separation does consume area, it simplifies higher-level services and application user logic.

Operations on linear arrays and memories make up the next layer of the hierarchy. At this layer services begin to provide semantic rich operations that absorb complexity from the user logic of accelerator designs. Services at this layer are developed to adapt linear arrays to common useful access patterns and operations. As examples, stream services generate streams of data responses from a single request message. This may be a linear stream from an array, or stream from a pointer-based data structure like a linked-list. Another service in this layer is the atomic read-modify-write service responsible for atomically updating data within the scope of the read/write service it is hierarchically composed upon. Atomic operations are incredibly useful to simplify programmer's responsibilities when developing applications however they are not provided by memory controllers on FPGA platforms. FPGA memory interfaces are intentionally unconstrained allowing accelerator designers to specialize their memory systems to suit their applications. Through this layer the ideals of a service-oriented memory system come to light. Services absorb logical complexity beyond the infrastructural complexity of platform interfaces. Working within the Fluid methodology service designers should target their development efforts at this layer and above. Their modules benefit from the portability provided by more primitive read and write services, and they can focus on their contributions rather than infrastructure.

The highest layer described in this hierarchy is the objects and methods layer. These services abstract away linear memory entirely and virtualize functionalities with abstract requests. The functionalities provided by these services range from data structures, containers, traversals, and processing. An example of such a service is the "worklist" queue. This queue is an abstract container for work items which are sub-interface-granularity data items. This service accepts work item data as a request and provides this data back in FIFO order as a response. While the logical complexity to the user is simple, the implementation reality is quite complex. In a typical implementation the user must manage all of the complexity themselves and develop the logic in their application design. Provided as a service, the worklist functionality is generally reusable and portable across memory devices. One source of this complexity is the disparity between the work item and memory interface granularity. This is apparent as the worklist service provides elastic capacity by spilling work items to a backing memory device, such as DRAM. The worklist service manages the granularity difference between work items and the memory interface to efficiently pack multiple work items per interface block.

The objects and methods layer is not the last possible layer imaginable in a hierarchy of memory services for FPGA designs. A designer could encapsulate an entire accelerator as a service, however any

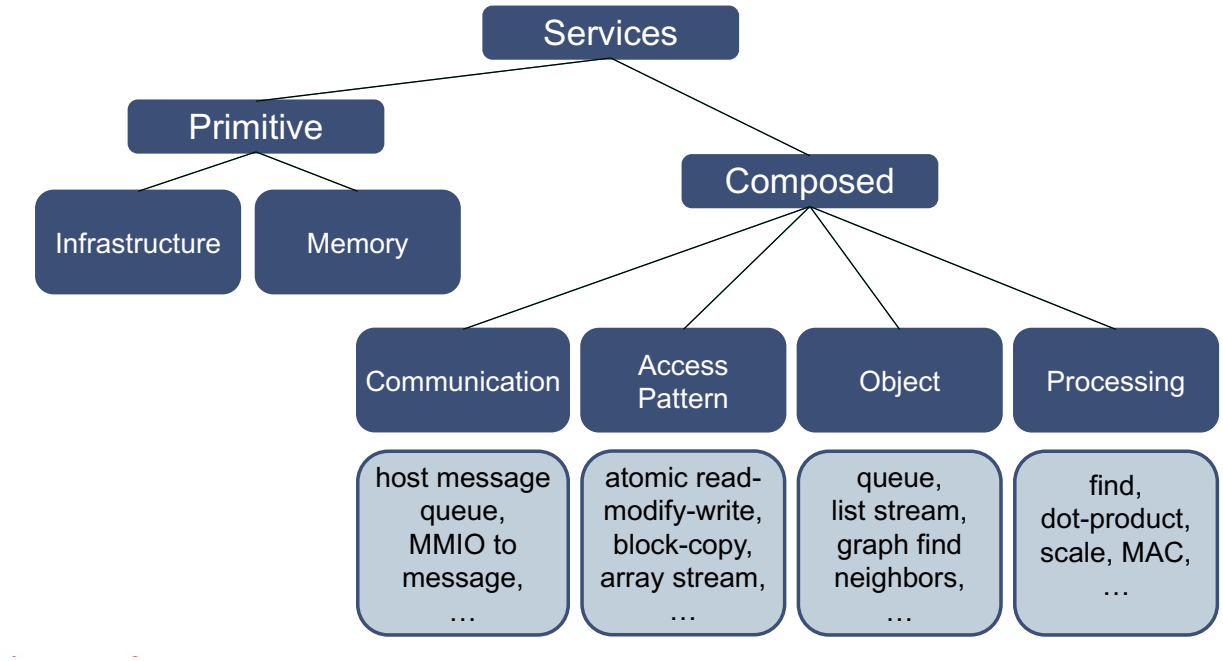


Figure 3.3: A taxonomy of Fluid services

further discussion would be a departure from the main topic. Furthermore, the composition and disaggregation framework, as described, would extend to larger and more complex functionalities abstracted as services.

3.7 A Taxonomy for Services

A service taxonomy is discussed in this section to categorize services further in order to understand the types of operations that are currently supported and organize future extensions to the service library. Functional abstraction and composability are major tenets of service development used to increase FPGA programmability. The Fluid design methodology encourages these practices allowing hardware designers to logically compose their modules into larger accelerator designs. The hierarchy described in the previous section directs Fluid service developers to consider and rely on composability in their service module designs. Figure 3.3 details a taxonomy to classify the functionalities provided by Fluid services, to describe the types of functionalities services can provide and their breadth of coverage.

Primitive services support the Fluid service abstraction. This set of services virtualizes the platform details of memory and communication devices. Primitive services are the foundation of Fluid's framework to enable platform and memory device portability. This set of services is intended to support accelerator and service development and are provided as a part of Fluid's framework designed and implemented as a

working prototype in this thesis. As discussed in the last section, read and write services are the primitive abstraction for memory that higher-level memory services rely on. Primitive services allow designers to focus on their contributions to their accelerator and/or higher-level service modules, rather than on the surrounding infrastructure. Expert designers may choose to spend their development time towards primitive services to optimize performance or to port their abstraction to new platforms. Cache services are one memory primitive for performance flexibility. Designers can specify caches around read and write services to optimize the performance of their service memory systems. Primitive services should be highly parameterizable to fit a variety of use cases and to ensure performance is not unnecessarily constrained by the designer's choices. Infrastructure services support system design, integration, and data communication. Primitive infrastructure services range from channel FIFO buffers to full Network-on-chip IPs. Infrastructure services support the communication abstraction described in Chapter 4.

The second major categorization for services are composed services. This set consists of all services that rely on lower-level services decoupled from physical device interfaces. Composed services begin to absorb complexity from user level accelerator logic. Each of these services, at a minimum, are composed with primitive services for memory and/or communication. Additionally these services may be composed with each other as detailed in the hierarchy. The service abstraction is structured to enable physical and functional composability. Physical composability ensures that clients and services can be physically connected in hardware. Functional composability supports the functional abstraction services provide where clients can rely on services to build higher level functionalities.

The first category within the composed branch continues to support Fluid's abstractions but relies on primitive services. Communication is essential to the memory as a service abstraction and communication protocols can be composed with primitive services to simplify accelerator designs. These services provide support for communication protocols through memory devices. Communication services enable multiple methods of host processor involvement in Fluid's abstractions. For example, typically a host processor can transmit and receive messages to the FPGA hardware through memory-mapped input-output (MMIO) operations over an IO bus or JTAG. The "MMIO to message" service creates requests from MMIO words and forwards them to the targeted service. These requests can be used to, for example, initiate a block data transfer from the host to local memory. This simplifies service development by maintaining the memory service abstraction even in the host processor's code. Additional communication services manage continuous streams of data through circular buffers in memory.

Access pattern services encapsulate logic for common memory access patterns. This category is broadly defined by services that transfer data without modifying and/or obfuscating its underlying structure. The client is still aware of the underlying representation of the data in memory. Access pattern ser-

vices typically require multiple memory transactions to complete their requested operations. For example, a service that streams data words from an array in memory to an accelerator kernel as a stream. Stream services transfer data to and from memory in a simple sequential access pattern. Block-copy services similarly transfer bulk data from one memory location or device to another. Not every access pattern service is a bulk transfer, read-modify-write is a classic atomic operation useful for updating smaller granularity words of a larger data block. This access pattern requires the service to issue a read followed by a write request to complete the read-modify-write action.

At a semantically higher-level, object services abstract state and data representations but also the behaviors defined by objects. These services support high-level operations typically only readily available in software. Object memory services provide abstract data containers and operations on data structures such as lists and graphs. Object services are highly specialized like the graph service. The graph service abstracts the data representation in memory and provides compound operations that supply relevant operations like fetching all neighbors of a source node. Object services greatly simplify accelerator designs and provide the level of convenience that software developers have come to enjoy from object-oriented programming libraries.

The service abstraction intentionally decouples concerns of processing and accessing data in memory. Fluid's memory abstraction encourages accelerator designers to think spatially in their design process requesting operations from services outside of their design hierarchy. The processing category for composed services incorporates processing into the service abstraction's scope. These services naturally compose with access pattern and object services which supply data through streams. For example, a streaming multiply-accumulate service accepts streams of floating point data to produce a result. Designers can compose multiple processing services as filters or layered transformations. Even large processing modules, once encapsulated into services, are reusable within the Fluid framework.

3.8 Services are Smart IPs

The service abstraction allows designers to make use of abstract functionalities in their designs implemented by service modules. However, this basic notion of a service is only part of the picture. Each service should be developed as a "smart IP". This notion is derived from the concepts developed by the Pandora paradigm [71]. The Pandora IP development paradigm establishes a model for smart IPs that are highly parameterized, generatable, and provide support for functional and performance debugging. The goal is to support developers to build highly specialized IPs that are easily tuned for a better overall design.

Fluid's service-oriented design methodology is inspired by the Pandora paradigm and influences its IP development principles. Fluid further develops a methodology to compose smart IPs into high-level functionalities and entire systems of IPs as services. Fluid extends the Pandora paradigm to system composition as well. The entire system of services and their composition is highly configurable facilitating a specialized system of functionalities available to FPGA accelerator designers.

Fluid services strive for the goals outlined in the Pandora paradigm and should be developed as smart IPs. To this end, each service should be parameterized, even so far as to have entirely different implementations of the same functionality with various levels of performance and/or underlying assumptions. For example, a service providing a stream of data words can support a variety of implementations for different data structures such as arrays, linked-lists, or graphs. The client requiring the data stream does not change based on the representation of the data, the system designer simply chooses the appropriate service implementation. Additionally, the service should be parameterized to support data streams with different widths or modify its internal buffering capacity.

In order to support additional flexibility, services in Fluid designs are not intended to be static, monolithic blocks. Each service is composable and changes to a service's implementation can therefore effect services that rely on it. High-level services and systems can be optimized through replacement. This process introduces increased complexity in integration and debugging. Therefore service modules should include instrumentation and introspection hooks for dynamic debugging and performance tuning. The fundamental read-write memory services provide a number of introspection features for accelerator designers. Intelligent statistics registers allow designers to probe the read-write service's performance as it relates to the rest of the design. For example, a designer may want to compare the request delay of reads before and after configuring a cache in their system. These statistics are available at run-time, after synthesis, when the instrumented version of the service module is used. The instrumentation hooks are exposed to the designer through configuration and status registers. These registers are connected to a centralized control and access module that enables access by the host CPU. The designer can then profile their designs at run-time using MMIO or JTAG to access the registers.

Chapter 4

Communication as a Service

This chapter presents Fluid's communication as a service model, a flexible, structured abstraction that facilitates communication between modules in an accelerator design. The communication abstraction begins at the module boundary allowing for abundant infrastructure flexibility. Connections between modules are logically point-to-point between their structured interfaces. The communication as a service model leverages abstraction to elaborate connections between modules across a variety of physical implementations. These connections are supported by a library of configurable IPs, substrates, and interfaces available to FPGA designers. Each physical connection implementation can be substituted transparently and specified by the designer separately. This allows designers to partition applications across multiple FPGAs and/or migrate services to host processor cores.

4.1 The Need for a Fluid Communication Architecture

Moving data into computation kernels is an essential, yet challenging component of FPGA accelerator design efforts. This includes accessing data in memory and communicating data between modules in an accelerator design. The previous chapter described an abstraction for accessing data in memory. The memory as a service abstraction is built around the central tenet that structure with flexibility increases FPGA programmability without reducing the efficiency FPGAs provide through specialization. Communicating data between components in a design is just as important as accessing data in memory, and similarly follows stylized but identifiable patterns. Abstractions increase programmability and fit best when applied at natural boundaries. We leverage a similar approach from abstracting memory applied to communication. Tightly coupling Fluid's communication abstraction with its memory abstraction increases flexibility, and allows memory services to be generally accessible, hierarchically compose and be scale within a system.

Today's FPGA developers are faced with a new reality as FPGA devices continue to scale and gain capabilities. This new reality is introduced by long wires on large FPGA devices and novel FPGA system-level integration schemes. Software developers rely on abstraction layers to hide complexity in order to make use of complex hardware and system designs. Fluid provides a communication as a service model to abstract communication in hardware designs so that designers spend less time on infrastructure and more time developing their compute kernels.

Wire delay has long been a topic of concern for VLSI circuit designers [39]. As process technologies continue to scale towards smaller transistors, wire delay dominates logic delay. FPGA architectures have attempted to mitigate this concern by transparently inserting registers along long wires to increase the clock frequency [54]. However, as scaling continues FPGA vendors are increasingly turning to multiple chip FPGA devices integrated on an interposer to increase manufacturing yields. Communicating data across these boundaries requires additional buffering. FPGA synthesis tools currently handle these concerns for designers inserting buffering to manage clock frequencies and structural challenges. However as scaling continues long wires are no longer a viable solution. FPGA designers must consider wire length in their designs as congestion and routing increasingly challenge synthesis tools. FPGA designers are also turning to latency-insensitive communication between hardware modules to combat the challenges of long wires [14]. Designers use this techniques to increase flexibility and insert additional buffering while retaining the synchronous nature of hardware design.

Novel system designs, both on FPGA SoCs and in data centers, present new opportunities for FPGAs as compute devices. Cutting edge FPGA devices provide hardened network on chips (NoCs) to combat scalability and heterogeneity, however these networks require additional effort to be used in accelerator designs. Additionally, FPGAs are now integrated into data processing systems inside the network infrastructure, storage devices and on processor interconnects [44]. These unique placements present many opportunities and unique challenges, especially in the datacenter. Designers must account for disaggregated systems with multiple FPGAs, numerous off-chip memory devices, and communication interconnects. Flexibility matters in this landscape. Accelerators on FPGAs may even have to communicate and share data with heterogeneous processing units which introduces additional complexity and challenges. Furthermore, as FPGAs are added as a "bump in the wire" in data centers, they become the center of their infrastructure to inspect and process data as it moves through the network.

Designers need an abstraction to manage this complexity and focus their design efforts on accelerator design rather than infrastructure. This thesis invokes a unified communication model that enables flexible infrastructure and simplified development. All agents within this framework must adhere to a message passing model through FIFO delivery channels. Beneath this communication abstraction infrastructure

can be varied to fit the application requirements. Furthermore, hardware can extend beyond the scope of a single FPGA. Even software on the host processor can transparently communicate with hardware modules in Fluid's communication as a service model.

4.2 Challenges in Developing a Communication Architecture

Designing a flexible communication architecture is not a trivial task. This task is compounded by the requirement that the communication abstraction is tightly coupled with a memory abstraction on reconfigurable hardware. FPGAs present an enormous opportunity in their reconfigurability, while at the same time challenging the designer as devices grow and increase in complexity. Constraining FPGA designers to a single abstract communication paradigm enables portability across communication substrates. However, this level of portability necessitates a library of IPs and shims to implement connections through the many external communication devices and resources available to the FPGA logic fabric. Much of the intellectual and engineering effort to develop a communication architecture for Fluid was directed towards supporting generatable and adaptable module interfaces. Generatable hardware requires structured, flexible IPs both for service modules and communication channels. The semantics and structure of interfaces both on the FPGA and to external resources first had to be constrained to a common interface. This basic support was necessary to enable module composition through abstract communication channels and support a latency insensitive communication model for Fluid memory services.

Adapting interfaces and bus widths is only part of the heterogeneity problem. Both communication channels and service modules introduce irregularity. Each communication channel has a corresponding implementation and physical structure, coupled with the logical semantics to connect a client with a service. This requires additional verification to guarantee correctness and reduce accidental integration complexity. Fluid develops a framework for service composition and channel elaboration with *provides* and *requires* keywords. This simplifies the service-to-channel and connection type verification process. A service is *provided* by a service module on a particular channel interface. Therefore, the channel interface has a specified structure and provided operation. A client specifies that a particular service is *required* on a given channel interface. For example, when a service *provides* a stream of data, the specifics of the data layout and format in memory and the logic handling the operation are transparent to the client. The client simply *requires* and expects the data stream on a given channel. It is through that specific channel that requests are made and satisfied with response(s).

Mapping a channel to a service is admittedly restrictive. However, client logic is simplified as one service is expected on each channel. Furthermore, each channel has a uniform interface specification and

protocol for composability. This supports Fluid's design framework to elaborate physical connections from logical connections with reduced effort. However, the channel interface is still parameterizable; the designer is free to choose bit widths for data and metadata at a granularity appropriate for their service module's functionality. This flexibility is intentional to allow a designer to customize the channel interface to their modules and retain the advantages of the FPGA fabric to specialize for efficiency and performance. This trade off limits overhead while supporting the Fluid methodology's logical communication model for generatable FPGA systems.

4.3 The Principles of Communication as a Service

This thesis presents a model for FPGA communication that enables flexible and logical communication between components in FPGA accelerator designs. Designers logically specify connections between modules in their system at a high level. The abstraction transparently supports multiple physical connection types (wire, FIFO, and channel) between client and service modules on the FPGA fabric. The abstraction extends to the host processor to simplify development through emulated software implementations of services. This allows designers to focus on their contributions rather than on infrastructure. Fluid's communication abstraction is built on the following principles:

- **Communication Abstraction** As a part of Fluid's design methodology modules must adhere to a standardized interface called channels. The channel interface supports abstract in-order communication between modules in a Fluid service-oriented design. Channels are latency-insensitive and support a loosely-coupled modular design style. Similar to services, channels decouple their functionality from physical implementation. Any given channel may be implemented by a multitude of physical communication substrates from wires to networks. Fluid's communication abstraction is tightly coupled to the memory as a service abstraction and each channel is delineated with a channel type that ascribes a service to that channel. Leveraging channels, modules access functionalities outside of their internal hierarchy within the abstracted surrounding infrastructure. Larger services, comprised of multiple disaggregated service modules, are hierarchically composed through channel connections. Additionally, clients are only aware of a service's existence not its physical location or implementation.

- **Hardware-Software Abstraction** Communication as a service enables the service abstraction to extend beyond the scope of a single FPGA device; this includes software on a host processor. Just as services enable functional abstraction encapsulating FPGA logic, software implementations of ser-

vice functionalities are available to accelerator designers. The channel-based communication abstraction extends to software services implemented by a thread on a processor core. Software services emulate a service module's logic replicating its functionality in software. Hardware and software services can exist simultaneously and communicate transparently. This allows for additional design flexibility and simplified development efforts.

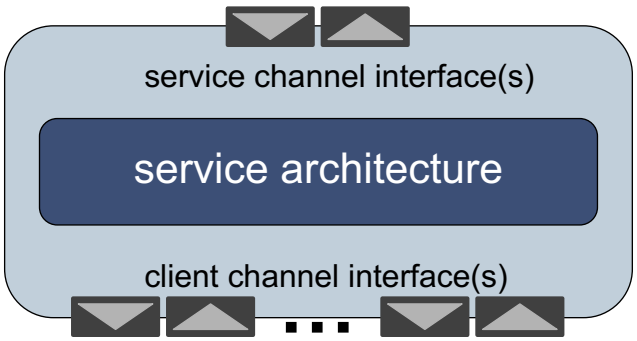
Fluid's communication as a service paradigm simplifies FPGA development efforts through its flexible abstract communication methodology. It increases FPGA programmability and simplifies design integration through developing and realizing support for these principles.

4.4 A Unified Communication Interface

Channels are the primary interface between all components within a Fluid design. A channel is an interface to a module that exists somewhere else in the system. All communication within Fluid's design methodology flows through channels as shown in Figure 4.1. It is a common practice to decouple modules with FIFO interfaces in accelerator designs and FIFO semantics are familiar to RTL designers. Fluid adopts FIFO semantics for its channel interfaces. The channel is a sender-receiver pair for request and response messages between service and client modules.

Channels follow the same decoupled design philosophy as memory services whereas channels, as an abstraction, represent a logical connection between modules decoupled from their physical implementation. This enables enormous flexibility in the infrastructure integrating modules and their supporting datapaths. Designers can flexibly integrate modules in their designs and choose from a library of channel implementations. The channel abstraction relies on the concept of latency insensitive communication and guarantees reliable, in-order delivery of messages. However, no guarantees are made about the timing or structure of the physical implementation of a channel. The channel abstraction allows for flexible accelerator implementation and optimization of the communication hardware.

Figure 4.1: Communication channels shown as FIFOs between a user-level kernel module to the left and service modules to the right.




```
interface channel
   interface txMsg tx;
   interface rxMsg rx;
endinterface
```

Figure 4.2: The memory service architecture in which processing kernels interact with abstracted memory services through channel interfaces.

4.4.1 Service Channel Abstraction

The channel interface was developed with composability and flexibility as first-class concerns. Channels are tightly coupled to the memory as a service abstraction and are fundamental to the architecture of service modules. Fluid's major architectural constraint is that services must communicate through channel interfaces within the system. A service is exclusively assigned to a specific channel; all requests and responses for this service are assigned to that channel. However, the service module architecture is not a rigid shell, it is a flexible black box for soft-logic service functionalities within the FPGA fabric. A rigid shell is understandable at the FPGA's edge where designers must be concerned with physical interfaces and timing constraints. Services do not constrain designers to a set number of interfaces. Service modules can provide a number of service functionalities assigned to multiple independent channels.

Structure provides regularity that forces modular design practices to enable disaggregation, composition, and reusability. Figure 4.2 shows the service architecture and interface RTL specification. A service module implementing a service must declare a channel for each service it provides. First-class support for composability is essential to the service abstraction, service modules declare client channels to communicate with any downstream services that are required. Similarly, client modules declare channels targeting service modules providing these services. Clients and services view each channel as a logically point-to-point connection between their channel interfaces.

Channels are request-response pairs to transmit and receive messages between clients and services. Typically, each outgoing request is met with an incoming response, it may simply be an acknowledgment, or can hold some greater meaning. As examples, a write request would effect a response that the data has been effectively stored in memory; a request to traverse and read a pointer-based data structure would return a data stream in multiple responses. Additionally, the request-response pair is separable and may be held by independent client modules, for example a module may issue requests and another module

processes the requested data. Forwarding responses provides a signalling mechanism that can be used to efficiently decouple accelerator stages. The accelerator stages are logically linear, however multiple services including memory requests may participate to assemble data relevant to the receiver.

4.4.2 Generalizing the Channel Abstraction

The previous section focused on channels in the context of service communication however the channel abstraction is much more general. Channels are extended to all components that communicate within Fluid systems. Modules in Fluid systems are spatially distributed across the FPGA fabric. The channel interface inherently affects the notion that abstract functionalities are requested and consumed from black boxes located somewhere in the system. Designers are not concerned with the implementation realities of the communication substrates or physical placement of their modules. Service functionalities are portable across platforms by relying on channel interfaces and Fluid's infrastructure supporting abstract communication.

As a general communication interface, channels support protocols beyond the request-response protocol required for memory service communication. Channels support a general message passing abstraction for virtually any data between modules. Messages can be used much more generally, even without requiring a response. Streaming messages between channel interfaces is a specialized subset of a channel's capability for continuous data transfer. Streaming messages often encode start and end of stream information in their metadata. In fact, memory services often generate and consume messages without adhering to a strict one-to-one request-response model. It is also important to note that a request for a given service may generate additional downstream work. Services are often composed hierarchically, lower-level services may send many requests to complete a request on behalf of the requester.

The channel abstraction applies even more generally to communication infrastructure. Fluid's communication architecture maintains the channel abstraction for infrastructure components like its network-on-chip (NoC). The NoC supported in the communication architecture is a highly parameterizable soft logic implementation. However, the network maintains the channel abstraction to deliver messages between modules. The Fluid framework also maintains the channel abstraction and interface for communication between services outside of the FPGA fabric. This includes software implementations of services on host processors. The channel abstraction and transparent communication is maintained for both hardware and software.

4.4.3 Logical Connections

The design of a standardized communication interface for components in Fluid systems affects many of our objectives. Abstracting communication decouples services and clients simplifying development while maintaining implementation flexibility for optimization. We support interactions between client and service modules through message passing over logically point-to-point latency-insensitive [14] channels. Fluid's design methodology establishes logical connections as the primary construct for connecting channels.

Logical connections provide a simplified communication model that allows for enormous flexibility and specialization within the communication infrastructure. Logical connections support flexible substitution of connection implementations to make use of the enormous capabilities of FPGA devices and their external interfaces. Large modern FPGAs require designers to adapt their design patterns to fit a new design paradigm. Design complexity increases as accelerators scale to larger devices and design efforts are bogged down by integration time. Moreover, long wires across FPGAs are discouraged as they degrade performance and increase routing congestion. Upcoming FPGAs feature hardened infrastructure and network on chips to address this communication challenge. The fact remains, developing infrastructure is tedious and inflexible at the RTL level. Adhering to Fluid's logical connection abstraction and channel interface allows designers specialize their infrastructure and adapt their designs without modifying the modules responsible for their core computations. Infrastructure is generated from a high level description that can be efficiently modified to adapt the accelerator to satisfy design constraints.

Designers must agree to the concept of logical connections between modules in their design when working within the Fluid abstraction as shown in Figure 4.3. The physical details of channel connections are transparent to designers. Each channel guarantees FIFO delivery of one or more messages. Designers must also be aware of and account for latency insensitivity in their designs, substituting different channel connection implementations cannot impact the functional correctness of the design. Any channel implementation can be substituted by specifying the desired connection type in the Fluid design framework.

Latency insensitivity allows the designer, through Fluid's design framework, to control the substitution and implementations of channel connections. Channels may take shape through a wired link, FIFO, network on chip, Ethernet network, or in-memory circular buffers. Modules linked by logical connections are eventually elaborated by the design framework with a physical implementation. These connections join a channel interface that provides a service to a channel interface requiring that same service. It is important to distinguish that these connections are not arbitrary, they must maintain the service tied to the channel. However, the physical details of a channel are free to change as the designer optimizes their

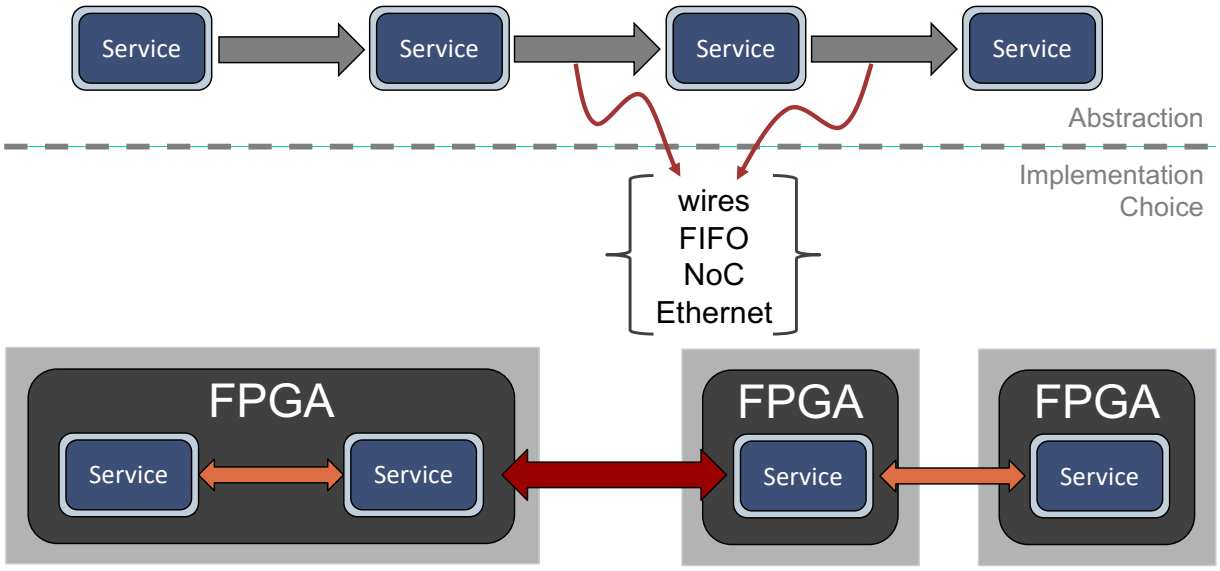


Figure 4.3: Logical connections communication abstraction and implementation reality where connections are elaborated with wires, a FIFO, NoC, or Ethernet implementations and services are placed across multiple FPGA devices.

system. Fluid eases module integration efforts as designers can optimize their data-paths at a level much higher than RTL. Furthermore, the communication as a service abstraction allows designs to incrementally adopt new technologies such as hardened NoCs and inter-FPGA network interfaces. Fluid provides IPs to support multiple connection types for transparent communication across FPGA devices in multi-FPGA systems. Logical connections encourage designers to experiment with new connection types and novel placements of modules across their systems.

4.4.4 Channel RTL Interface Specification

The channel abstraction is implemented by a corresponding SystemVerilog RTL interface. The interface follows standard RTL syntax and design practices. This design choice was intentional so that modules working within Fluid's design methodology remain RTL compatible and much of the "magic" occurs outside of the designers' modules. Both client and service modules communicate with each other through channels with the same interface syntax. The interface's SystemVerilog syntax is a transmit and receive pair with FIFO handshake signals:

```
1 interface channel
2     interface txMsg tx;
3     interface rxMsg rx;
```

```
4  endinterface
5
6  interface txMsg;
7      t_msg msg;
8      logic tx;
9      logic txFull;
10 endinterface
11
12 interface rxMsg;
13     t_msg msg;
14     logic rxPop;
15     logic rxEmpty;
16 endinterface
```

FIFOs are a natural and common structure found in hardware designs. FIFO buffers are often used to synchronize and pass data within hardware modules and across boundaries. Channels conform to the familiar FIFO semantics, with a few key distinctions. Channels correspond to a particular service and may have dynamically-variable latency. Channels are the primary abstraction for communication within Fluid and designers instantiate these interfaces at the boundary of their modules as if they are instantiating service modules. These boundaries are a natural fit for generation, and Fluid's design framework implements and elaborates channel connections from the designer's high-level specification.

Providing a clean common interface simplifies module integration and reduces hardware design costs. Furthermore, the channel interface limits abstraction to the boundary of a module thus limiting the impact the abstraction has on the development process and module's internal logic. The channel interface is structured to simplify the design framework's code generation complexity and enables accelerators to be specified at a higher level of abstraction. However, providing a standard RTL interface does introduce some limitations. RTL's abstraction is explicitly structural at the bit and cycle level. An argument can be made for continuing the abstraction inside the RTL designer's modules with abstract communication interfaces. This would require the generator to analyse the intended communication pattern and rewrite the module's interface. This is a valid approach that would provide additional implementation flexibility, however it is outside of the scope of this thesis. Fluid's design methodology successfully supports FPGA RTL designers and simplifies their development efforts without requiring a drastic departure from standard practices or introducing overheads.

Bluespec SystemVerilog Channel Interface

Fluid abstractions are language agnostic, not limited to any particular hardware description language. Bluespec SystemVerilog is a high-level hardware description language based on atomic rules to move state rather than clock cycles [10]. The Bluespec compiler handles the scheduling of rules for the designer and provides strong support for static elaboration and type checking. Bluespec reduces design efforts and increases productivity for hardware designers. Bluespec is supported as a language in Fluid. Some library service modules and the network on chip were developed in Bluespec. Interfaces in Bluespec are an important language construct to define modules and bundle wires. The channel interface for service modules written in Bluespec has the following syntax:

```
1  interface Service#(type req_typ, type rsp_typ);
2          interface TxMsgChannel#(req_typ) txPort;
3          interface RxMsgChannel#(rsp_typ) rxPort;
4  endinterface
5
6  interface TxMsgChannel#(type req_typ);
7          (* always_ready *) method Bool   txFull();
8          (* always_ready *) method Action tx(req_typ m);
9  endinterface
10
11 interface RxMsgChannel#(type rsp_typ);
12         (* always_ready *) method Bool    rxEmpty();
13         (* always_ready *) method Action  rxPop();
14         (* always_ready *) method rsp_typ rx();
15 endinterface
```

It is not necessary for accelerator or service module designers to have prior knowledge of, or use Bluespec in their design flow. However, Bluespec SystemVerilog is available as a supported language. Fluid's design methodology is independent of any hardware description language. Modules designed in any hardware description language can be used as long as the channel interface is implemented at the module's boundary.

4.4.5 Channel Software Interface Specification

The Fluid communication abstraction, while intended for FPGA accelerators, is not limited to FPGA modules. As an abstraction channels are applicable to components in both hardware and software. Providing a uniform interface for software and hardware services is essential to support the Fluid's abstractions. The channel abstraction is central to the communication as a service abstraction—services communicate via channels regardless of their implementation. The channel abstraction hides the details of not only communication but the implementation of the service requested through messages on the channel. Fluid defines the following channel interface for software services:

```
1  class rx_channel {
2          rx_channel;
3          bool notEmpty();
4          active_msg rx_msg();
5          void pop();
6  };
7
8  class tx_channel {
9          tx_channel();
10         bool notFull();
11         void tx_msg(active_msg m);
12 };
```

Clients are completely agnostic to a service's implementation whether it be in hardware or software. A software service thread implements a service's functionality and participates in the abstraction through channels. These channels maintain the same abstraction hardware modules see, and similarly have multiple implementations depending on the FPGA platform and processor's integration. The client module's logic does not change even if a service's functionality is implemented by a software thread on the host processor. Software emulation of services provides additional debugging and development flexibility. Designers can test their hardware services in the real system, substituting software services as necessary to incrementally develop their accelerator and service designs.

4.5 Active Messages

The Fluid design methodology relies on active messages as its primary communication abstraction. Message passing reduces engineering efforts in distributed systems as data is passed explicitly between distributed processors through messages rather than by implicit sharing data in memory. Messaging frameworks avoid errors as data is shared by communicating avoiding the problems of shared memory. The active message abstraction is a slight departure from standard message passing where active messages explicitly pass control as well as data. Active messages explicitly provide processing instructions in addition to the implicit control that is expressed as a message is received.

Active messages are communicated synchronously between modules across channel interfaces. An active message comprises a defined number of argument words and a data field. The number of arguments is fixed but their size is not. This provides structure with some room for flexibility. The number of arguments, currently tied to four, was determined through design studies in building applications. While this number is currently constrained there is ongoing work to abstract the message format beyond any physical constraints. Fluid defines the syntax of active messages as follows:

```
1  typedef struct {
2    t_arg arg0,
3    t_arg arg1,
4    t_arg arg2,
5    t_arg arg3
6  } t_head;
7
8  typedef struct {
9    t_head head,
10   t_data data
11 } t_msg;
```

Fluid's messages include both arguments and data, where arguments can be used to pass instructions to and/or aggregate data acquired by the services involved in completing the request. Passing control through memory requests—as is the case in many distributed systems–enables the elastic, composable model for high-level memory services. Arguments pass through the lowest-level memory services and pass from request to response in the message's header. Carrying metadata is a convenient mechanism to transfer data and processing instructions to services, however it is not a requirement for all channels. The metadata may be optimized away by the designer or synthesis tools. For example, a directly connected

channel streaming data between from one module to another would optimize away the metadata and even the entire response side of the channel.

The active message format separates data and arguments to communicate instructions to the service that is responsible for processing the message and/or supplying data. The service module designer defines the semantics of a message to their service through its supporting documentation. Both requests and responses use the same unified message format; however request and response messages are semantically different. Both the service and client agree upon the message definition as a de facto contract. The contract determines which arguments control the service's operation, which arguments are modified to satisfy the request, and which arguments are passed back to the client in the response message.

The active message communication model tolerates the dynamically-variable latency of both communication and service operations intrinsically. Each message requests an action to be completed at some time in the future and perhaps qualified with a response. Message passing models decouple communicating components which is advantageous to describe and reason about highly decoupled FPGA systems. Furthermore, services providing high-level memory operations are semantically different than familiar load-store memory operations. A service has it's own processing time, and may require other services with their own processing times. Message passing loosely couples each component where a client is not guaranteed a response in any particular time interval. This is a natural fit for the Fluid design methodology where services act on-the-side satisfying the request as client processing kernels continue their computations.

4.5.1 Message Driven Memory Operations

Active messages introduce an important concept in Fluid's design methodology, message driven memory operations. Each active message that passes though a service affects some operation. Read and write services are the fundamental services at the memory interface boundary. They buffer message arguments and return them with the response—an essential implementation detail that supports the active message model and service composability. Designers rely on this pattern to pass metadata and control through memory requests. This is a much richer memory model where even at the most basic level, memory is more than a read or write. The transaction communicates control, beyond accessing data. Responses trigger downstream actions through arguments. This philosophy simplifies communication logic across modules. Clients and services can communicate through memory requests rather than through separate direct interfaces. Each response, by way of the arguments and supplied data, provide the relevant data to computation kernels. The bifurcated data and metadata in each message allows services to generate

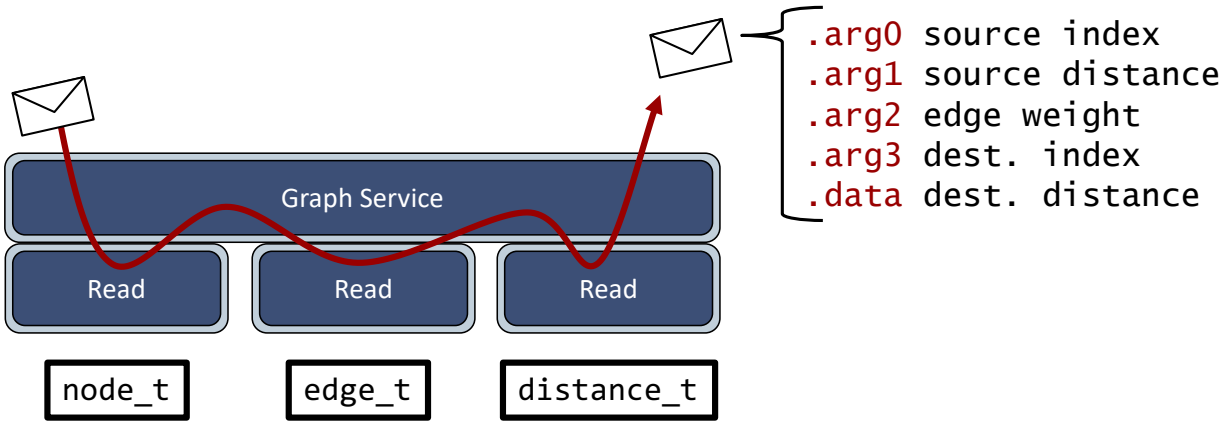


Figure 4.4: Graph service, demonstrating message driven operations, that abstracts the logical complexity to aggregate data from a sparse data structure into a relevant response.

rich responses packed with information. Services process and manipulate data from multiple memory references into dense responses that simplify the client's logic.

Memory services act on-the-side and allow user-level kernels to focus on computation. Active messages support the message driven memory operations model to pass control and metadata through memory operations. Active message arguments are used to pass data across multiple memory accesses. Responses pack relevant data passed through multiple memory requests as shown in Figure 4.4. The graph service receives an active message from a client requesting a source node's neighbors. In this example the graph is represented with a sparse representation and separate parts of the data structure are accessed with their own read service. The graph service satisfies a request with multiple dependent memory requests. Through each request the service extracts the relevant data and passes it along though arguments to the next request. The service builds a response message with all of the relevant data from the graph data structure. Alternatively, the designer could track and build responses within their service module. That approach requires additional logic within the service module to buffer metadata and build the response. It is much simpler to pass metadata through arguments building responses on the move. In fact, the graph service generates multiple requests for each neighbor following the edges from the source node. A single request message generates multiple memory requests and, eventually, multiple responses to the client. Services absorb this type of complexity which would typically be present in the accelerator's user-logic.

4.6 Supported Connection Types

Each channel connection is eventually physically elaborated by Fluid's design framework. The framework supports and provides IPs for a variety of connection types from wires to network on chips on the FPGA fabric, and Ethernet to in-memory circular buffers for connections between FPGA devices. The designer specifies the connection type but relies on the framework to generate the necessary logic to implement the connection. Fluid currently provides the following set of connection types to the designer:

4.6.1 Wires and FIFOs

Wires are the simplest method for connecting channels. Clients and services each hold one end of the channel interface in their RTL module. As a direct connection, the design framework joins channel interfaces with wires and no additional buffering. This connection type is the simplest and allows for the tightest client-service coupling. A designer may specify additional message buffering through FIFO connections. Additional buffering increases resource utilization but is often necessary for bursty data transmission and pipeline buffering.

4.6.2 Network on Chip

A network on chip is necessary for multiple client modules to target and share a single service module. The simplest network available is an arbitrated crossbar. The crossbar generates multiple virtually private service channel interfaces that can be connected to client channels. The crossbar arbitrates messages in a round-robin protocol and returns responses to the requester. The arbitrated crossbar is useful in cases where only a few clients share a service. Fluid's design framework also supports a fully parameterizable NoC. The NoC is generated at compile time for the design's required use case. The framework provides the interface to describe the network at a high level for example its topology and data width. The network is then generated according to the specification and required number of endpoints to implement the channels. The NoC increases scalability and flexibility for communication channels. Multiple channels can share the same NoC transparently without impacting the functionality of their design. Services of multiple types and different channel widths are able to share the same NoC. Fluid design framework also introduces the appropriate logic to adapt channels to the NoC's physical interface structure.

4.6.3 Ethernet Network

The Fluid communication abstraction is not limited to designs on a single FPGA device. Currently, FPGAs are found in data center settings and feature high-bandwidth network interfaces to communicate data

over Ethernet networks. The Fluid communication framework supports connections across an Ethernet network through the Intel Inter-Kernel Links (IKL) protocol and architecture [8]. IKL is a streaming protocol supporting virtually direct connections for hardware modules across multiple FPGA devices. Support for IKL is similar to the NoC discussed above, the logic required to adapt channels to the IKL interface is provided in Fluid's infrastructure library. This includes the logic to create flits from active messages as the IKL interface is a fixed bit width.

4.6.4 In-Memory Circular Buffer

The circular buffer connection type is essential to extend communication across FPGA devices in shared-host multi-FPGA systems where no other direct connection exists. Some compute systems feature multiple FPGA devices in a single server with access to host memory. This more traditional system design integrates FPGA devices with the host on the PCIe bus. FPGAs in these systems have limited access between devices to transfer data directly. Fluid provides an channel connection implemented as a circular buffer located in the host's main memory. Both FPGA cards must have access to the same host's memory to use this connection type. The circular buffer supports large data transfers through the PCIe bus to/from host memory. As each channel is bidirectional, two circular buffers are needed for each channel. The size of each buffer in host memory is parameterizable through the host software main control thread.

4.6.5 Software Channels

The channel abstraction for software threads has multiple implementations just as hardware channels do. Software threads are agnostic to the channel mapping as the software channel interface abstracts the implementation details. Current support for software channels and services relies on the Intel Open Programmable Acceleration Engine (OPAE) SDK [46], however this SDK provides multiple methods to transfer data from/to software to/from FPGA hardware.

Memory Mapped Input-Output

FPGA devices with embedded and host processors provide memory mapped access to registers on the FPGA fabric. Embedded CPUs access these registers through bus masters between the processing core and programmable logic fabric. Host processors for dedicated FPGA cards similarly support memory mapped access over the shared PCIe bus. These communication mechanisms are typically used to control the FPGA logic, provide run-time configuration data, and poll status on the FPGA fabric. However, memory mapped input-output (MMIO) access can be manipulated to transfer data. MMIO interfaces are

limited in performance for transferring bulk data as this is not the intended use case. Nonetheless, our evaluation shows this is a viable option for limited data transfers between host processors and FPGAs. The Fluid framework provides a software class and hardware logic to implement channel connections through MMIO transfers on the Intel FPGA cards with OPAE.

In-Memory Circular Buffer

An in-memory circular buffer is the second option for channel connections between hardware and software. The software-hardware circular buffer channel relies on the same principles as the previously described hardware-only circular buffer channel. The circular buffer resides in host memory and the logic to send and receive messages from this channel is provided by the Fluid framework. A software class to send and receive messages through in-memory circular buffers with the software channel interface is also provided.

Chapter 5

Composing Service-Oriented Designs

Fluid's design methodology supports reusable modular design practices and enables designers to build abstraction layers in reconfigurable hardware. The previous chapters described Fluid's service abstractions for memory and communication that decouple their implementations from the accelerator designer's modules. In this chapter Fluid's design framework is presented that supports a service-oriented paradigm for hardware design composition. This chapter begins with an example of a service-oriented application design. The example motivates the service-oriented design style that absorbs complexity from user-level kernel modules. This chapter then describes the Fluid accelerator design framework in detail. It discusses the details of the Fluid design framework, the syntax to describe client and service modules for use in the Fluid design framework, the syntax to describe a system design and the Verilog files generated implementing the specified design. A discussion of the high level design composition abstraction concludes the chapter. The discussion details the design framework's significance in encapsulating infrastructure descriptions so that they are adaptable and reusable as intellectual property.

5.1 Example Service-Oriented Application Design

This section describes a simple application with kernel modules distributed on the FPGA fabric that desire to increment variables in memory. While the example is trivial in nature, it describes a realistic scenario where the memory as a service paradigm fits. Furthermore, the application provides a reference while discussing the features and mechanisms of Fluid's design composition framework for building service-oriented FPGA accelerators.

Consider an example where there are many distributed processing kernels located on an FPGA fabric. From time to time, these kernels need to increment the entries of a shared table of counter variables in off-chip DRAM. Following today's common approach (Figure 5.1), each processing kernel is required to

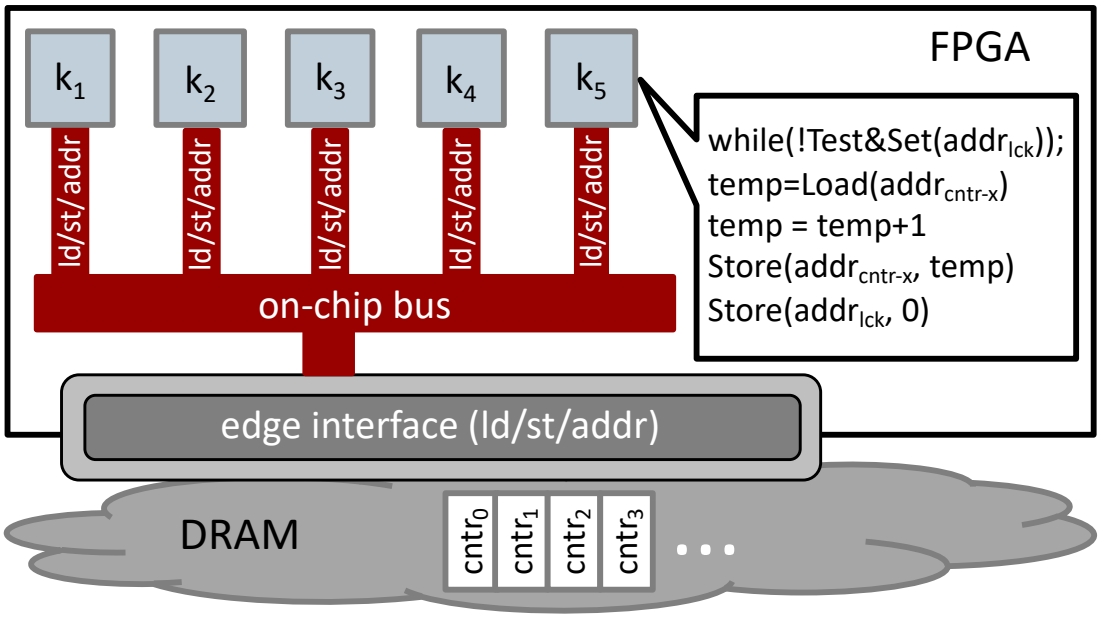


Figure 5.1: Processing kernels incrementing a shared table of counters in DRAM. Each kernel must be aware of the interface and locking semantics, as well as the table data structure details.

understand the table explicitly as an array in DRAM and manipulate it through load and store primitives via a middleware layer like the AXI or Avalon bus[1] [43,96]. In addition to following a correct locking discipline, for each increment operation, a processing kernel needs to issue a load to bring the table entry from DRAM and then to issue a store to write the updated value back to DRAM. While straightforward, this is not the best way to think about hardware design given FPGAs' inherent customizability and hardware concurrency.

Consider an alternative scenario (Figure 5.2) where a centralized service module acts as a proxy to provide the abstraction and implementation of the counter variable table. The processing-focused kernels only need to send simple, fire-and-forget increment requests to the service module to effect the table increments. The processing kernels, unaware of the table's implementation, use the logical indices of the table entries as arguments to the requests. Only the service module needs to understand the table layout in memory and DRAM specifics. For example, the service hides the complexity and hardware implementation of the logic to update a single word when the granularity of DRAM accesses is larger. As a centralized agent, atomicity of the increment operation is trivially observed by the service module without the need for locking or cross-kernel communication. The performance is improved because the read-increment-write sequence is contained within the service module and can be placed near to the

[1]These busses are implemented on the FPGA fabric in soft logic. Like other modern busses, AXI and Avalon busses are in reality switched interconnects that connect bus initiators with address-mapped bus targets using split-phase address and data transactions.

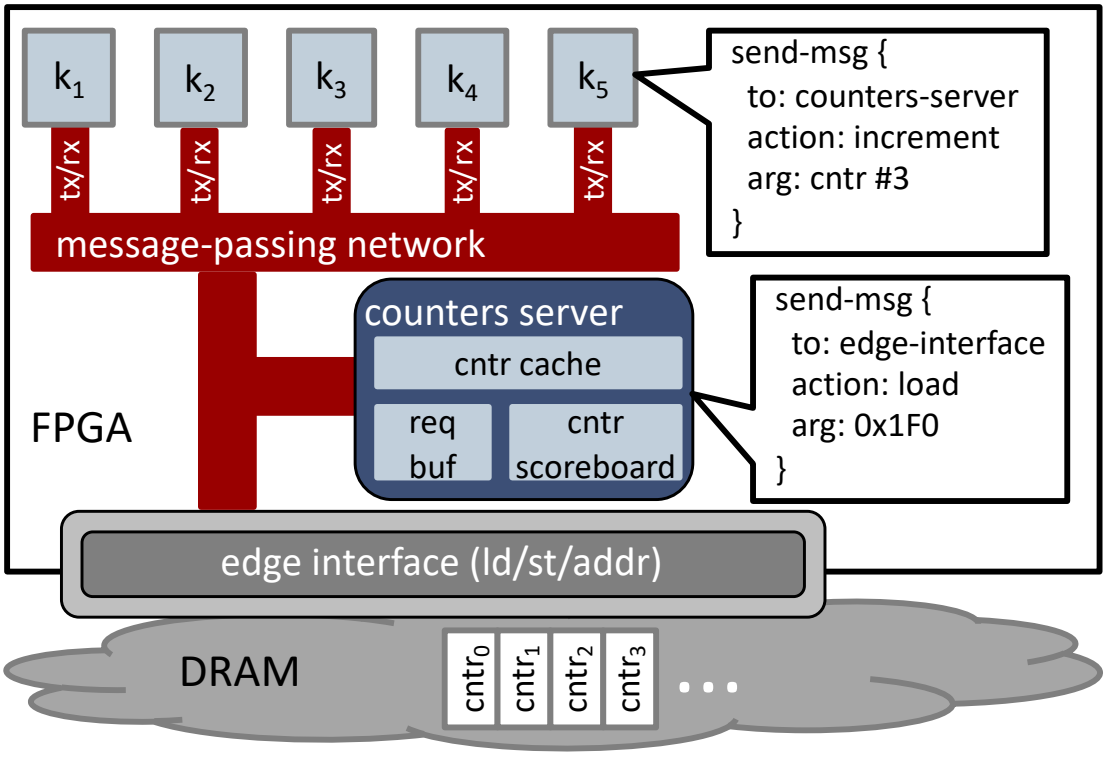


Figure 5.2: A service module acts on the kernel's behalf to manipulate counters in memory in response to a message request.

physical DRAM interface. The service module could even introduce caching and merging optimizations without affecting the design and operation of the processing kernels.

Fluid's service-oriented model enables the programmability and support, that software designers have come to expect through abstraction, with soft-logic modules on the FPGA fabric. In the second scenario, the counters service module provides a service-level abstraction of a memory-resident table of counters to the processing kernels. The service's abstracted operation is implemented in soft-logic on the FPGA fabric, in the same manner as the processing kernels. While ultimately the table is stored in DRAM memory locations and accessed with loads and stores at the FPGA fabric's edge, there is no reason for each and every in-fabric processing-focused kernel to be designed to operate against such an explicit and generic view of memory. Kernels and their designers should expect and come to rely on high-level semantic-rich memory operations, provided as services, within the FPGA fabric.

5.2 Fluid Design Composition Framework Overview

This section presents an overview of Fluid's design composition framework that provides a programming interface and RTL generator tool to build service-oriented accelerator designs. Fluid's programming in-

terface allows designers to specify their applications at a high level in Python, instantiating their modules, services, and logical connections rather than writing RTL. This design framework enables designers to compose service modules correctly, build systems of services, and integrate their user-level kernel modules into an overall design.

The Fluid programming environment's primary objective is to provide a means for hardware developers to build accelerators from a service-level specification. Accelerator designers working within Fluid's scope are responsible for designing their kernel modules, selecting services from the service catalog, writing a Fluid accelerator composition file, running the system generation tool, and then their vendor specific FPGA tools. A Fluid accelerator is described at a high level in Python. The designer's Python files are provided as an input to the design framework's tool that instantiates and connects both kernel and service modules.

Fluid's programming interface increases productivity and convenience for designers to build systems and experiment with their designs. The first level of support virtualizes platform details, specifically external memory devices. This virtualization layer enables higher-level service modules that are generally reusable and made available to designers in an extensible library. Providing a common base layer for all memory services increases portability and allows the designer to target the memory device appropriate to their use case, without impacting functionality. The lowest level memory services virtualize standard AXI and Avalon bus interfaces used to access physical memory devices exposed at the top level module interface. Fluid's design tool can be used with FPGAs from Intel and Xilinx, however it provides special support for Intel FPGA platforms. The generator provides this additional support for Intel programmable acceleration card (PAC) platforms with the OPAE shell and software development kit. This allows designers to quickly build FPGA designs on the Intel PACs with host communication and control. Fluid's programming interface also allows the designer to export arbitrary ports from any module to the top level module's interface. This enables an "escape hatch" from the abstraction to support arbitrary external interfaces as well as control and status registers.

5.2.1 Design Framework Details

An accelerator designer is not restricted to any language in developing their kernel modules. However, designers must adhere to `channel` interfaces in their modules to request and consume services, as discussed in Chapter 4. The designer's modules request a service abstractly through messages sent via the appropriate channel, unaware of the service's physical implementation. Each channel is tied to a particular service that is instantiated in the accelerator composition file and later elaborated by the design

generator. The designer also specifies channel connections, between their modules and service modules, and their implementation details in their high level design file.

As a part of the service abstraction and design methodology, each service may require a number of other services to build its functionality. Therefore, services rely on the same abstraction provided to kernel modules through channels. Both service and accelerator developers are only aware of operations ascribed to a channel, Fluid's generator tool is responsible for elaborating connections and instantiating the modules required within the design. This dependency is exposed through `provides` and `requires` keywords. A service module provides a service on one of their channel interfaces. One service is tied to each interface, however a service module may provide more than one service. Any module, including service modules, may require a service on a channel interface. Again, one service is tied to each interface, but a module may require multiple services on separate interfaces. This architectural concept defines a type for each channel and allows the generator to connect interfaces correctly. This is an important distinction from other hardware system design tools. Current design integration tools are only concerned with the physical correctness of a connection, verifying the bus width, control signals, and inserting shims to correct for disparities. Fluid's framework enables semantic checking to verify the correct operations are linked by a channel connection.

Fluid's design framework relies on two major design components that allow designers to conveniently specify and generate systems from service-level descriptions. The required design components are the catalog and registry files. These files are described here as separate entities, but the catalog can be enclosed in the registry file. The catalog lists the services available to a designer and the registry describes the overall service-oriented design. The catalog supports the service abstraction by providing Python class descriptions of hardware service modules that designers can instantiate at a high level and access within their hardware designs. Each class describes a service module, its channel interfaces, and the services ascribed to each channel. A designer expresses their accelerator design at a high-level in the registry file. The designer instantiates service and client modules from classes found in the catalog and logically draws connections between them. Each channel connection is also designated with a connection type that defines its implementation. The designer also specifies the external platform interfaces that the lowest-level read and write services attach to within the registry code. The final design is compiled by a series of passes that generate a top level design file in Verilog that instantiates modules, connections, and infrastructure.

5.3 Curating Services

This section details the syntax for describing a module as a Python class in the catalog file. As shown in the example from the first section, services abstract FPGA memory operations to simplify accelerator development and support high-level functionalities within the FPGA fabric. In order to realize the design, each module has a corresponding catalog entry. The designer later builds their design, at a service-level, pulling from the catalog of library components.

Fluid's design methodology enables memory systems and entire accelerators to be composed of loosely connected components, simplifying development without relinquishing flexibility. A service comprises a logical operation and its physical implementation. Services are implemented by service modules developed as soft-logic in Verilog or Bluespec [10] available in the service library. Each service must have a corresponding catalog entry so that it is accessible to accelerator designers in Fluid's design framework. The catalog defines a service as a Python class that describes its channel interfaces, operation types ascribed to each channel, parameters, additional ports, and properties. The catalog entry is a binding contract between accelerator designers and the service developer. The listing below sketches the abbreviated entry for the "counters" service:

```
1 class Counters(Service):
2   def __init__(self, name):
3     super().__init__("mkCounters",
4                      SourceType.BLUESPEC_SYSTEM_VERILOG, name)
5     self.add_provides("count", 32, 32, channel_type="IncCounter")
6     self.add_requires("table_read",  channel_type="Read")
7     self.add_requires("table_write", channel_type="Write")
```

Service designers specify the RTL module that implements the service, and source type in the **__init__** method. Each service module included in the catalog has a corresponding RTL module in the service library directory. The generator tool pulls the module with the matching catalog provided name from the library. Services implemented in Bluespec are handled slightly differently and the generator builds a wrapper for the modules built from Bluespec source. Currently support is limited to Verilog and Bluespec services, however designers can choose to build services in any language that synthesizes to RTL. This could include HLS tools, but that is outside of the scope for this thesis and left to future work. The `add_provides` method is used to designate the service that the module provides and the properties of the channel. The first two properties are the argument width, and the data width respectively. The framework uses the `channel_type` to verify correctness when clients connect to this channel.

Service composition extends service module designers the same ease-of-development support that user-level designers are afforded through Fluid's service-oriented design methodology. Not only can services make use of abstract portable operations through other services, this freedom to build complex service hierarchies enables rich customization without redevelopment cost or knowledge of the outside module's implementation complexity. Returning to our previous example, the counters service requires read write services. This is specified in the counters service catalog entry in the `add_requires` methods. The read and write services provide virtualized access to the the table of counters in memory. The counters service does not need to be aware of the location of the table in memory nor the hardware details of the physical memory device.

Designers also have the option to specify additional RTL ports, outside of the scope of the channel interface, through `add_control` and `add_extern_connection` methods. External wires and registers are exported to the top level module's interface. This allows designers to specify arbitrary connections and interfaces from their modules to components outside of the Fluid generated system. Controls are special purpose registers for control and status. The generator tool elaborates control ports, and optionally connects them to a module that supports indexed access to each control and status register via a bundled port interface. This feature is provided as a convenience for designers to configure and monitor modules from host processors. The generator also supports a Verilog template language in Python where designers can generate the configuration and status register module that bundles these registers.

Modular design promotes reusability and is a best practice in RTL design, service implementations may include RTL modules within their internal hierarchy as additional sources. Fluid's design framework helps track and manage these files as well. In a service's class definition designers can specify additional modules to include. The `add_property("additional-files", ["file_name.sv", ...])` method is provided where the designer lists all additional design files used within their service module. This allows the generator to copy these files to the final generated design directory. Designers benefit from the additional support to reduce time spent tracking down included modules and design files.

5.4 Instantiating Services in a Design

Normally, a Verilog module instantiates another module as an internal object; an instantiated module's input and output ports are wired up by the enclosing instantiator module. A service module is not instantiated in the traditional sense, they are accessed logically outside of the instantiator module's internal hierarchy. Designers express abstract connections to services, through channel interfaces, in their clients but not the service's physical implementation. This service-oriented development model enables

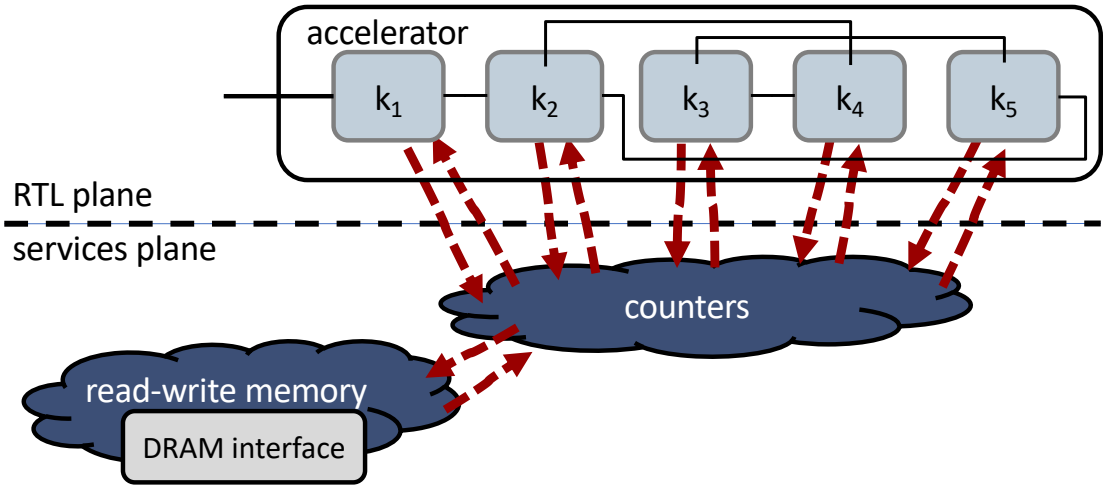


Figure 5.3: A service-level sketch of the design from Fig. 5.2 showing the normal accelerator design module hierarchy, abstract services and channel connections.

convenient and flexible system design that can be expressed at a high level.

In Fluid service-oriented designs, service modules live "extracorporeally"—outside of a typical user-level compute accelerator's module hierarchy. Any client module indicates the desire to access a service module by declaring a **channel** interface in its port list (e.g., "cntrs" below). This interface is left dangling only to be completed later during post-processing rewriting in the generator. In the example each kernel module, as a client, has a channel interface that requires the counters service. The RTL kernel module is agnostic to the implementation of the service and channel connection.

```
1  module kernel(input clk, channel cntrs);
2    ...
3    always_ff @(posedge clk) begin
4      cntrs.txPush <= do_inc && !cntrs.txFull;
5      cntrs.txMsg.arg0 <= cntr_id;
6      cntrs.txMsg.arg1 <= how_much;
7      cntrs.txMsg.arg2 <= no_ack;
8    end
9    ...
10 endmodule
```

Client and service module instantiations are expressed by the designer in the registry file separate from the Verilog source code. The registry file specifies the logical organization modules and their connections. Figure 5.3 shows a logical sketch of the counters accelerator design from Figure 5.2. The design, in effect,

exists in two planes: the RTL plane and the services plane. In the example design, five kernel modules exist in the RTL plane accelerator. The kernel modules are only aware of the counters service through their channel interfaces. Connections are shown as red arrows to an abstract counters service. These connections are expressed by the designer only in the registry file. All services, shown in the service plane, are instantiated in the registry file. The services plane may contain many more services than the accelerator is aware of in order to support the services required by client modules in the RTL plane. In this example, the counters service requires a read-write memory service to access external memory. The listing below sketches one kernel, the counters service, and read/write services captured in the registry file for the counters accelerator:

```
1  from header import *
2  from visualizer import *
3  from opaeplatform import *
4  from catalog import Kernel, Counters, ReadWrite
5
6  kernel = Kernel("kernel_0")
7  counters = Counters("counter_svc_module")
8  mem = ReadWrite(platform.ccip, 64, 64, "mem")
9
10 app.noc(kernel.requires["cntrs"], counters.provides["count"])
11 app.direct(counters.requires["table_read"],  mem.provides["read"])
12 app.direct(counters.requires["table_write"], mem.provides["write"])
```

The designer first imports the standard support packages for the framework, visualizer, and OPAE FPGA platform. Next the client and service modules are imported from the catalog. The designer can now instantiate these modules as shown in lines 6–8. Next, a connection is made to link the channel interfaces between the counters service and kernel module shown above as a NoC connection. Similarly the connections between the counters service and the read/write services are specified with a direct wired implementation. Connections must respect the provides/requires relationship, the framework verifies connections for correctness in construction and operation.

5.5 Platform Setup and Design Generation

Once the registry file has been written by the designer that specifies their desired service-oriented system, the file is passed to the generator. The generator then: 1) parses the registry and catalog to construct a

memory system module with appropriate external interfaces, 2) instantiates the modules in the design, and 3) implements physical connections from their logical specification. The design generator is responsible for elaborating the top level design and verifying channel connections. The generator also elaborates the top level interfaces, clocks, resets, external ports, and control registers. For example, this includes the clock, reset, CCIP interface to host memory, local memory Avalon interfaces, and memory mapped input output registers for OPAE supported designs. The platform setup specification is shown below:

```
registers = OPAERegisters("soma_csr", SourceType.VTL, __file__)
platform = IntelOPAEPlatform(registers)
app = Application(platform, path_prefix="examples/counters/")

... modules and connections ...

pass_manager = PassManager(app)
pass_manager.add_pass(OPAEAnalysisPass)
pass_manager.add_pass(OPAETopPass)
pass_manager.add_pass(OPAEBSVWrapperPass)
pass_manager.add_pass(OPAENOCWrapperPass)
pass_manager.add_pass(OPAEFinalPass)
pass_manager.add_pass(
              generate_visualizer_pass(VisualizerType.DEFAULT))
pass_manager.schedule_and_run_passes()
```

The platform as well as the control and status registers are instantiated with python variables with similar syntax to client and service modules. The Application class instantiates the overall application. The designer attaches a `PassManager` that provides a handle to the compilation flow. The flow requires the: Analysis Pass, Top Pass, and Final Pass; which are triggered by a `schedule_and_run_passes` call. The BSV wrapper pass is required for service modules generated from Bluespec and generates a SystemVerilog wrapper with channel interfaces rather than individual ports. The Visualizer pass generates a top-level block diagram of the system design.

5.6 Design Visualizer

Block diagrams have a lot of legacy in hardware design to conveniently and efficiently integrate IPs into an overall system design. A convenient design visualizer is provided in the Fluid design framework. This

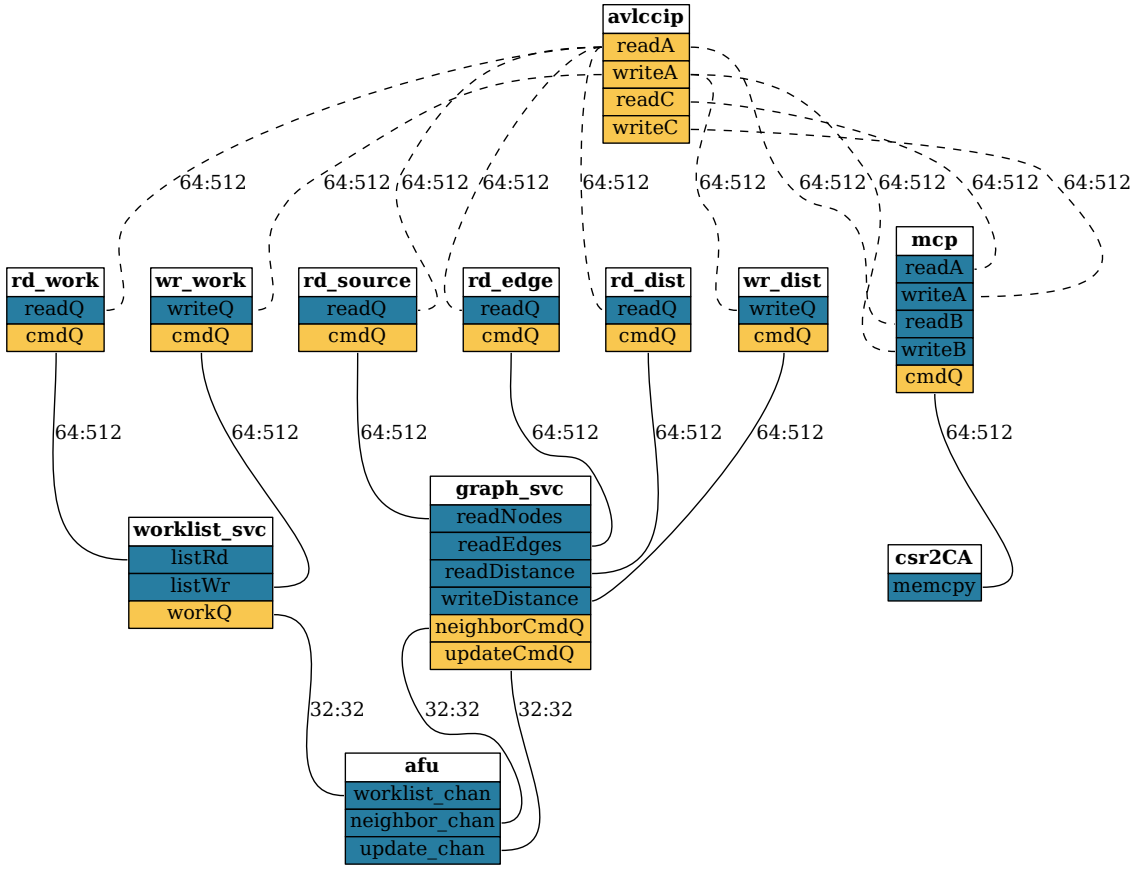


Figure 5.4: A service-level sketch of an accelerator design generated by the design framework's system visualizer.

allows designers to generate service-level sketches of their systems as a high-level overview and to verify their components, connections and connection types. An example visualization is shown in Figure 5.4. Each block represents a module in the design, either a service or client module. Channel interfaces are colored based on whether they are providing or requiring a service. Channels that provide a service are colored in teal, where channels requiring a service are colored in gold. Connections between channel interfaces are drawn with lines with labels denoting the arguments width and data width in bits. The example in Figure 5.4 shows a complete design for a graph application accelerator. The user-level kernel code is in the "afu" module which requires services from the "worklist" and "graph" services modules. These services are supported by a number of individual read and write services which connect to a shared interface at the top of the sketch. These connections are shown with dashed lines denoting that they are implemented via a network on chip.

5.7 Improving Design Description Capability

Fluid's design framework empowers designers to integrate modules and architect their accelerators through a descriptive and extensible programming interface. It owes its flexibility to Fluid's structured abstractions that allow hardware to be easily described at a high level, specifically in Python. Fluid's design methodology applies principles from service-oriented architecture to hardware design to encourage modularization for reuse and provide structure at a module's boundary with a clean channel interface. Furthermore, Fluid's design methodology allows for abundant freedom to implement infrastructure for connections between modules' channel interfaces. Simplifying and constraining module interfaces and constraining the abstraction to a module's boundary enables Fluid's high level design framework and powerful descriptive capabilities.

Fluid's design framework gives the designer transparent control over implementation details expressed at a high level in their accelerator description discussed above as the registry file. Through the framework's programming interface, designers can describe channel connection implementations without ever touching the RTL. At compile time Fluid's design tool generates the connection infrastructure from a flexible library of modules and tools to build the desired connections. Moreover, the designer can express and control the implementation details of the infrastructure that the tool generates. For example, the designer describes connections between channels with a NoC implementation. Designers also have control over the implementation details of the NoC such as the topology and width when they instantiate their application class as shown below:

```
1 app = Application(..., noc_width=224, noc_topology="double_ring")
```

Fluid's highly descriptive design interface increases productivity and enables the designer to efficiently integrate their modules and conveniently architect their accelerator at a high level rather than RTL. Raising the level of abstraction for design description is a key component of Fluid's methodology and is accomplished through a high level programming interface. The interface allows accelerator designers to describe their architectures including the infrastructure details in Python. Using a high level language like Python for the programming interface provides additional flexibility so that the interface can expose design details when necessary or hide them to simplify the design effort increasing productivity. Described above, Fluid's programming interface defines primitives to build connections between channel interfaces. It is also possible to build reusable functions that encapsulate design patterns within the high level programming interface. In fact, this creates another layer of abstraction where infrastructure can be encapsulated and reused as unit of intellectual property. Consider a set of streaming module with a common multi-channel interface. A function that connects the channels between a source and sink service

module can be defined as a reusable function as shown below:

```
def connect_channel_fifo(app, fifo_class, name, source, sink):
    fifo = fifo_class(name)
    app.direct(source.requires["out_pkt"], fifo.provides["in_pkt"])
    app.direct(source.requires["out_meta"], fifo.provides["in_meta"])
    app.direct(source.requires["out_usr"], fifo.provides["in_usr"])

    app.direct(fifo.requires["out_pkt"], sink.provides["in_pkt"])
    app.direct(fifo.requires["out_meta"], sink.provides["in_meta"])
    app.direct(fifo.requires["out_usr"], sink.provides["in_usr"])
```

The `connect_channel_fifo` function is built from primitive connection syntax to make multiple connections between modules supplied as arguments to a single function call. This is a simple example, however it demonstrates a useful function that provides a reusable design pattern. The function also inserts a specialized FIFO between the source and sink services that is similarly provided as an argument. Encapsulating infrastructure as an IP boosts productivity providing layers of abstraction in the programming interface. Furthermore, composing hardware at a high level, including reusable infrastructure, makes hardware design more accessible.

Fluid's programming interface simplifies design composition, compared to RTL, but still exposes implementation details so that designers have control over the final output. Implementation details and module parameters are exposed through through the `add_property` method. Properties can be modified in a module's class definition or more programmatically through functions that return instances of the class.

```
def make_channel_fifo(clk_i, rst_n_i, dual_clock=False,
                      clk_o=None, rst_n_o=None):
    class ChannelFIFO(Service):
        def __init__(self, name):
            super().__init__("channel_fifo",
                             SourceType.SYSTEM_VERILOG, name)
            if dual_clock == False:
                self.add_extern_connection("Clk_i", clk_i)
                self.add_extern_connection("Rst_n_i", rst_n_i)
                self.add_property("parameters",
```

```
11                                          {"DUAL_CLOCK":"0",
12                                           "MEM_TYPE":'"M20K"',
13                                           "USE_ALMOST_FULL":"1",
14                                           "FULL_LEVEL":"450",
15                                           "SYMBOLS_PER_BEAT":"64",
16                                           "BITS_PER_SYMBOL":"8",
17                                           "FIFO_DEPTH":"512"})
18              else:
19                  self.add_extern_connection("Clk_i", clk_i)
20                  self.add_extern_connection("Rst_n_i", rst_n_i)
21                  self.add_extern_connection("Clk_o", clk_o)
22                  self.add_extern_connection("Rst_n_o", rst_n_o)
23                  self.add_property("parameters",
24                                          {"DUAL_CLOCK":"1",
25                                            ... })
26              ...
27      return ChannelFIFO
```

The function above returns the a ChannelFIFO class, allowing the designer to select a single or dual clock implementation. This example function describes a choice between single and dual clock FIFO. The designer also provides the clocks as parameters. Fluid's programming interface is abundantly expressive and extensible allowing designers to expose low level design details into the high level interface. While most of the parameters are hard coded in the ChannelFIFO function above, additional arguments could be added to control other parameters for the FIFO.

Chapter 6

Building Services

This chapter details the current set of services provided in the Fluid library and catalogs their descriptions. This library of services is a proof of concept example set designed and implemented for this thesis. Services can be added to the library to augment the existing set, compose additional functionalities with current services and/or provide additional utilities. Another way the service library can grow is to include variations of the same service with different performance and area characteristics. The current set of services is focused on support for memory abstraction, high-level operations on data in memory, and processing streams of data. This catalog describes each service's message format, general functionality, and implementation details.

The term "service module" refers to the implementation of a service that will be instantiated in a design. A service is typically implemented as a soft-logic service module in the FPGA logic fabric. A service developer must provide a catalog definition of their service, as a Python class, in addition to the service module implementation. A catalog definition describes the service operation, the request message format, and response message format. Application developers use the Python class to instantiate services in their system designs as an object. Fluid's generation framework uses the channel specifications in a service's class to properly verify and elaborate connections between clients and services. A service module can be implemented in any hardware description language; Verilog, SystemVerilog and Bluespec SystemVerilog are currently supported. Additionally, some services may have a functionally equivalent software implementation. All services must adhere to the channel interface to accept requests and return responses to clients within Fluid's design methodology.

6.1 Linear Memory Services

Accessing data in a linear address space is the familiar and fundamental abstraction for most compute devices. The Fluid library provides a set of services for linear memory access that all memory services build upon. In addition to read and write access, the library provides caches and a atomic read-modify-write service. Linear memory services are the fundamental building blocks for all Fluid memory services. These services impose the channel abstraction on standard memory interfaces and support Fluid's message passing through memory paradigm.

6.1.1 Read and Write Services

Read and write services form the lowest-level foundation of Fluid's memory as a service abstraction. The Fluid service library provides read and write primitive services as the base-level abstraction layer to virtualize the FPGA device's memory interfaces. Clients access data in memory with the following message format to/from read and write services:

Message Type	Argument	Description
Request	arg0	User defined. Argument returned with response.
	arg1	Index of block data to access within the scratchpad range.
	arg2	User defined. Argument returned with response.
	arg3	User defined. Argument returned with response.
	data	Write data. Read not used.
Response	arg0	Argument value from request.
	arg1	Index value from request.
	arg2	Argument value from request.
	arg3	Argument value from request.
	data	Read data. Write not used.

Table 6.1: Read and Write Service Message Format

Each read-write service abstracts the details of the physical memory device where the data is stored. Clients no longer use load/stores and addresses, they access data with an index into a linear scratchpad. Read and write services are located at the memory device interface and support Avalon, AXI, and CCIP

interfaces. The service abstraction provides a portability layer for clients across memory devices with these supported interfaces. For example, an accelerator initially accessing data in a local memory device with an AXI or Avalon interface can seamlessly move to host memory with the CCIP interface. The Read and Write services abstract the structural and semantic differences of the different bus interfaces, and the accelerator logic does not need to change moving from one memory device to another. The service library also provides a read/write service for on-chip BRAM memory.

In addition to abstracting on and off chip memory devices, read and write services virtualize memories as private scratchpads. Multiple read and write services can even share access to the same memory device. If a designer specifies multiple independent read/write service modules, each service appears to have private access to a region in memory. This region is allocated statically by the designer. The global base address and range are set through configuration registers for each read/write service. Separating memory regions is up to the designer's discretion, for example regions can separate data structures to parallelize independent requests.

The Fluid infrastructure library provides a crossbar so that multiple read and write services can share the same physical memory device. The interface crossbar is a core infrastructural component hidden beneath the read/write service abstraction. The crossbar's scheduling algorithm is selectable supporting both round-robin and priority scheduling. In addition to sharing memory devices, the infrastructure crossbar can interleave memory requests across multiple memory devices. The crossbar interleaves requests by address across the configured number of memory devices.

In addition to memory device selection, multiplexing, and address interleaving available through the interface crossbar, read/write service modules are highly configurable following "smart IP" principals. The block diagram for a read/write service is shown in Figure 6.1. Read/write service modules either provide ordered or unordered responses. Ordered read/write services employ a reorder buffer to track responses for unordered memory interfaces like CCIP. Each read/write service allows a configurable number of outstanding requests, in addition to response ordering. This parameter influences the area consumed by the service module as it must buffer request arguments to return with responses. Limiting the number of outstanding requests also provides a control knob to limit the bandwidth allocated to each read/write service. Both ordering and outstanding requests are compile time parameters configured when instantiating read/write service modules.

Read/write service modules with run-time instrumentation and introspection features are available in the library for debugging purposes. These services always return responses in order but do support the outstanding request limit parameter. The instrumented read/write services track the queue depth for outstanding requests, the access delay for memory requests in cycles, and the inter-arrival delay for read-

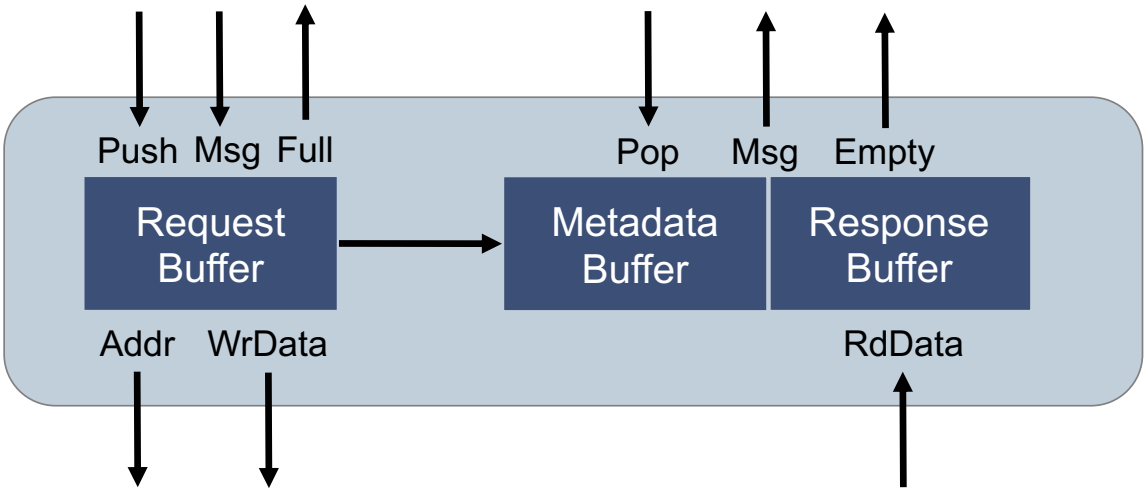


Figure 6.1: A generalized read/write service microarchitecture diagram, that supports the memory service abstraction adapting edge memory interfaces to service channel interfaces.

/write requests to the service module. These profiling features are fully synthesizable allowing designers to instrument their designs at run-time on the FPGA.

6.1.2 Memory Coherency and Consistency

Read and write services support private scratchpad regions in memory. The current set of read/write services are shared by clients multiplexing their interface at a structural level. Similarly, read/write services share a physical memory device by multiplexing its interface. Clients are responsible for managing consistency guarantees. This is not inherent to Fluid's memory abstraction, however, it is a limitation of the current implementation of the linear memory services. Future read/write services could provide support for consistency commands and coherency across multiple services to increase scalability.

Messages are an essential primitive in Fluid's memory and communication abstraction. Designers primarily share data by communicating through messages. This can take multiple forms: responses manage consistency to share data in memory and/or modules share data encapsulated in message arguments. It is possible to share data in memory, however designers are encouraged to communicate between modules through message passing rather than through memory. Fluid implements a relaxed consistency model by default. This consistency model is impacted by the model of the memory devices beneath the abstraction. For example, requests to read/write services encapsulating data in memory with a CCIP interface are unordered, a response signals to the requestor that the data is observable in memory. By contrast, requests to read/write services on AXI and Avalon interfaces are ordered, waiting for a response is recommended not necessary if the data is stored in a single device.

Fundamentally, read/write services provide access to encapsulated data in private scratchpads. Clients, whether that be an accelerator or hierarchically composed service, are responsible for managing sharing by communicating through messages. Beyond the default consistency model, services can implement additional alternative models themselves. For example, the graph service implements a sequentially consistent consistency model. All requests for node data are ordered with respect to writes within the same interface granularity block, and requests to update node data are atomic operations. This is convenient to encapsulate the ordering logic within the higher-level graph service module, however it effects a scalability limitation with a single ordering point. Serializing the graph update operation stalls requests within the service module impacting the performance of the overall design. Future services should consider consistency and ordering at a finer granularity. Additional opportunities exist to explore consistency across abstraction layers as sharing may exist at multiple levels. Data could be shared in the read/write services and higher level services may be shared separately.

6.1.3 Cache Service

A cache service is available as a performance modification to read/write services in the memory service library. The cache is available as a stand-alone service or integrated into a combined read-write service module. The available cache implementations are parameterizable in the number of requests outstanding and the cache's size. Cache services conform to the same message format and operation as the standard read/write service. The cache service is a private cache for a tightly-coupled read-write service pair. The current cache module does not provide support for coherency across multiple caches or read/write service scratchpads. The currently available cache service modules are direct mapped, write through designs. This design choice was made to simplify the cache implementation and simplify data sharing by not requiring a cache flush operation. While the provided cache design is simple, a cache makes a substantial difference when appropriately added to a Fluid memory service system.

6.1.4 Read-Modify-Write Service

The read-modify-write (RMW) service allows clients to atomically modify sub-block granularity items accessed via a read-write service. This operation mimics an atomic RMW instruction available on general purpose processors. Memory interfaces on FPGAs are inherently unconstrained so as to not limit performance and allow the designer the freedom to optimize their accelerators. However, this freedom is inconvenient when commonly used, convenient memory operations like atomics are unavailable. The message format for the RMW service is detailed below:

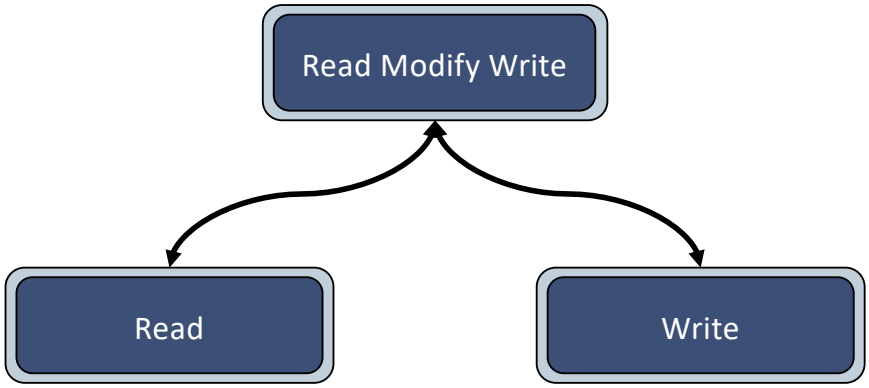


Figure 6.2: The channel connection relationship for a read-modify-write service requiring read and write services on separate channel interfaces.

Message Type	Argument	Description
Request	arg0	Not used.
	arg1	Index of sub-block item to modify within the scratchpad range.
	arg2	Not used.
	arg3	Not used.
	data	Value to write to sub-block index.
Response	arg0	Index of sub-block item.
	arg1	Block index.
	arg2	Value written to sub-block index.
	arg3	Not used.
	data	Not used.

Table 6.2: Read-Modify-Write Service Message Format

Clients request updates to sub-block granularity items and the RMW service affects these updates on their behalf. The RMW service is composed on separate read and write services that share the same address space. The RMW service module provides its service on one channel, requires a read service on one channel and requires a write service on another. The provides-requires relationship between the RMW service module and requisite read/write service modules is shown in Figure 6.2. The library

implementation of the RMW service supports multiple in-flight requests and tracks dependencies for requests to the same block. The RMW service module completes requests in-order, however it will coalesce subsequent update requests to the same block.

6.2 Counters Service

The counters service was developed as a toy example to demonstrate the capabilities of the Fluid memory service architecture. The counters service abstracts a table of counters in memory. Clients send request messages to increment these counter variables by a desired value. The client's requests can either be sent in a fire-and-forget manner or with an acknowledgement response. The counters service completely abstracts the implementation details of the table of counters and the typical load/store/address semantics of memory. The counters service module is similar in operation and structure to the read-modify-write service; each increment operation happens atomically. The counters service requires a pair of read and write services that share the same address space for the counter variables in memory. A client's message format to the counters service is:

Message Type	Argument	Description
Request	arg0	Index of counter in table.
	arg1	Value to increment counter by.
	arg2	Boolean: high for acknowledgement response message.
	arg3	Not used.
	data	Not used.
Response	arg0	Index of counter in table.
	arg1	Value to increment counter by.
	arg2	Boolean: high for acknowledgement response message.
	arg3	Not used.
	data	Not used.

Table 6.3: Counters Service Message Format

6.3 Memcopy Service

The memcopy service is inspired by the similarly named C library routine. The memcopy service allows clients to transfer contiguous blocks of data from one memory location to another. This service is generally useful for block copy transfers between memory devices. For example, transferring data from off-chip to on-chip memory or from host to local memory. The client sends a single request to the memcopy service module to initiate the transfer with the following message format:

Message Type	Argument	Description
Request	arg0	Destination block index.
	arg1	Source block index.
	arg2	Number of blocks to copy.
	arg3	Direction of the copy channels: 0 (A->B), 1 (B->A)
	data	Not used.
Response	arg0	Destination block index.
	arg1	Source block index.
	arg2	Number of blocks to copy.
	arg3	Direction of the copy channels: 0 (A->B), 1 (B->A)
	data	Not used.

Table 6.4: Memcopy Service Message Format

The client specifies the size of the transfer, the source index of the first block, destination index, and direction of the transfer. The memcopy service requires four channels for read services A/B and write services A/B. The memcopy service completes transfers from the source index to destination index; the underlying memory devices are abstracted by read/write services therefore the memocpy service is even unaware of the device supplying and receiving the data. The memcopy service completes the transfer on the behalf of the client, leaving the client free to complete ongoing computation.

6.4 Worklist Service

The worklist service provides a virtually unlimited capacity FIFO queue. This service was developed to support applications with iterative worklist-driven algorithms. This class of applications iteratively

remove and compute on work items, often generating more work that gets added to the worklist until the worklist is exhausted. Multiple work items may be generated by a single iteration at run-time. The size of the worklist often grows exponentially and cannot be statically determined for non-trivial applications. The worklist is a simple abstraction, clients add and remove work items to a FIFO queue. This simplicity is evident in the worklist service's message format:

Message Type	Argument	Description
Request	arg0	Not used.
	arg1	Not used.
	arg2	Not used.
	arg3	Not used.
	data	Work item.
Response	arg0	Not used.
	arg1	Not used.
	arg2	Not used.
	arg3	Not used.
	data	Work item.

Table 6.5: Worklist Service Message Format

The worklist service successfully hides the hardware complexity of an elastic and dynamically sized queue from the client modules. The worklist service's message format is the simplest in the current set of services. Clients push a new work item as the message data and pop these items back in FIFO order. If the worklist is small enough, it would be sufficient for the worklist service to be a FIFO buffer on the FPGA fabric. However, as discussed, this is not feasible for non-trivial problem sizes. The worklist service provides simple abstract queue semantics, yet hides a lot of implementation complexity necessary to support the large number of work items that can be added to the worklist at run-time. The worklist service module bundles work items into interface granularity memory blocks and tracks the number of valid work items in each block. As the worklist grows these blocks are automatically spilled and filled to a circular buffer in off-chip memory. The worklist service module prefetches and parses work bundles to ensure the client is not waiting on work from memory.

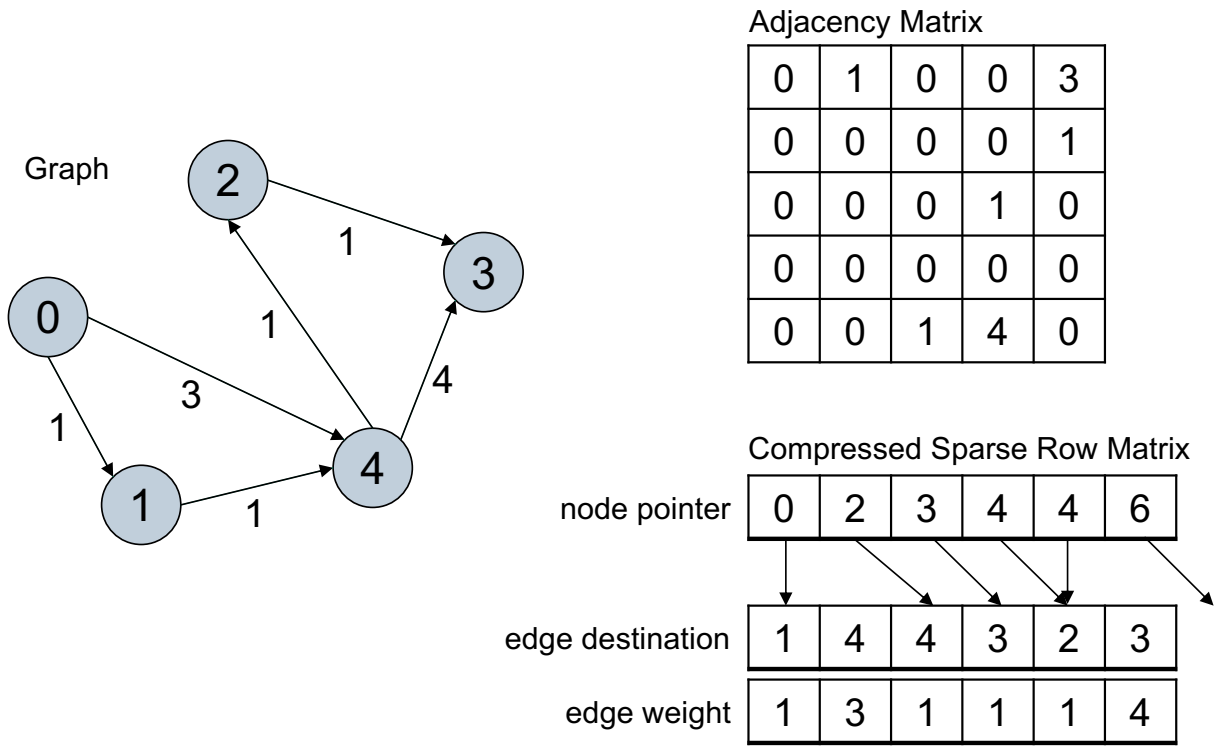


Figure 6.3: A graph (left) represented in memory as an adjacency matrix (right.top) and as a compressed sparse row matrix (right.bottom).

6.5 Graph Service

Graphs are an important data-structure that not only organizes data but codifies relationships between data items. Graphs store a data items as vertices, and relationships between data are edges. There are many ways that graphs are represented in memory, from dense representations such as an adjacency matrix or sparse representations like a compressed sparse row (CSR) matrix. A comparison of graph representations is shown in Figure 6.3. Sparse representations compress the storage required for a graph where nodes have few edges, this leaves most of an adjacency matrix filled with zeroes. This problem compounds as data sets grow, most storage space does not effect the computation, leading to sparse graph representations popularity. Sparse graph representations present their own challenges with irregular memory access patterns, as pointers are used to build the CSR data structure. The CSR format is also popular for sparse matrices in general linear algebra computations.

The graph service provides two services: “get neighbors” and “update neighbor”. The graph service abstracts the details of the graph representation from client modules. A designer implementing an accelerator for graph analysis can focus on the graph computation rather than the representation details of the graph data structure. The “get neighbors” service starts from a source node index, follows the edges

to return a set of messages one for each destination node. The "get neighbors" service streamlines the algorithm kernel by providing a compound operation that, with one request, returns not just one neighbor but a stream of neighbors and their distances (so the kernel logic does not need to iteratively look-up and query each neighbor). This encapsulation does not add extra logic cost or logic delay. The return messages pack information codified in the graph structure into the message arguments as shown in the message format:

Message Type	**Argument**	**Description**
Request	arg0	'Source' vertex index.
	arg1	Current source vertex distance.
	arg2	Boolean: 0 (Return source index), 1 (Return source distance)
	arg3	Not used.
	data	Not used.
Response	arg0	Returned source index or distance.
	arg1	Neighbor/destination node index
	arg2	Not used.
	arg3	Edge weight.
	data	Current neighbor parent index/distance value.

Table 6.6: Graph: Get Vertex Neighbors Service Message Format

The graph service also manages a dense vector that stores metadata for each node. The current metadata is returned for each destination node in the response message. The node metadata is locked when this message is returned. Clients can atomically update and/or unlock the node's metadata with the "update neighbor service". Atomic updates through this tight coupling simplifies the client module's user logic where the client interacts with the graph through entirely abstract requests. The "update neighbor" service defines the following message format:

Message Type	Argument	Description
Request	arg0	Source index or distance.
	arg1	Neighbor/destination node index.
	arg2	Boolean: 0 (Unlock node), 1 (Update and unlock node)
	arg3	Not used.
	data	Neighbor node parent index/distance value.
Response	arg0	Not used.
Only following an update.	arg1	Not used.
	arg2	Neighbor/destination node index.
	arg3	Neighbor/destination node value.
	data	Not used.

Table 6.7: Graph: Unlock and/or Update Neighbor Service Message Format

As the graph representation is completely abstracted from the designer multiple implementations of the graph service module can exist for different representations. The Fluid library currently provides a graph service module expecting a CSR matrix representation for the input graph. This graph service module requires three read and one write service. The read services access the vertex, edge, and metadata structures and the write service accesses the metadata vector. The graph service handles multiple outstanding requests for source vertices. Each "get all neighbors" request generates a lot of memory traffic accessing the CSR matrix with many outstanding requests. Responses from memory trigger downstream logic in an elastic pipeline decoupled around memory accesses. The design aims to exploit available memory-level parallelism. A scoreboard structure tracks conflicts when accessing the metadata vector and stalls the pipeline when the service is waiting for an update or unlock request. It is a best practice to specify a cache for the read/write services that access the node metadata vector to maximize on-chip data reuse.

The graph service module is also extensible to general linear algebra CSR matrices. the Fluid library provides a simplified version of the graph service module with no changes other than removing the scoreboard and locking logic. This implementation serves a sparse matrix and dense vector applicable to sparse-matrix vector multiplication. Clients supply a row index with each request for each column from

the matrix with non-zero data. Responses from the generalized graph service match column data from the matrix with the corresponding row data from the dense vector. This response is immediately useful to the client in performing a matrix-vector multiplication.

6.6 Streaming Services

Stream services transfer data in a sequential response pattern. This set of services support an important class of applications well studied and suited for hardware acceleration. Stream services change the presentation of data from its original layout in memory to a FIFO semantic stream. Stream services go beyond data transfer to/from memory, stream services modify data and perform computations as well. Stream services operate at a coarse granularity completing a request before starting on another. The stream services use a unique command tag to initiate a stream and detect whether an arriving message is relevant to the current operation. The command tag identifies the stream that the service is currently working to complete. The first request message sets this command tag, any subsequent messages must match this tag, any mismatched requests are dropped. This ensures that streams are not interleaved incorrectly in memory. Stream service modules accept a new command tag after the number of transfers specified in the first request message is complete.

Message Type	Argument	Description
Request	arg0	Command identification tag.
	arg1	Stream start index.
	arg2	Number of transfers.
	arg3	Not used.
	data	Not used.
Response	arg0	Not used.
	arg1	Not used.
	arg2	Not used.
	arg3	Not used.
	data	Steam data.

Table 6.8: Read Stream Service Message Format

Message Type	Argument	Description
Request	arg0	Command identification tag.
	arg1	Stream start index.
	arg2	Number of transfers.
	arg3	Not reserved.
	data	Stream data.

Table 6.9: Write Stream Service Message Format

6.6.1 Array Stream Services

Array stream services provide streams of data stored in a linear array in memory. The service module requests data transfers to/from the array on the behalf of the client. Both read and write stream modules continue to read and write data to the array until their transfer is complete. Array stream service modules implement data marshalling to adapt the stream data width to/from the read/write service data width. The data streams follow FIFO semantics with buffering and flow control. An array read stream service is also available with a software implementation that emulates the functionality of the hardware service module. Clients do not need to be aware that the service is implemented in software; the change does not impact functionality, just performance.

6.6.2 Linked-List Read Stream Service

The linked-list stream service adheres to the same message format as read array streams. Clients requesting the stream are unaware of the data format change as the stream service abstracts these details below the streaming service message format. A linked-list is the customary example of a data structure with an irregular memory access pattern. Each subsequent memory access is dependent on the "next" pointer stored in the each node. The performance of linked-list operations is entirely dependent on the latency to read each subsequent node. This effect is often dependent on the list's layout in memory, which is explored in the analysis of connection types in Figure 8.10. Strided linked-lists that force a row buffer miss in the memory controller represent a worst-case scenario reducing the overall throughput to stream the data from the list.

6.6.3 Stream Processing Services

This set of stream processing services abstract computations that modify data streams. Stream processing services support small computations to entire streaming applications. The stream processing service modules provided in Fluid use a common shell and follow the write stream message format from Table 6.9. Using the write stream format, these modules can be composed in a streaming pipeline. Simple stream processing services map computations to every data item as it moves through the pipeline. For example the map service applies a general operation to each element without effecting the order of size of the stream. The streaming multiply accumulate (MAC) service module receives the stream of neighbors from the graph service. The MAC service wraps a floating point multiplication unit and returns a stream of results. The floating point unit is pipelined taking multiple cycles to produce a result, the MAC service does not stall the pipeline but provides control logic to detect the start and end of new items to accumulate for each row of the matrix.

6.6.4 Pigasus Services

Pigasus services are a subset of streaming services that provide operations to process Ethernet packets in hardware. The set of services available in the catalog together form the Pigasus intrusion detection and prevention system that achieves 100Gbps using a single FPGA and host processor [101]. The Pigasus design features a series of services that process packets through a series of filters that identify suspicious packets. Each Ethernet packet is matched against a set of rules, when a potential match is identified suspected rules are sent with the packet to downstream services for further match processing. Structurally these services provide and require three symmetric channel interfaces for streams of packets, metadata, and rules. Each channel uses the following message format that is compatible with Avalon-Streaming interfaces:

Message Type	Argument	Description
Flit	arg0/arg1	Not used.
	arg2	Channel number for data being transferred.
	arg3	Bits 7-2: Number of empty symbols. Bit 1: End of packet. Bit 0: Start of packet.
	data	Stream data.

Table 6.10: Avalon Streaming Message Format

The Pigasus accelerator system is composed of many modules for processing and stream distribution all adhering to the service abstraction. The larger filter services are composed of these smaller services as "macro" services providing the essential functionalities in the Pigasus design. The set of "macro" Pigasus services available in the catalog are:

- Ethernet
- Channel FIFO
- Channel (de)mux
- Channel Bypass Front/Back
- TCP Reassembly
- Fast Pattern Match
- Port Group Match
- Non-Fast Pattern Match
- DMA Engine

The Ethernet service provides and supplies streams of packets from the Ethernet IP converting from the Avalon Stream interface to the service compatible message format. The channel FIFO service is infrastructural providing buffering capacity, clock crossing, and organizes the packet/metadata/rule streams between packet processing services. The channel (de)mux service respectively splits and combines the packet/metadata/rule streams onto a single channel; this allows multiple streams to share a single channel and share limited infrastructure such as Ethernet ports. The channel bypass service allows the designer to insert a bypass channel FIFO or scale up the pipeline with a parallel service when the initial service asserts its full signal. The reassembly service sorts TCP packets in order so that they can be searched and organizes the full packet payloads before they are passed to the packet-rule matching services. The fast pattern match service selects valid rules at 100Gpbs line rate to pass downstream. The port group match service further filters packet-rule matches by comparing the packet and rule headers. The non-fast pattern match service matches the packet payload against the string pattern identified in the suspected rules. The DMA engine service passes any remaining packets and suspected rules to the host processor where a full match is processed and determined.

Chapter 7

Constructing Communication Channels

Abstract communication is essential to the Fluid's design methodology. Section 4.6 introduced the channel connection types supplied in the Fluid communication as a service library. This chapter details the implementation details of each channel type. It also provides an evaluation study of the effects of choosing a particular connection type. This chapter helps develop an intuition for choosing connection types from the communication infrastructure library and the manner in which their implementation details impact performance.

7.1 Wire and FIFO Connections

Both service and client modules must adhere to the channel interface in order to participate in and benefit from Fluid's methodology. The channel interface is a sender receiver pair, where each port direction is reflected for service and client modules that are connected. The simplest channel connection is a direct connection with wires via a SystemVerilog interface as shown in Figure 7.1. This connection type is the most straightforward and tightly couples clients and services. The connection is purely combinational and synchronous. A FIFO connection decouples the channel interfaces and inserts additional buffering capacity. Inserting this buffer adds minimal delay, but is essential to smooth bursty traffic and minimize stalls. The FIFO connection type supports dual clock domains for clock crossings between streaming services. The capacity of a FIFO channel is parameterizable and large FIFOs make use of BRAM resources for storage.

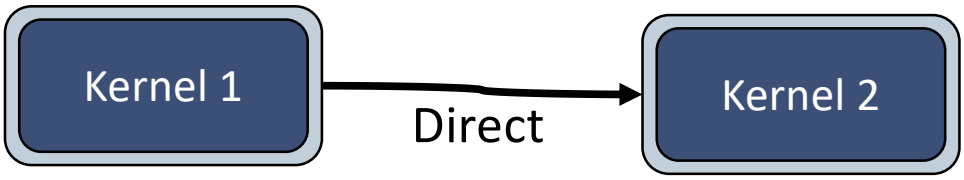


Figure 7.1: Direct connection between streaming kernel modules.

7.2 Connections Through a Network on Chip

Fluid is an abstract and flexible design methodology for FPGA accelerator composition. The methodology's principles abstract FPGA resources and make them accessible to modules outside of their internal design hierarchy. The on-chip communication infrastructure to support Fluid needs to be just as flexible and shareable. Network on chips are common in modern SoC designs, especially to support the memory system and shared communication interfaces. A soft-logic network on chip addresses the flexibility and scalability aims for the communication substrate on the FPGA fabric. Fluid allows designers to implement any channel connection using a network on chip. The network is not restricted to access any single resource or limited to sharing peripheral interfaces. Network on chip connections, just as other connection types, are transparent to connecting modules. The NoC is a shared physical resource on the FPGA fabric where multiple connections share the same NoC. Any given channel—connected through the NoC—is unaware of the topology, flit data width and buffer depth. Channel functionality is not impacted by any implementation choices in configuring the NoC including sharing with multiple other channels.

The Fluid design framework generates a network on chip tailored to the system that the designer specifies. The network on chip is generated by the CONfigurable NEtwork Creation Tool (CONNECT) [72]. The CONNECT tool and networks are adapted to the Fluid's channel interface, and are tightly integrated into the framework. This allows designers to leverage a highly specialized network on chip within their Fluid systems. The goal is that designers can connect modules with the same convenience that comes from modern switched Ethernet and WiFi networks. Designers can therefore parameterize and test network configurations in soft-logic and later leverage this experience as FPGAs evolve to include a hardened NoC. From experience gained working with the NoC within Fluid, designers can recommend architectures and topologies that can be hardened in future FPGA devices.

The CONNECT NoC design is a highly optimized packet-switched network. The tool relies Bluespec SystemVerilog and the language's static elaboration features. CONNECT supports a number of common network topologies and even custom topologies. The router architecture is highly parameterizable in the number of ports, buffer depth, allocation algorithm, routing table, and number of virtual channels (VCs).

Fluid's design tool provides many common network topologies and makes use of router parameters to customize network implementations. The provided NoC topologies include:

- Single Switch
- Line
- Ring
- Double Ring
- Star
- Mesh
- Torus
- Fat-Tree

Fluid designs use CONNECT networks and routers with virtual channels and peek-flow control. Any Fluid channel can be implemented with the NoC with the provided adaptation logic. The Fluid high level programming interface abstracts the implementation details of the NoC, the final design is encapsulated by channel interfaces. All adaptation logic is contained within the NoC wrapper module. These adaptations are handled via the design framework automatically without designer intervention. The design tool maps ports to client and service modules, assigning an individual ID to each module. These IDs are used by the network to route messages through the shared network just as if it were a private physical connection. Wide messages are serialized as individual flits when the NoC width does not match the width of the message. Fluid'd design tool inserts marshalling logic to adapt wide messages to the NoC flit width. Breaking a message into flits does not impact functionality and happens transparently, however this has an impact on performance as it takes multiple cycles to send a message. Narrow flit width NoCs have the advantage of lower resource utilization compared to designs that transmit wide messages in a single cycle. Channel connections sharing the NoC are not required to have the same message bit width. Each channel is individually adapted to the network width, handled by Fluid's infrastructure. The design framework can choose a few methods to adapt heterogeneous channels to a single NoC. The first method generates a NoC with homogeneous channels that match the physical parameters of the widest channel. Reduced bit-width channels are extended and truncated at the NoC edge in the first method. This construction is simple but penalizes reduced bit-width channels if the NoC itself is narrow requiring multi-flit message

transmission. A second method generates a NoC design with heterogeneous channels. This method addresses the inefficiencies of extending reduced bit-width channels on narrow NoC designs. Each channel has tailored logic to disassemble and assemble messages from flits. Reduced bit-width channels benefit as they take a smaller number of flits to be transmitted across the NoC.

Sharing a powerful resource like a NoC does not come without challenges. As the channel interface is most commonly used for exchanging requests and responses between client and service modules, a deadlock can arise in the network. In order to avoid this problem, networks in Fluid designs utilize virtual channels giving priority to response messages. NoC designs that require multi-flit messages also present a challenge when multiple clients connect to and share one service. Flits may arrive at the service module's channel from multiple clients interleaved with one another. To reduce the burden on service modules to track and reassemble the message, Fluid enables virtual links in CONNECT before generating the NoC. Virtual links lock the output ports on the NoC so that mult-flit messages are not interrupted. Each message is guaranteed to arrive in it's entirety so that the message is complete after reassembling sequential flits. Sharing the NoC presents another set of challenges at run-time. As a shared physical resource the network has its own limitations. Multiple channels sharing the NoC must share its bandwidth capacity. Fluid encourages designers to experiment with NoC topologies and configurations to tailor their systems for their required performance and resource characteristics. Run-time introspection and instrumentation features are recommended for client and service modules who's channel connections use the NoC. There is ongoing work to optimize channel ID mapping and placement that takes into account the NoC topology and reduce the number of routers a message touches.

Looking forward to upcoming FPGAs with hardened NoCs; it is not possible to flexibly customize the NoC. Hardened NoCs are advantageous as high-bandwidth communication resources available without consuming logic on the FPGA fabric. However, hardened NoCs have a finite number of ports that must be shared by connecting modules. As an abstract connection type, designers can still use channels to utilize and route communication across these physical resources. While this support is not a part of the current framework for NoC ports, the approach for sharing physical memory interfaces with read/write services and a crossbar is applicable to share a hardened NoC.

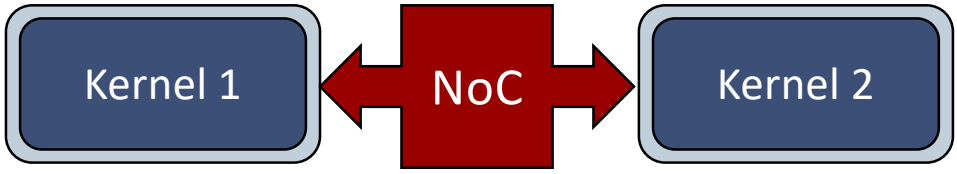


Figure 7.2: Network on chip connection between streaming kernel modules.

7.3 Circular Buffer Connections in Off-Chip Memory

The circular buffer connection type was developed to transfer large amounts of data between FPGA devices on the same host. The circular buffer connection uses a circular buffer data structure located in the host's main memory. This method is useful for systems with multiple PCIe FPGA cards hosted in the same server where no direct communication links, such as Ethernet, exist between the FPGA cards. The circular buffer was developed and evaluated on Intel PAC cards with OPAE. The Intel PAC shell provides a memory interface that allows the logic on the FPGA card to access the host's memory directly. The host code allocates buffers and prepares buffers for sharing through the OPAE SDK. Multiple FPGA cards can share access to the same pre-allocated regions in the host's memory.

The circular buffer connection type is supported by soft logic infrastructure provided in Fluid's design framework. The framework inserts this logic when the circular buffer connection type is specified. The circular buffer is managed entirely in the FPGA hardware and relies on read/write services sharing the same address space in host memory. Each message sent through circular buffer channels is encapsulated with a counter to indicate new messages in the circular buffer. The counter serves as the signalling mechanism between the producer and consumer side of the channel. The consumer logic is more involved than the producer side. Multiple variants were developed and explored to improve the performance of the circular buffer channel. The first option polls sequential index checking the counter on each response for indications of a new message. Multiple parallel requests are sent in parallel to reduce latency accessing host memory. Once a new message is indicated, requests are sent to check the next index and stale responses are disregarded. Each new message received is sent to the receiving channel interface. An optimized implementation of the consumer side polls at an offset index rather than for each subsequent index. Once a new message is indicated at the offset, the consumer enters a streaming mode. In the streaming mode requests are made without checking whether there is new valid message available. Each response is verified that it is a valid new message by the counter, and once the stream is exhausted, the consumer logic reverts to polling mode. The offset is configurable in the optimized consumer logic—the simplest offset choice is to poll for the end of the stream, however choosing an offset closer to the head of the buffer reduces the transfer latency. The consumer can start streaming mode before all messages are present, catching up to the end of a bulk transfer as it is completed.

The circular buffer connection type is also useful for host-FPGA communication. The FPGA logic for host-FPGA communication using a circular buffer reuses the same modules from hardware-only communication. The buffer is directly accessible to software on the host as the circular buffer data structure is located in the host's main memory. Fluid provides a software class to support the channel abstraction

for service threads on the host processor. The software class maintains the channel abstraction, with the same semantics as hardware services. The software channel class writes the message data and a "new message" tracking counter to a cache-line for synchronization in the same manner as the hardware logic. The receive-side channel polls the counter before accessing the data in the circular buffer. This implicit synchronization method signals new data without directly communicating pointers between the producer and consumer. As discussed above, polling is expensive for the FPGA logic. However, due to caching and relatively closer access, software always polls for new data sequentially rather than offset optimizations done in the FPGA hardware.

A slight variation on the circular buffer connection uses MMIO transactions to push and pull data to/from the FPGA for host-FPGA communication. The FPGA provides select control and status registers (CSRs) in soft logic as the backbone of this connection type rather than main memory on the host. Threads on the host processor poll these registers abstracted by a class implementing the channel interface. To send a message, the host pushes data through subsequent MMIO writes to CSRs on the FPGA, through the channel class. Once a message is fully assembled in CSRs, the FPGA logic managing the connection writes the message to a FIFO in the FPGA logic. The FPGA module on receive side of the channel fetches new messages from this FIFO. Similarly, an FPGA module sends a message to the host on the transmit side of its channel where it is then placed into another on-chip FIFO. The host pulls from this FIFO through multiple MMIO reads to CSRs after checking a status flag indicating a new message is available.

7.4 Ethernet Network Connections

Communication across computer networking technologies is essential as FPGAs are increasingly appearing in the data center. Many Ethernet network IPs are available for FPGA devices, however each IP is often different as it must adapt to the specific protocol and performance characteristics of the connection. Fluid provides Ethernet support as a channel connection type supported by two Ethernet frameworks: IKL and Pigasus. Ethernet connections are transparent to modules within the Fluid framework. The Intel Inter-Kernel Links (IKL) [8] communication architecture supports channel connections between modules across Intel Arria 10 PAC devices. IKL is currently supported on the Intel Arria 10 PAC's 40Gbps interface. However, the IKL architecture anticipates that other FPGAs and Ethernet speeds are relatively straightforward to support. Fluid provides the necessary shims to adapt channels to the IKL Ethernet IP interface. Layering support on IKL adds a layer of separation to hide unnecessary details and enable portability for Ethernet connections on future Intel FPGAs. IKL is a streaming protocol and architecture for direct communication between IPs on FPGAs across a network. IKL is designed to be lightweight,

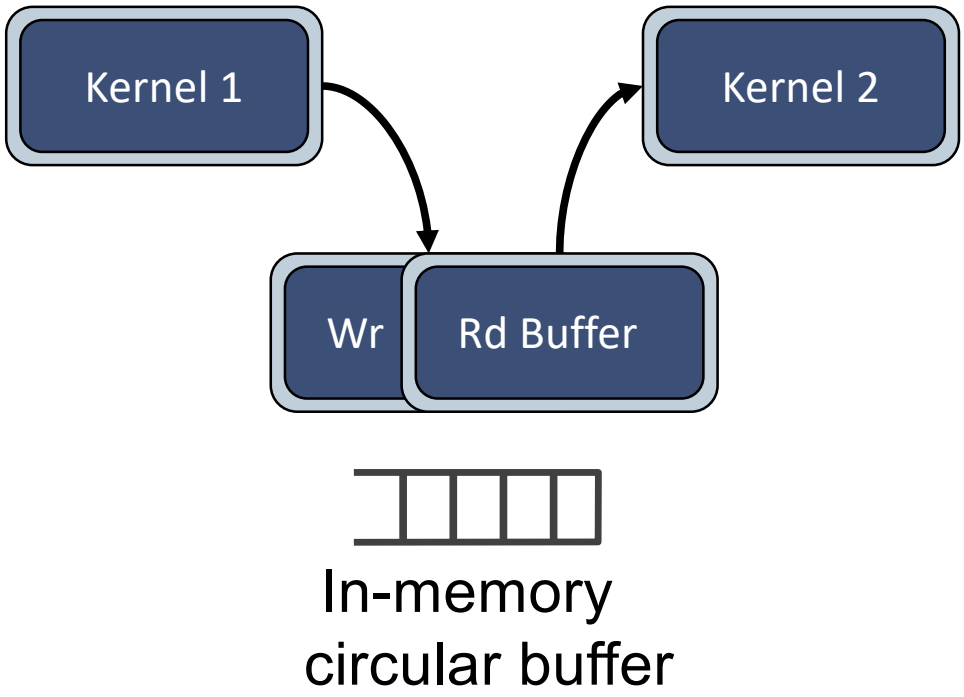


Figure 7.3: Circular buffer connection between streaming kernel modules through host memory.

low-latency, and transport agnostic. The IKL protocol abstracts the details of the network interface and transport interconnect. IKL provides a ready-valid protocol and fixed data width. IKL supports four virtual channels that share a single networking interface on the FPGA. Each of the virtual channels is configurable to independently target at most four other FPGAs in the network through the IKL API. Additionally, the IKL IP builds QoS and reliability into their architecture to guarantee data delivery and deal with flow control. Pigasus supports multi-FPGA communication with a set of Ethernet services. Modules relying on Pigasus Ethernet services must use the Avalon-Stream message format described in Table 6.10. These services transfer Ethernet packets rather than raw data. The Pigasus Ethernet service supports up to 100Gbps currently, more than double the throughput of IKL.

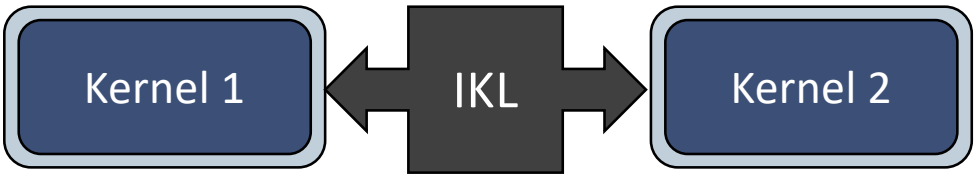


Figure 7.4: Ethernet connection between streaming kernel modules through the IKL IP.

7.5 Evaluating Channel Connections

Selecting a channel connection type, while functionally transparent, has real performance impacts. In the following evaluation, each channel connection is tested to build intuition for design choices. The hardware to hardware connection evaluation was completed on the Intel PAC with Arria 10 FPGA. The evaluation of hardware to software connections was completed on the Intel PAC with Stratix 10 FPGA. All connection types, except for Ethernet over IKL, are available on both Arria and Stratix PACs. The Intel PACs are PCIe-based accelerator cards designed for use in the data center. Each PAC provides a role and shell architecture providing a common interface across FPGA devices. The PACs are supported by the Open Programmable Acceleration Engine (OPAE) SDK which facilitates software access to the FPGAs and the FPGA shared access to host memory. Accelerators on the FPGA are granted access to host memory regions by the host software, and access these regions over PCIe using a proprietary interface called CCIP. Accelerators can use physical or virtual addressing to access host memory. The virtual address mode is configured by the Intel FPGA Basic Building Blocks (BBB) provided in an extension to OPAE. Virtual addresses are cached but initially translated by a software driver on the host. Host software is written in C or C++ to pin shared memory regions and access registers on the PAC FPGAs. The host CPUs in the test setup are all Intel CPUs but their specifics are not germane to this evaluation.

The system design for the channel evaluation is shown in Figure 7.5. The design streams data through two kernel modules that applies a transformation to each item in the data stream. The first kernel module requests a stream of data from a read stream service. The request asks for a number of data items, located in host memory, as a stream of messages. Each message in the data stream has a 256-bit data width, the stream service modules manage the marshalling between the 512-bit interface width and the stream message width. The first kernel then intimates a write stream which is handled by the second kernel and subsequently passed to the write stream service. The write stream service writes each data item into an array in host memory as the final modified result. The read and write stream service modules each require read and write services respectively which abstract the CCIP interface and host memory.

7.5.1 Hardware-Hardware Channels

These experiments evaluate connections for channels between hardware modules, and demonstrate the real performance impacts of channel implementation choices. All modules in these experiments are implemented in the FPGA logic fabric. The evaluation design streams sequential data from an array in the host's memory and the services access the array with physical addresses. The total size of the transfer is varied to cover a relevant set of transfer sizes from a single word to multiple pages. The connection

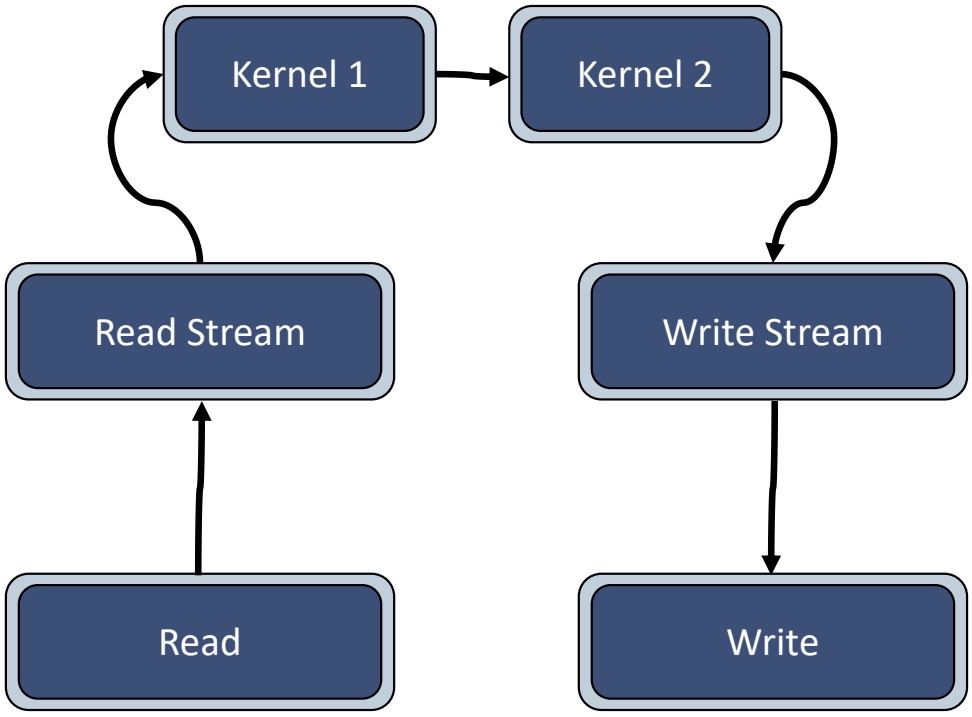


Figure 7.5: Evaluation setup service system for connection substitution between channel interfaces.

between the two kernels is changed for each test. The first setup evaluates "direct" connections which are simply wires between the kernel's channel interfaces. The second implementation tests the channel connection through a NoC requiring multiple flits to send a single message. The third "NoC Share" test changes the connections between the stream and read/write memory services to NoC connections in addition to the inter-kernel connection. The NoC bandwidth is now shared by two additional channels for the read/write stream services communication with the read/write services. The fourth test implements the kernel to kernel connection through IKL and Ethernet. The "IKL Traffic" test adds a traffic generator to the previous test to consume half of the available network bandwidth. The final test implements the kernel to kernel connection with the circular buffer in host memory. In all designs, modules are implemented on the same FPGA device. This choice was made to be able to draw more direct comparisons and isolate the effects of connection choices. While this setup draws a direct comparison, some connection types are more amenable to multi-FPGA systems. The IKL and Circular Buffer connections would only be used by designers for inter-FPGA communication whereas the direct and NoC connections are available only when modules are on the same FPGA device. However, a meaningful comparison does exist for these connections in cases when the connection resources are shared—such as the NoC, IKL, or Circular Buffer.

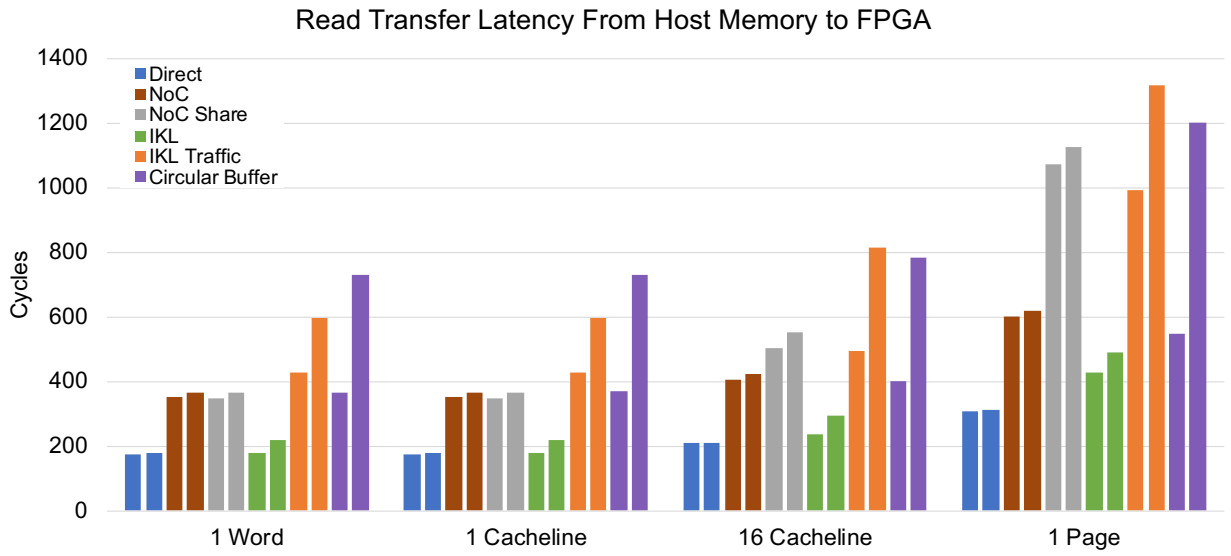


Figure 7.6: Read transfer latency from host memory to FPGA comparing hardware-hardware communication mechanisms with transfer sizes from one word to one page.

Figure 7.6 presents the connection evaluation results for transfer sizes from one word to one page. The experiment measures the latency in cycles to complete the request to transfer data from host memory to each kernel. The results are shown in pairs of bars where each color represents a channel configuration. The first bar is the latency to the first kernel module, and the second bar for the second kernel module. For a direct connection the disparity between each bar is the latency of the first kernel. Any additional disparity for other connection types arises from the structural characteristics of and runtime influences on the connection that may introduce additional latency. For example, the NoC connection requires multiple flits to transmit a message. This was a specific design choice to highlight the impact these choices have, and to make comparisons easier between the NoC and IKL connection types. The IKL connection also requires multiple transactions and cycles to send a message. The largest disparity between bars for a single connection is in the circular buffer test. This difference in latency arises from the implementation details of the channel discussed in Section 7.3, where the logic to fetch from the buffer waits for the transfer to be mostly or entirely complete. Even with this additional latency, the circular buffer is practical to transfer data between FPGA devices if the transfer size is large enough.

Compared to a real use case, the IKL test is optimistic whereas the circular buffer test is pessimistic. In these tests the IKL traffic stops at the PHY loopback, passing the data across a real data center network would incur additional latency. Diversely, the circular buffer test must share the limited PCIe bandwidth to access the buffer data structure in the host's memory. Both the buffer read and write traffic must share the same memory interface when the circular buffer connection is between modules on the same FPGA.

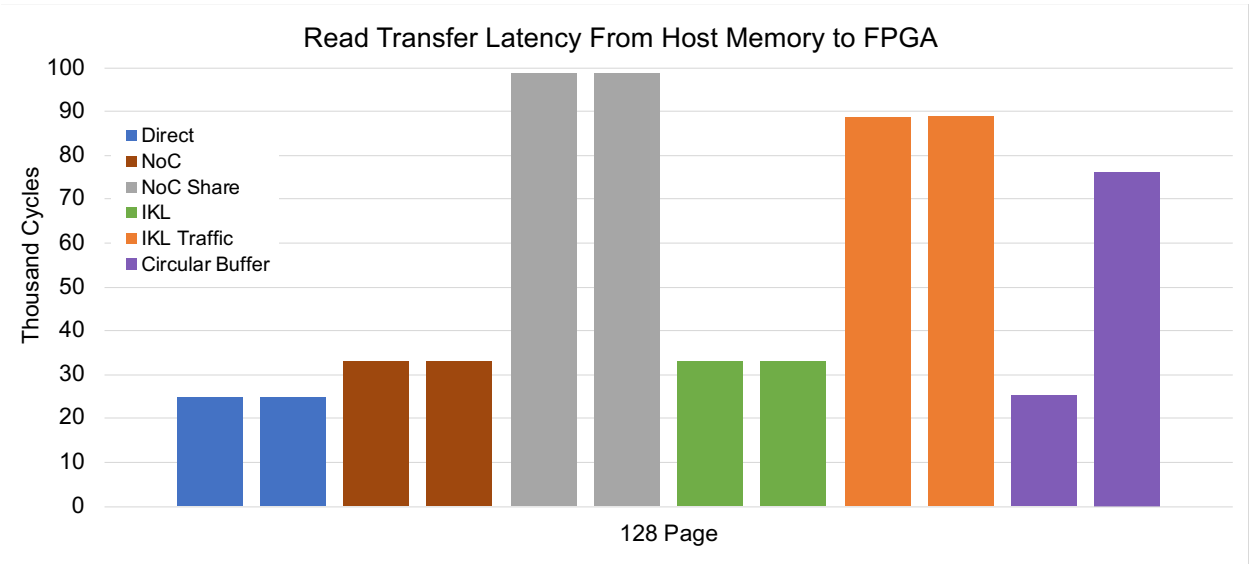


Figure 7.7: Read transfer latency from host memory to FPGA comparing hardware-hardware communication mechanisms for a large multi-page transfer.

This pessimistically represents a real use case where the circular buffer connection type would be used.

An interesting comparison arises from sharing logic resources to implement a connection and/or physical interfaces to external devices. The NoC shared, IKL traffic, and circular buffer tests all demonstrate the impacts of connections across shared resources. The NoC shared test adds real traffic onto the NoC from the read and write services. Sharing the NoC doesn't materially impact the transfer latency until the transfer size grows beyond tens of cachelines. The IKL traffic test adds synthetic traffic to consume network bandwidth, and immediately influences the transfer latency. The circular buffer test doesn't explicitly evaluate sharing, but the impacts of the limited PCIe bandwidth are apparent as the buffer is accessed through this shared interface to host memory.

Even with small transfer sizes in Figure 7.6 sharing external interfaces immediately leads to a 3X increase in latency in the IKL share test. The NoC is impacted less by sharing, as all traffic remains on chip. Furthermore, the NoC has much higher bandwidth compared to the IKL interface. The latency to complete a transfer reach parity across all shared infrastructure connection types as transfer sizes increase. For the largest transfer shown in Figure 7.7 the circular buffer test has the lowest latency for shared connection types. This is somewhat surprising, especially compared to the NoC shared test. However, this demonstrates that the circular buffer is an effective channel implementation for large data transfers between FPGA devices. This is particularly important for systems that do not have Ethernet connections between FPGAs or designers want to reserve Ethernet bandwidth and ports for other purposes.

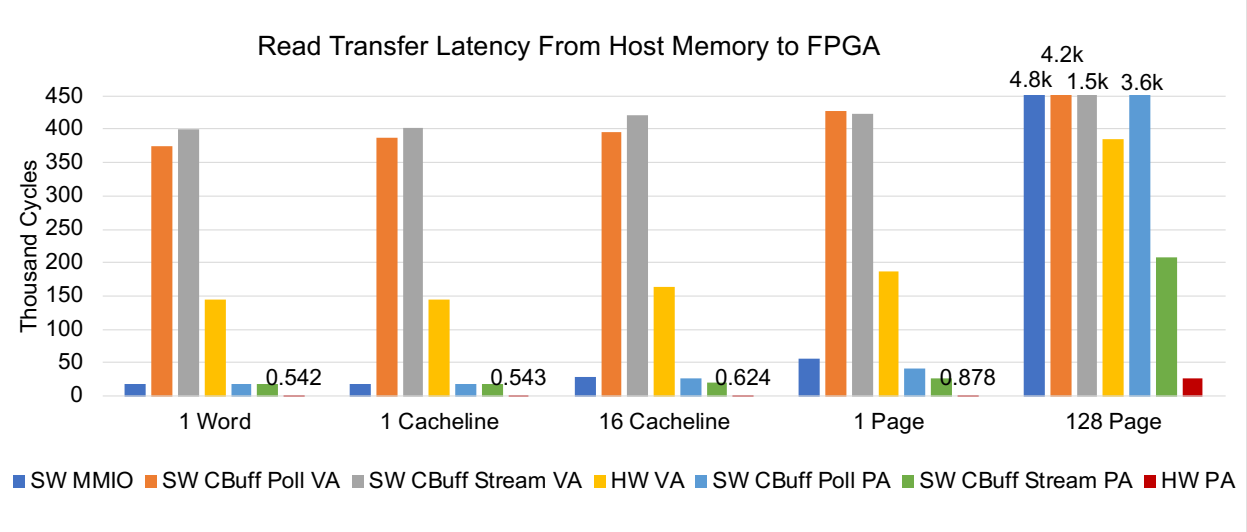


Figure 7.8: Read transfer latency from host memory to FPGA comparing hardware-software communication mechanisms and hardware direct memory access.

7.5.2 Hardware-Software Channels

Communication between software and hardware in the data center setting is just as important and often more challenging. These experiments evaluate hardware-software connections and their performance characteristics. The evaluation setup for these tests uses the Stratix 10 PACs. The communication channels between the kernel module, in the FPGA hardware, and the read stream software thread, on the host CPU, is the primary focus of these experiments.

Transferring data from the host with the assistance of software services.

This experiment mirrors the evaluation of hardware-hardware channel connections for software-hardware connections. In these tests the kernel requests a transfer of data from an array in host memory as a data stream. The stream request is received by the software service thread through a channel, the software thread fetches the required data, and responds with the data stream. Infrastructural modules on the FPGA provide the necessary logic for the communication channels. The software-hardware channels have two implementation choices: MMIO pushed data transfers or through a circular buffer. These channel connection types are evaluated with varying the data transfer size. Both MMIO and circular buffer connection types are completely abstracted from the hardware kernel module and software service thread by the channel interface. The MMIO connection uses multiple 64 bit MMIO transactions for each message sent or received. The effect of using physical and virtual addresses when accessing host memory is also evaluated in these tests. A hardware implementation of the read stream service is provided for

comparison.

The results of the evaluation are shown in Figure 7.8. The MMIO channel transfer latency is approximately the same as the circular buffer latency when the buffer is accessed with physical addresses. Software services and MMIO channels are most useful when the accelerator on the FPGA must use virtual addresses. In this scenario, we recommend that small transfers from host memory be handled by software services and hardware services for very large transfers. Software services that push data through MMIO channels are best—even compared to the hardware stream service implementation—for transfer sizes up to 10s of cachelines. Using virtual addresses greatly increases the transfer latency in all test cases. This is due to the OPAE SDK implementation detail where the software driver does the virtual address translation.

When an accelerator uses physical addresses hardware services always complete transfers in the shortest number of cycles. For software-hardware communication with physical addresses circular buffer channels are the lowest latency implementation. The performance of the circular buffer channel is especially apparent, compared to MMIO, for large transfer sizes. Comparing the results for the circular buffer poll vs stream strategies, especially as the transfer size grows, shows that the streaming optimization improves performance with a 2.7-18X latency reduction.

Chapter 8

Evaluating the Fluid Methodology

This chapter evaluates the Fluid design methodology and demonstrates that raising the level of abstraction for FPGA development increases flexibility, reduces design complexity, and the abstraction doesn't negatively impact the final accelerator design in resource utilization nor performance. We evaluate Fluid's service-oriented abstractions and design framework in a case study of a graph application, a sparse matrix-vector multiplication accelerator design, and a network function accelerator.

8.1 Breadth First Search Case Study Implementation

We selected a graph algorithm, breadth-first search (BFS), as an application case study for Fluid's service-oriented memory architecture. We chose this algorithm as it is memory-intensive, data-centric, and irregular; this provides an opportunity to examine memory services in a stressful and dynamic application.

The baseline accelerator is designed as an elastic pipeline with kernel stages decoupled by memory accesses—this is similar to the baseline FPGA-only accelerator design by Wang et al. [90]. The design aims to exploit available memory-level parallelism, and maximize on-chip data reuse. The baseline accelerator accesses the input graph data at the platform interface granularity. The graph is stored in the stored in the compressed sparse row format (CSR) format allowing for efficient traversals of the data structure. Data access is provided through read/write services shown in at the bottom of Figure 8.1. Each service enables independent virtualized access to a particular part of the CSR data structure. Decoupling the memory services in this way supports the elastic pipeline design where each response from memory triggers a downstream kernel.

The BFS accelerator is a worklist-based implementation where a queue data structure stores pending tasks for iterative processing until the worklist is empty. The baseline accelerator implements the worklist by packing work items into interface granularity bundles stored in a circular buffer in memory. At any

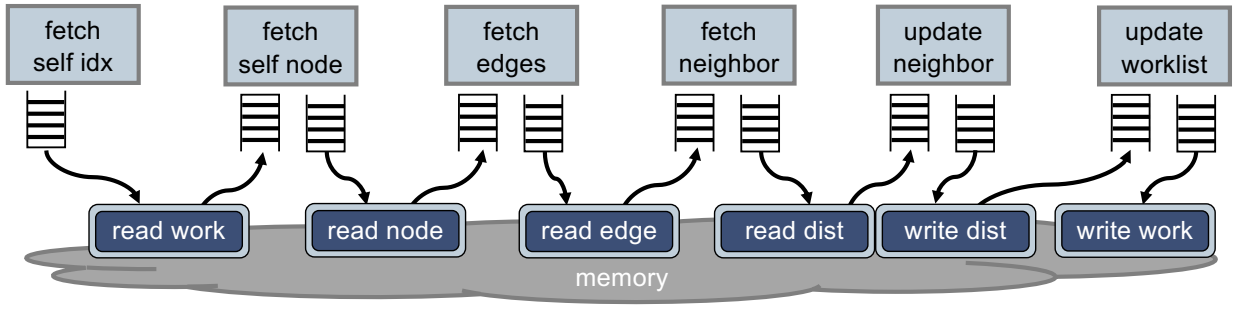


Figure 8.1: Baseline BFS elastic pipeline accelerator with read/write services. Each service enables access to a part of the graph data structure in memory.

point the worklist can contain tens or thousands of pending work items depending on the input graph and phase of the computation. The worklist size is highly dynamic and irregular, the number of items in the worklist cannot be statically determined and therefore must spill to DRAM. Additionally, control logic is required to push partially full work bundles to the in-memory circular buffer if the head of the pipeline is work starved. This type of complexity is trivially managed in software but entrenched in the typical hardware development process.

Our service-oriented accelerator design absorbs complexity from kernel stages and simplifies their code to look much like the algorithm pseudocode. The fully abstracted BFS pipeline is shown in Figure 8.2 with worklist and graph services. The pipeline now has a reduced number of kernel logic stages, and the remaining stages are much cleaner, as the services absorb some of the logic from these stages and surrounding complexity.

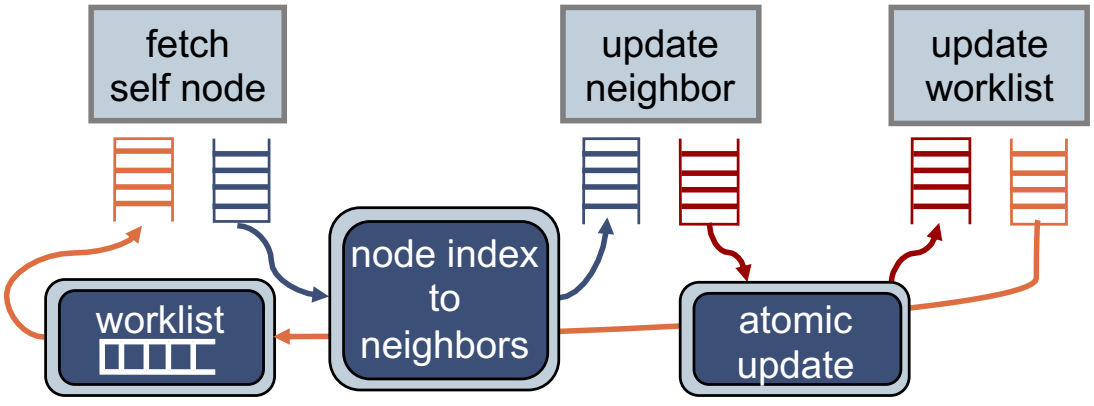


Figure 8.2: Final BFS accelerator including graph services—that abstract the graph traversal and atomic neighbor distance update—and a worklist service.

Graph	Vertices	Edges	Type	Description
rome99	3.3k	8.8k	Directed, Weighted	Road network in Rome.
cond-mat	40k	351k	Undirected, Weighted	Collaboration network on the condensed matter archive.
USA_FLA	1.1M	2.7M	Directed, Weighted	Road network in Florida.
USA_east	3.6M	8.7M	Directed, Weighted	Road network in the eastern United States.
rmat_256k	256k	4.2M	Directed, Unweighted	Synthetic graph with default distribution parameters.
rmat_1m	1M	16M	Directed, Unweighted	Synthetic graph with default distribution parameters.

Table 8.1: Benchmark Graphs Used in Evaluations

8.1.1 Evaluation Setup

For this study, we demonstrate the Fluid framework on the Intel FPGA Programmable Acceleration Card (PAC) D5005 [47] and the Intel Open Programmable Acceleration Engine (OPAE) SDK [46]. This FPGA card features a Stratix 10 FPGA, multiple DRAM channels, and access to host memory over a proprietary interface for the PCIe bus. We use a set of synthetic and standard network graphs [26] as inputs which are summarized in Table 8.1 for this evaluation.

8.1.2 Functional Abstraction

Our first scenario evaluates the overheads introduced by abstracting portions of the BFS pipeline as services. Fluid's memory as a service abstraction addresses the logical complexity of managing high-level data structures in hardware and encourages code portability. Comparable performance is achieved while greatly reducing the complexity of the accelerator pipeline and kernel code—therefore reducing the overall effort to develop a BFS accelerator. Across the set of benchmark graphs we measured the computation throughput in traversed-edges-per-second (TEPS). Abstracting the worklist and graph operations as services does not reduce performance shown in Figure 8.3 nor increase resource utilization as shown in Table 8.2. This is unsurprising as the logic required to provide these functionalities has to be included in the baseline user-level kernel logic without services.

The first benefit of Fluid's abstraction addresses logical complexity that plagues hardware design and leads to long development cycles. The "node index to neighbors" service streamlines the algorithm kernel by providing a compound operation that, with one request, returns not just one neighbor but a stream of neighbors and their distances (so the kernel logic does not need to iteratively index and query each

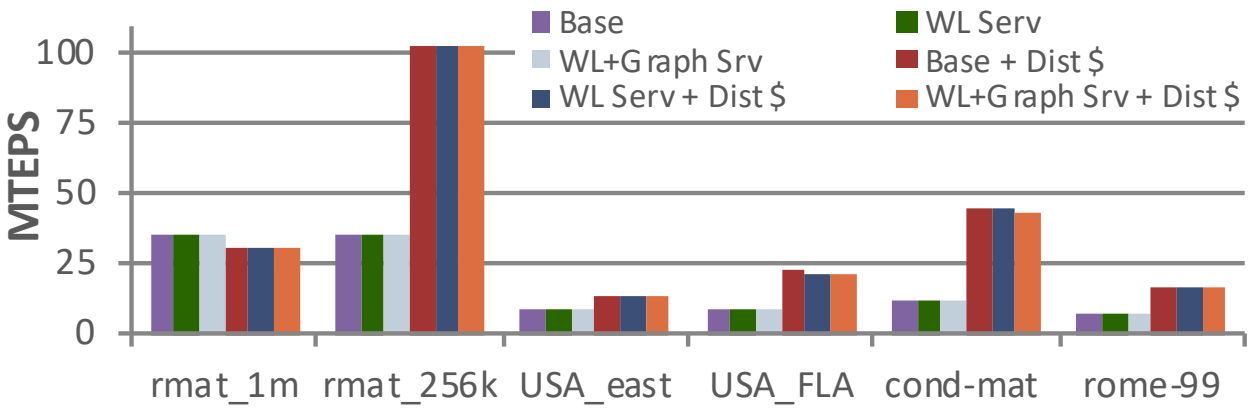


Figure 8.3: Performance achieved in millions of traversed edges per second (MTEPS) for the BFS accelerator for the base implementation, worklist service, and graph services implementations showing that the service abstraction does not negatively impact performance.

Design Configuration	ALM	Registers	BRAM
Base	141495	163643	10590080
Worklist Service	141598	164131	10590080
Worklist + Graph Services	138817	164895	10590080

Table 8.2: Resource Utilization for Service Abstractions on the Intel Stratix 10 PAC

neighbor). The effect of adopting the graph service abstraction is along the lines of factoring out the CSR storage and logic, which would have been included with the algorithm stages, and relocating this logic to the graph services. This encapsulation does not add extra logic cost or logic delay. Moreover, different graph representations could be substituted within the graph service module without affecting the accelerator kernel logic as long as the same interface is maintained.

Service-oriented memory architecture is also effective in addressing hardware-induced difficulties. For example, DRAM interfaces on FPGAs operate on a multi-word granularity as large as 64 bytes. The baseline accelerator needs to account for this inconvenient detail at every turn. When updating a neighbor node's new distance, the baseline accelerator must read an entire 64-byte block, modify the affected sub-word, and write back the entire block. Worse yet, the kernel module must check if back-to-back updates are to the same block to ensure the updates are correctly merged. This kind of complexity arising from the operational and structural specifics of the interface is dealt within the "atomic update" graph service, beneath the abstraction. Similarly, our service-oriented accelerator abstracts the worklist as a service, and the kernel modules interact with this service at the work-item granularity. As far as the designer is concerned, the worklist is an "unlimited" capacity circular buffer. These services support memory operations at the sub-word granularity regardless of the underlying memory interface, resulting in both lower design effort and increased portability. Furthermore, services can be specified with performance

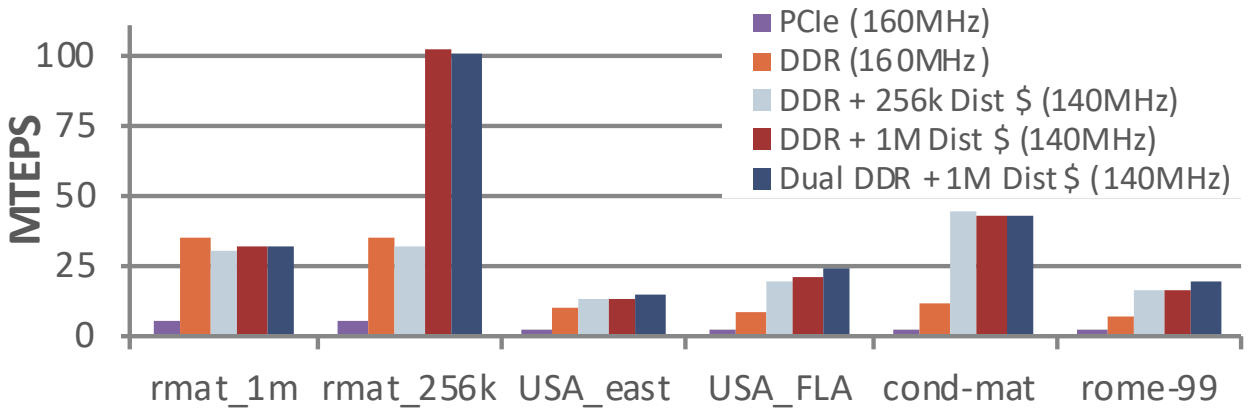


Figure 8.4: Performance across memory devices available in the PAC system and node distance caching for the BFS accelerator shown in MTEPS. These results highlight the flexbility of services across diverse memory interfaces and cache configurations.

enhancements like a cache. We demonstrate this in the "atomic update" graph service, taking advantage of temporal locality in the neighbor distance memory operations.

8.1.3 Memory Abstraction

The second set of experiments evaluates the flexibility of service-oriented memory architecture across various physical memory devices. Changing the memory device behind services is a trivial adjustment with the support of our framework, allowing designers to tune for performance through modifying the registry file without modifying user-level code. We evaluated the fully abstracted BFS accelerator design across five memory device configurations. The read/write services virtualize each interface providing uniform properties such as response ordering. In the first configuration the graph data structure is initialized in host memory. The BFS accelerator accesses the graph over the PCIe bus and CCIP interface. The next configuration initializes the graph in on-board DDR memory. Subsequent tests performance optimizations specified at the service-level. We include a direct-mapped cache for the "visited/distance" tracking array and evaluate two cache sizes. We also evaluate a dual DDR configuration interleaves the data structures across two local memory interfaces.

The results of this evaluation are shown in Figure 8.4. As anticipated, performance improves with each optimization conveniently enabled by making small changes to a high level design file rather than RTL redevelopment. As anticipated, utilizing host memory results in the worst throughput as it has the least available bandwidth and greatest latency. Migrating the graph data structures to local DDR memory provides a 5.8-8.1X improvement in throughput over PCIe accessed host memory. Caching the neighbor distance provides another increase in performance on average across the benchmarks as distance updates

exhibit temporal locality. The throughput of the accelerator is highly input dependent as is expected for irregular graph algorithms. However for some graphs the cache does not increase performance, in fact the cache reduces the achievable frequency and causes a drop in performance. It is notable that changing the memory infrastructure to the dual DDR memory configuration does not lead to the expected performance speedup, this is examined further in Section 8.1.5 using Fluid's introspection features.

8.1.4 Communication Abstraction

The third set of experiments evaluates Fluid's communication abstraction by modifying the BFS design to introduce a network on chip shown in Figure 8.5. Fluid allows designers to quickly port their designs to make use of new connection implementations without impacting their design's functionality. In order to build high performance scalable designs, communication infrastructure just as important as the memory device. Networked communication is the future for scalable designs including hardened network on chips and multi-FPGA systems hosted on an Ethernet network. There is an increased complexity to integrate modules as designs scale for larger FPGA fabrics. Arbitrary connections unnecessarily complicate system integration and reduce programmability. Fluid constrains designers with its communication abstraction, but allows the designer to flexibly parameterize their interfaces and connection implementations at a high level to generate new designs. Moving the communication infrastructure outside of the developer's

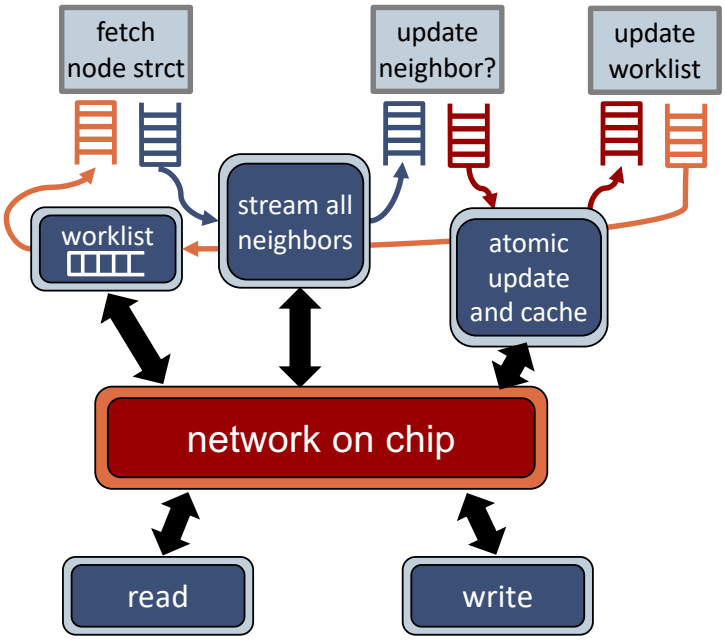


Figure 8.5: The BFS accelerator with a network on chip inserted as the communication architecture between higher-level services and pair of supporting read-write services at the interface.

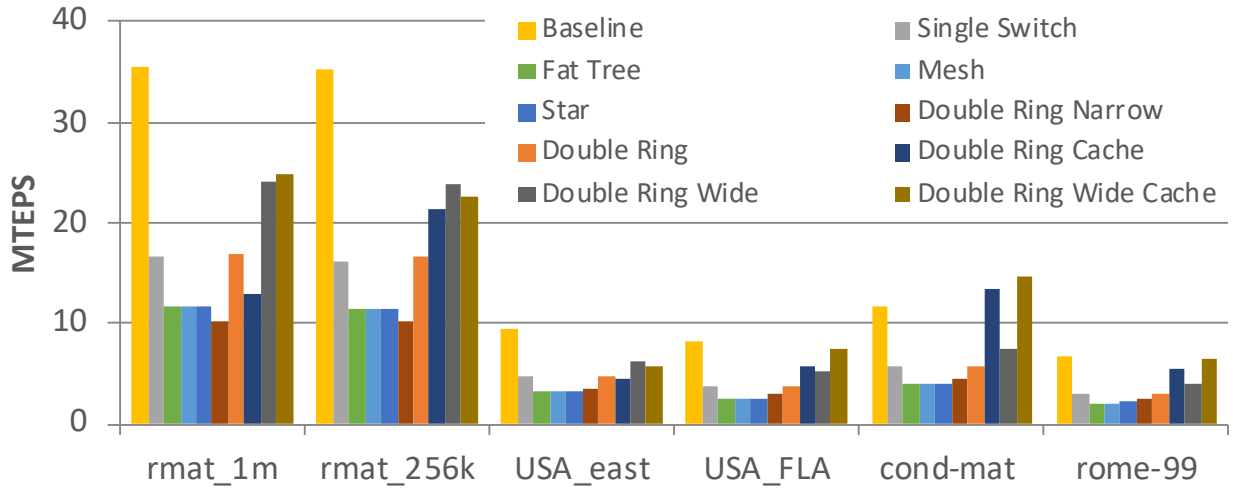


Figure 8.6: Performance across a variety of NoC configurations for the BFS accelerator shown in MTEPS. These results highlight the flexbility of the communication architecture through the NoC.

responsibility unlocks opportunities in optimization and flexibility to adapt to new FPGA architectures and hardened resources as they are made available.

We evaluate Fluid's communication as a service model by substituting direct connections with a shared network on chip. The network on chip is a soft logic implementation discussed in detail in Chapter 7. This demonstration helps evaluate design choices that could be hardened in the future. Fluid's framework allows designers to quickly generate and evaluate network microarchitectures. We adapt the BFS design to use a NoC to distribute data from the high-level services to the read/write services at the FPGA's edge. This modification is made by specifying the NoC connection type in Python. The designer also supplies the NoC parameters for topology and flit width. Fluid's generator builds the NoC, adapts the channel interfaces, and sets routing paths for connections that share the NoC.

The BFS design is evaluated across multiple NoC configurations. We compared each design to the baseline with direct connections in Figure 8.6. We observe that across all benchmarks the NoC degrades performance. This result is due to a combination of factors including: reduced clock frequency, additional cycles to pass through multiple routers, and traffic congestion at routers. The clock frequency is reduced by up to 50% for more complex NoCs featuring more complex routers with many ports. It is also important to note that the soft logic NoC implementation consumes many FPGA resources. Therefore, most configurations we evaluated featured flit widths smaller than the service message width. Multiple flits and therefore cycles are needed to transmit and receive each message, except for the "wide" configurations. The "narrow" networks required eight flits per message and all other networks split each message into four flits.

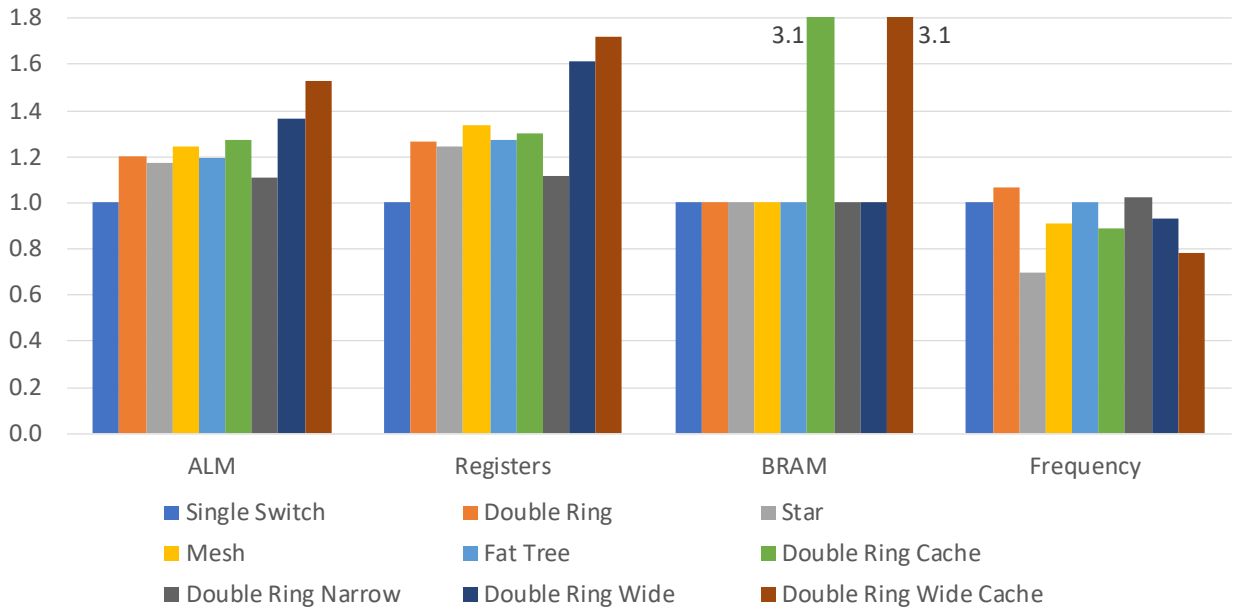


Figure 8.7: Resource utilization compared across a variety of NoC configurations for the BFS accelerator normalized to a single switch network on chip.

We include comparisons that include caches on the client side of the NoC for the atomic update service in our evaluation. Caches are only included for requests that exhibit temporal locality, namely the atomic update read/write sequence. These results are not indented to be compared to the baseline which is provided without caches. Considering just the double-ring network configurations, we observe caches can make up the performance difference for bandwidth lost on narrow NoCs. Performance on narrow networks with caches can even exceed that of full message width networks. However, even with the cache the performance does not reach the baseline throughput without a cache.

As discussed above, the NoC available in Fluid is a soft-logic implementation. Due to this fact, the NoC implementation consumes logic area, increases routing congestion and reduces the design's achievable clock frequency. However, this design study is successful in exemplifying the descriptive capabilities of Fluid's design framework that are enabled by the communication abstraction. Transparently specifying connection types is efficient and convenient through the high level Python interface. Designers can select direct connections or wires editing a few lines of code. The network topology and width are described by the designer in Python by editing their parameters in the application class's instantiation. The network is generated and integrated into the design without requiring the designer to drop below the high level design description. This study demonstrates the improvement in programmability and descriptive capability enabled by Fluid's design methodology and design interface.

The NoC connection type study also provides an exciting opportunity to evaluate NoC microarchi-

tectures at run-time on the FPGA fabric. We compare resource utilization for NoC topologies in Figure 8.7. The results are normalized to a single switch network topology. In our performance evaluation we determined that the simplest topologies result in the best outcomes. The double ring topology emerged as the recommended topology with selective caching. This is confirmed in our resource consumption study where, even with caches, the double ring with multi-flit messages conserves logic resources compared to the full width double ring.

Irregular applications like BFS are an important class of FPGA accelerators. Our evaluation of the NoC was conducted through the lens of this challenging memory intensive application type. For this application class it is important to preserve bandwidth and designers should favor wider networks. The next most important recommendation is that networks should feature simple, efficient router designs. This influenced our choice to deeply evaluate the double ring topology. We recommend simple efficient network topologies for future FPGAs with hardened NoCs. Even with high-bandwidth hardened NoCs, good tools are required to set routing paths to minimize the number of hops and traffic through each router. Developing tools to efficiently map designs on future NoC enabled FPGAs is an important open research question.

8.1.5 Instrumentation and Introspection

As the number of services grow in a complex service-oriented memory system, it becomes difficult to debug and tune services. To ease this burden, our framework supports instrumentation and introspection features through a statistics interface. We provide a set of simple statistics IP cores to probe and instrument service logic. The current set of statistics cores include counters, samplers, and a protocol checker. These cores are available as Bluespec IPs and can be added to any service implemented in that language. Instrumented versions of the Read and Write services are available in the service library. The statistics gained from the instrumentation cores allow designers to gain high-level insights about system behavior and fine-tune their memory systems. Statistics cores are fully synthesizable and therefore can be used in simulation, prototyping, and the final design.

Debugging the dual DDR BFS design. As previously noted the dual DDR memory configuration does not produce a speedup proportional to the increase in bandwidth. The absent speedup can be attributed to a bottleneck caused by either: (1) insufficient reorder buffer space in the underlying read/write services limiting the number of outstanding requests or (2) a short supply of requests from the graph service to utilize the available bandwidth. This is a great case study for performance debugging via the framework's introspection features. Using the instrumentation to track the queue depth and request interarrival time

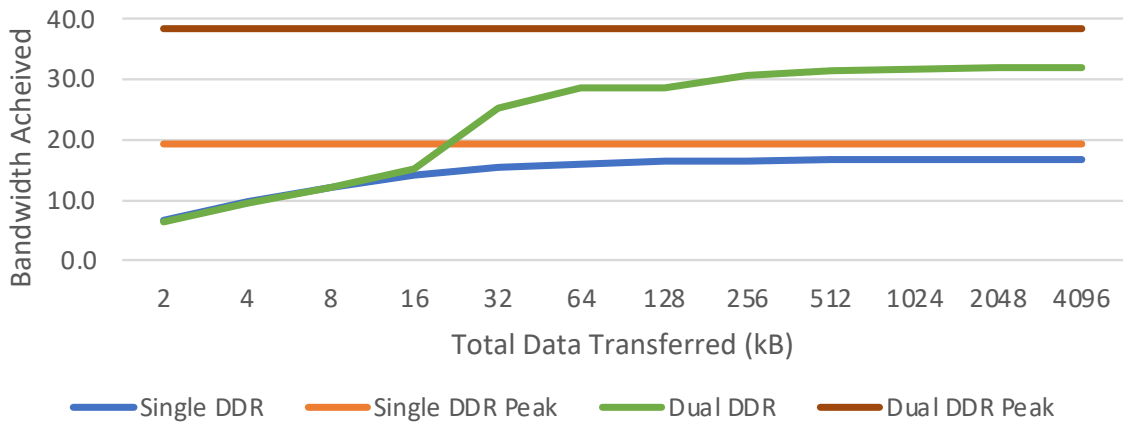


Figure 8.8: Read service bandwidth microbenchmark for single and dual DDR channel infrastructure configurations.

for the read/write service for the visited/distance data it is clear that the bottleneck is in the graph service. The queue depth never reaches the outstanding request limit indicating that it is starved for requests. Furthermore, the interarrival time indicates there are long stalls between new requests. The graph service is unable to utilize the available bandwidth of both DDR channels due to the algorithm selected for the accelerator design. This is a limitation of the graph service implementation that creates a single ordering point that doesn't scale to multiple memory devices. The read after write hazard for updating the visited/distance value stalls the accelerator pipeline limiting its ability to utilize the bandwidth of multiple DDR channels.

Figure 8.8 presents a microbenchmark demonstrating single and dual DDR channel read bandwidth. These results show that it is possible to access the bandwidth of multiple memory channels through Fluid's memory services. The results demonstrate that it is possible to achieve 87% and 83% of the theoretical peak single and dual DDR4 bandwidth respectively. While the BFS application was unable to achieve a speedup from multiple memory channels, another application could utilize this available bandwidth. The underlying infrastructure to connect multiple DDR channels to the read/write services is transparent to the designer, enabling performance improvements unlocked by multiple memory interfaces. Furthermore, designers modify the underlying infrastructure through Fluid's high level programming interface without modifying their kernel modules in RTL.

Evaluating the worklist service's memory demands. While developing and optimizing the worklist service functionality a case for run-time introspection also became apparent. This requirement arose as the worklist service's memory demands are run-time and benchmark-dependent. We monitored the memory demands of the worklist server on its supporting read/write servers. Specifically, we tracked the average

Benchmark	Average	Maximum
rome99	0.02	1
cond-mat	699.12	1036
USA_FLA	27.87	113
USA_east	20.38	208
rmat_256k	85.19	11395
rmat_1m	50.63	44891

Table 8.3: Work-item Bundles Spilled to Memory

and maximum number of bundles of work-items spilled to memory across the suite of benchmark graphs shown in Table 8.3. The results find the demands highly irregular, and independent of graph size. From these insights we determined that using on-chip SRAM was not going to be sufficient as storage, and caching wouldn't have been efficient. Instead, these statistics led us to prefetch ahead of the accelerator to keep up with work-item demands. This demonstrates the utility of instrumentation to assist in service development, in particular with run-time effects on performance.

Instrumentation for design decisions. In our further BFS accelerator design studies we used sampler cores to instrument request completion delay, inter-arrival time and queue depth for each edge memory service. Figure 8.9 highlights the request delay behavior across three benchmarks. In general, we observed that the average delay is much closer to the minimum, however the irregular memory access pattern leads to high maximums. We also observed that the maximum request delay for the work-item write service (WrWrk) relates to the ratio of edges to vertices. The rmat-1m graph has a significantly higher number of edges comparatively resulting in additional updates and more work to process, this is observed in the

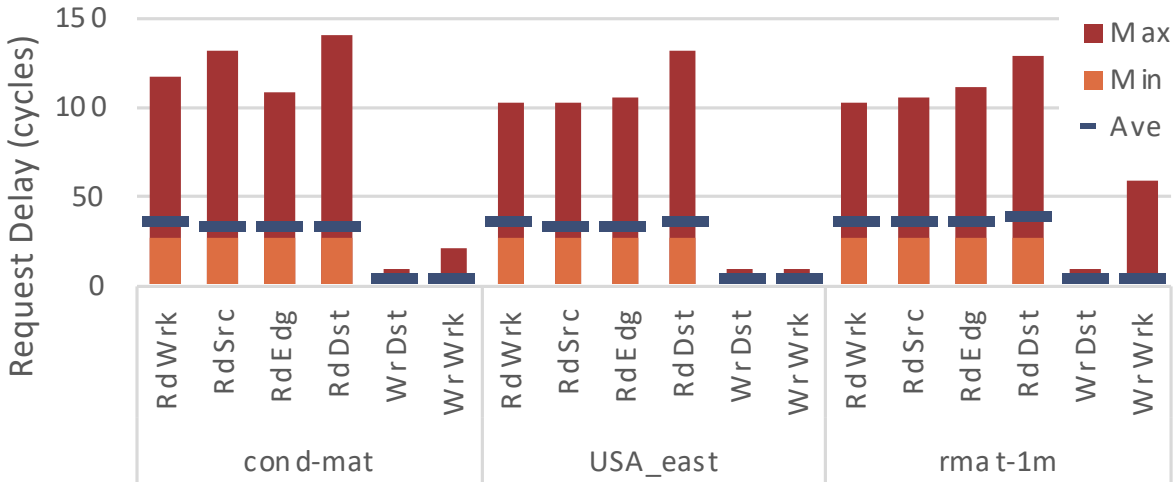


Figure 8.9: Introspection of request delay, enabled by statistics sampler cores, for each edge memory segment service across three benchmarks from the BFS accelerator case study.

WrWrk service.

8.2 Abstracting Data Structure Complexity with Software Services

This set of experiments tests a set of scenarios where services stream data from memory abstracting the underlying data structures. We probe the effectiveness of software implementations of services and fine grained interactions between hardware and software. The high-level system design remains the same as the hardware-hardware channel experiments shown in Figure 7.5 from Chapter 7. However, in some tests the read stream service is implemented by a software thread rather than a hardware module on the FPGA fabric. This substitution occurs transparently, the kernel requesting a data stream is unaware that software is handling the request. We compare services that provide the data stream regardless of the underlying data structure representing the data in memory. The experiments were conducted on an Intel PAC D5005 featuring an Intel Stratix 10 FPGA. The PAC card is hosted by an Intel Xeon processor. Software services and the main host threads are implemented in C++ and use the Intel OPAE SDK.

This evaluation design features a streaming kernel module that requires a general data stream. In these experiments, the data is located in the host's main memory. A read stream service provides a stream of data of a variable length, however the representation of the data in memory is no longer a contiguous array. Accessing data in a pointer-based data structure like a linked-list requires additional complexity that is abstracted away by the service. Clients simply request and receive a stream of data unaware of the data representation in memory. We evaluate the throughput accessing data stored in an array, linked-list and strided linked-list. The linked-list is used to demonstrate how irregular access patterns impact performance. The strided linked-list test input strides the linked-list node locations in memory at an 8kB offset to effect a row-buffer miss in the DDR memory. Additionally, a comparison is made between software and hardware implementations of the services that provide these data streams.

The results in Figure 8.10 show the stream throughput for data transfers. The effects of the data representation in memory are immediately apparent. Accessing data sequentially from an array results in the highest throughput and the pointer chasing required to fetch data from a linked-list reduces throughput. Comparing software and hardware services unsurprisingly favors the hardware service implementation. The hardware service module directly accesses data in memory, without any additional logic delay and communication latency required to support software threads as services. However, software services are initially easier to develop and modify. Software benefits from years of support and experience to simplify development efforts and absorb complexity. Nonetheless, the service abstraction allows the effects of transitioning data structures and service implementations to go unnoticed by the kernel modules on the

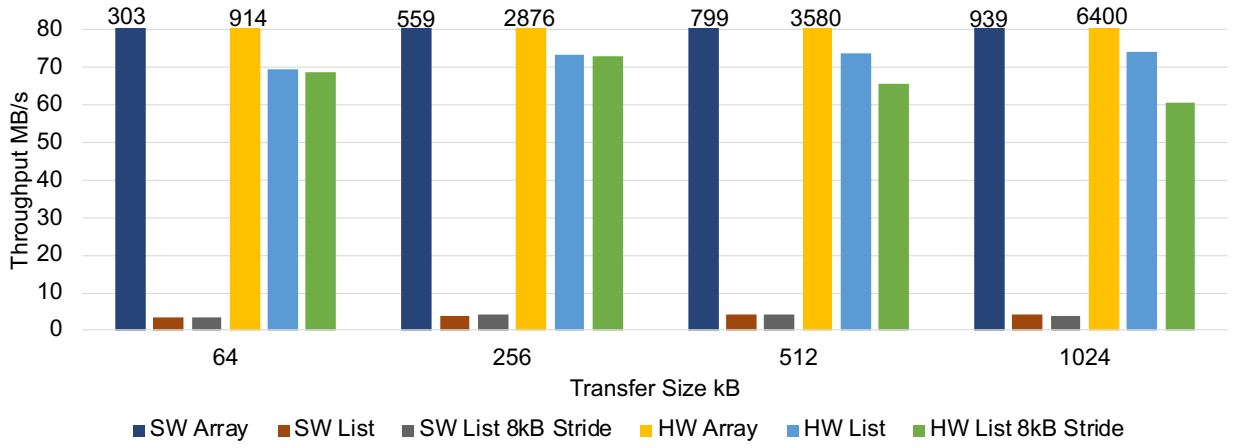


Figure 8.10: Transfer throughput substituting hardware and software implementations of services and targeting different data structures.

FPGA fabric.

We observe that involving software for fine-grained requests adversely impacts raw performance accessing data in memory. Similarly the experiments from Chapter 7 involving virtual addresses suffer a performance hit as the translation is done by software. However, there are some cases where software services are useful for data transfers where small transfers can be pushed over MMIO transactions.

8.3 Sparse Matrix Vector Multiplication Design Study

This design study demonstrates the practicality of service-oriented hardware design and that software implementations of services can simplify development efforts. This section evaluates a sparse matrix vector multiplication (SpMV) accelerator design. Matrix multiplication is a standard computation kernel found in many scientific compute applications. The SpMV accelerator design generalizes the graph service from BFS case study to a much broader application.

8.3.1 Evaluation Setup

This design study was completed on the Intel FPGA PAC D5005 with local DDR memory and access to the host's main memory. The Intel OPAE SDK provides the underlying driver that supports shared access to host memory for the FPGA accelerator. We use the set of input graphs from the BFS case study found in Table 8.1 in the evaluation of our SpMV accelerator design. We also use sparse matrices from Matrix Market [11] as inputs which are summarized in Table 8.4.

Matrix	Rows	NNZ	Type	Description
beaflow	497	53k	Real, Unsymmetric	US economic transactions in 1972.
cry10000	10k	50k	Real, Unsymmetric	Diffusion model study for crystal growth simulation.
memplus	18k	126k	Real, Unsymmetric	Computer component design memory circuit.
fidapm11	22k	624k	Real, Unsymmetric	Matrix generated by the FIDAP package.
af23560	24k	484k	Real, Unsymmetric	Transient stability analysis of a Navier-Stokes solver.

Table 8.4: Benchmark Graphs Used in Evaluations

8.3.2 Functional Abstraction

The previous BFS case study evaluated a scenario where a baseline accelerator was modified and services encapsulated the logic around memory accesses greatly simplifying the final accelerator design. The SpMV accelerator development process was very different and much simpler overall. The SpMV hardware design process is discussed in detail below where the design starts from an available graph service; this service immediately can be tested on the FPGA in a functionally complete SpMV accelerator with software services making up the remainder of the design. This demonstrates Fluid's ability to separate functionality from implementation to simplify the development process. The prior case study on the BFS accelerator used a sparse matrix representation for the input graph. This SpMV accelerator design reuses the graph service from the BFS accelerator and generalizes the graph service for generic sparse matrix and vector access. In this design, the kernel module sends requests to the graph service for each row of the matrix. The service responds with a stream of messages, one for each column. This response, which contains floating point data from the matrix and vector, is streamed through a multiply accumulate processing service, and lastly a read-modify-write service. The SpMV accelerator was designed and implemented with available services in the library and software services.

8.3.3 Simplifying Development through Software Emulation

Although previous experiments identify very limited cases where software services are beneficial in terms of raw performance, supporting software services within Fluid's design methodology provides opportunities to ease the development process. Traditionally, hardware development is a tedious task with long development cycles. RTL development allows designers to build highly specialized designs for performance and/or efficiency. However hardware development can be error prone requiring designers to

account for each bit at every cycle. For example, accessing data in off-chip memory requires hardware designers to account for, often variable, latency and manage data consistency within their designs as FPGA memory controllers lack support for atomic operations. Additionally, arithmetic operations are much more complex in hardware compared to software. Currently available FPGAs feature high-performance hardened floating point multipliers. A hardware designer building an efficient design must account for the multi-cycle latency of the multiply units by pipelining operations. Hardware designers must account for these real timing and structural concerns. Alternatively, software development is much simpler supported by many abstraction layers including the instruction set architecture, compilers, and libraries.

Through the service abstraction, hardware designers can selectively emulate portions of their design as software services. Hardware designers can test their designs in real hardware, emulating portions that are still in development. This design practice was used to develop portions of the SpMV accelerator. The service module that provides access to the data structures in memory was previously developed and made available in the service library as the graph service. The logic to request each row in a matrix is painless to develop with the use of this service. Each row is accessed by a request to the graph service which responds with a stream of messages, one for each column with non-zero data. In our first implementation of the SpMV design, this stream was sent to a software service that completed the multiply accumulate operation. The code for the multiply accumulate software service is shown below:

```
1  void mac_sw_service(rx_channel msg_channel, float* y, int nnz) {
2      int cnt = 0;
3      while (cnt < nnz) {
4          if (msg_channel.notEmpty()) {
5              spmv_msg new_message = msg_channel.rx_msg();
6              y[new_message.arg[0]] +=
7                  new_message.data[0] * new_message.data[1];
8              msg_channel.pop();
9              cnt++;
10         }
11     }
12 }
```

The next implementation integrated a streaming multiply accumulate (MAC) service into the hardware design. This service accepts the responses from the graph service feeding the floating point numbers into a floating point unit on the FPGA. As the underlying representation of the matrix in memory is sparse,

the stream from the graph service is often irregular. A new message is not available each cycle, the MAC service handles this irregularity by predicting the end of each row speculatively. Every new message is fed into the floating point unit, but not every result is written back to memory. The prediction logic controls write requests—as this is speculative, some writes are requested for incomplete rows. Software developers don't consider this type of complexity, the CPU architecture and compiler abstracts these details away. Hardware designers must account for the latency of the multiply, each individual memory operation, and the memory interface granularity. To simplify development and verify the MAC service, its output was once again sent to a software service that would handle the read-modify-write operation:

```
1  void rmw_sw_service(rx_channel msg_channel, float* y, int nnz) {
2      int cnt = 0;
3      while (cnt < nnz) {
4          if (msg_channel.notEmpty()) {
5                 spmv_msg new_message = msg_channel.rx_msg();
6                 if (new_message.arg[1]) {
7                    y[new_message.arg[0]] = new_message.data[0];
8                    cout << "Updating y[" << new_message.arg[0]
9                    << "] = " << new_message.data[0] << endl;
10                } else {
11                   cout << "Skipping y[" << new_message.arg[0]
12                   << "]" << endl;
13                }
14                msg_channel.pop();
15                cnt++;
16         }
17     }
18 }
```

This operation is trivial in software, however it is much more involved to implement in hardware. FPGA memory controllers often do not provide support for masked writes nor atomic memory operations. The final SpMV accelerator substitutes a read-modify-write service as the final stage of the pipeline. Software services support iterative development and testing for each service at runtime on the FPGA. Software services can emulate functionalities as they are developed in hardware. Designers can transparently substitute software services for hardware services without changing their design's functionality.

This allows designers to work in parallel to implement services and test the accelerator design.

8.3.4 Memory Abstraction

This section evaluates memory device portability for the SpMV accelerator. Targeting different memory devices requires only a slight adjustment through the Python specification to build the new design with the help of the generation framework. Migrating from one memory device does not functionally impact the design, and does not impact the user-level logic at all. Each design change, however, does impact the performance of the SpMV accelerator as each memory device has its own set of characteristics.

Each design iteration directly substitutes the memory device without changing the interface width the only design variables are: access latency and memory capacity. We evaluated the SpMV accelerator across five memory devices: host memory with virtual addresses, host memory with physical addresses, local on-board memory, local memory with a cache for the result "y" vector, and on-chip BRAM. Host memory is accessed over PCIe though Intel's CCI and OPAE drivers. Software, as a part of the driver, assists the hardware in virtual address translation. The remainder of the memory device types use physical addresses.

The first set of experiments use Matrix Market inputs. These matrices were selected as they represent real world data sets, with sparsities from 79% to 99%. The matrices were also carefully selected to fully evaluate all memory types including on-chip BRAM. The Stratix 10 FPGA is a large flagship device featuring multiple Megabytes of on-chip BRAMs, large enough to fit the complete data set. The throughput

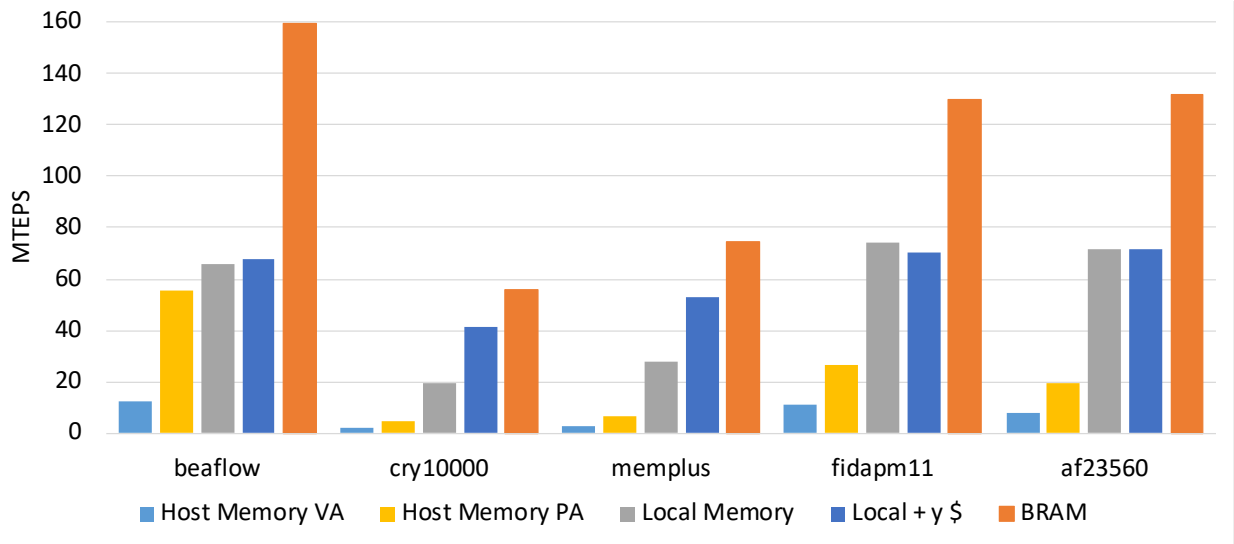


Figure 8.11: Performance across memory devices available on the Intel PAC and node distance caching for the SpMV accelerator shown in MTEPS. These results use standard matrix market benchmarks.

of the accelerator is measured in MTEPS for each memory device shown in Figure 8.11. We observe the overall trend that the densest matrices result in comparatively higher throughput. This is expected as there are more non-zero columns per row on average and more data to process.

From our evaluations we make the recommendation that virtual addresses result in too much overhead, they should only be used as necessary. We observe that host memory in general is much slower except for densest input matrix. It is preferable to contiguous regions of memory so that the accelerator can use physical addresses to access buffers in the host's main memory. Cache performance followed our expectations from the BFS case study. The cache is only helpful on the sparsest matrices and can even reduce performance for some tests. Unsurprisingly, tests where the data is stored in BRAM result in the highest throughput. However, BRAM is only relevant for these matrices because the data-set fits, this is not the case for large data-sets. Interestingly, on very sparse matrices the advantage of BRAM is not that strong compared to local memory with caching. This is understandable as the bandwidth is uniform across all memory types.

Our second set of experiments evaluates the SpMV design using input graphs from the BFS case study. These graphs are much more challenging due to their size and increased sparsity. The evaluation results are shown in Figure 8.12. Most graphs in these experiments are too large to fit in BRAM. While the matrices in the first set of experiments are representative of real data-sets, they were selected to fit on the FPGA. Fluid allows a designer to re-target their accelerator without modifying any RTL as their application space and data-sets change. The memory as a service abstraction improves design performance

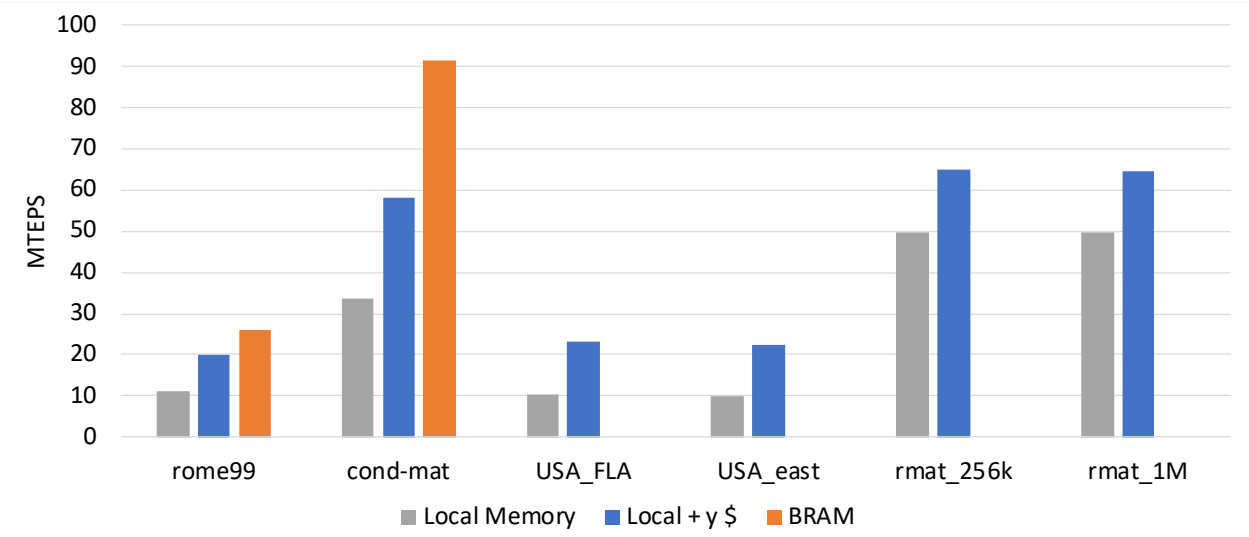


Figure 8.12: Performance across memory devices available on the Intel PAC and node distance caching for the SpMV accelerator shown in MTEPS. These results use the graph benchmarks from the BFS study.

portability allowing designers to experiment at compile time. It is trivial to transparently migrate their service-oriented designs across many memory devices.

8.4 Service-Oriented Pigasus Intrusion Detection and Prevention System

The efficiency and advantages of Fluid were further evaluated in the context of the Pigasus architecture for a Intrusion Detection and Prevention System [101]. This application is highly relevant, challenging, and at the cutting edge of performance. The flexibility and convenience of the Fluid design framework is highlighted in this evaluation where a service-oriented Pigasus design can be generated from a high-level in Python, rather than a typical RTL development flow. We observe that Fluid's design principles, abstractions, and code generation framework help the designer make the best use of the FPGA by increasing programmability and enabling new design opportunities.

Intrusion Detection and Prevention Systems are challenging network functions critical for the security of any network. The triumph of the Pigasus design is that it processes traffic at 100Gbps line rate in a single server with one FPGA and five CPU cores resulting in 38X less power use than a CPU-only approach. The Pigasus design is a *FPGA-first* design, meaning that the FPGA is responsible for the bulk of the processing and accepts network traffic directly; the CPU cores are used only as necessary as a complexity offload engine. The CPU cores implement the "full match" functionality that checks packets against rules identified in the earlier stages of the pipeline. Only a small subset of the network traffic ever reaches the CPU cores.

The service-oriented Pigasus design was a collaborative effort, through which Fluid's design methodology and tool were successfully employed and evaluated. The Pigasus RTL modules were developed by this student, Zhipeng Zhao, for his PhD thesis. In turn, Fluid significantly improved his productivity and increased the flexibility of the Pigasus design. Fluid's design tool and high-level interface significantly improves productivity to generate the RTL for new design choices within minutes. Pigasus accelerators can be generated from a logical description of the design rather than through the tedious and often error-prone RTL development process.

8.4.1 Evaluation Setup

The evaluation setup for the Pigasus design features an an Intel Stratix 10 MX FPGA Development card [45] as a SmartNIC in a 16-core (Intel i9-9960X @ 3.1 GHz) host machine. The Stratix 10 MX FPGA has 16MB of on-chip BRAM, 10MB of eSRAM, and 8GB of off-chip DRAM. The CPU cores run a modified version of Snort 3.0 [81], for the full matcher.

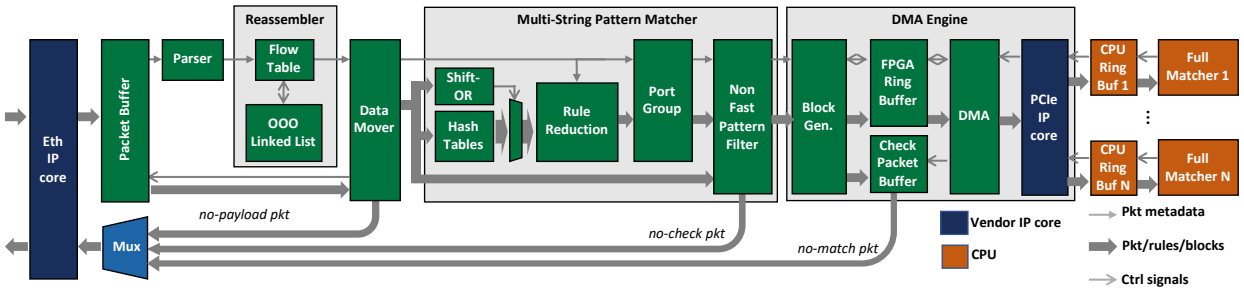


Figure 8.13: Original Pigasus intrusion detection and prevention system [101].

8.4.2 Functional Abstraction

The original Pigasus design features a single FPGA that handles the bulk of the processing for detecting suspicious packets hosted by a multi-core processor. Pigasus processes network traffic scanning packets that match a series of rules that identify malicious traffic. The original FPGA architecture, shown in Figure 8.13, passes packets through a series of filters: (1) TCP reassembly, (2) Multi-string Pattern Matcher, and (3) DMA engine to the CPU cores. The CPU cores process any remaining matching packets and rules in the full matcher.

This original high-performance design was developed with standard hardware design practices in mind and was well suited as an FPGA or ASIC design. However, this approach wasn't necessarily the best for FPGAs to take advantage of their inherent reprogrammability. ASIC-style design practices typically result in brittle designs with poor reusability, scalability, and portability. This is apparent in the original Pigasus design which featured tightly-coupled custom interfaces that limit reusability and lack the flexibility to incorporate emerging communication resources like a hardened NoC. The disadvantages of ASIC-style design are especially apparent in the multi-string pattern matcher (MSPM) that implements the entire packet-rule matching flow in a single module. As a monolithic unit, the MSPM cannot scale each individual rule matching filter without scaling the entire module or modifying the RTL. This "one size fits all" approach is sufficient for common-case network traffic, however it is difficult to determine the common-case for all deployments and 100Gbps line rates are at the cutting edge of current networks. Furthermore, the design is restricted to FPGA devices that can fit the entire design. This design style is especially limiting porting to emerging FPGA deployments with multiple-FPGAs in the data center or devices with hardened NoCs.

An evolution of the design, inspired by the Fluid methodology, refactors the original Pigasus design into a service-oriented Pigasus design shown in Figure 8.14. This service-oriented design breaks each architecturally significant functionality in the original design into disaggregated, parameterizable services.

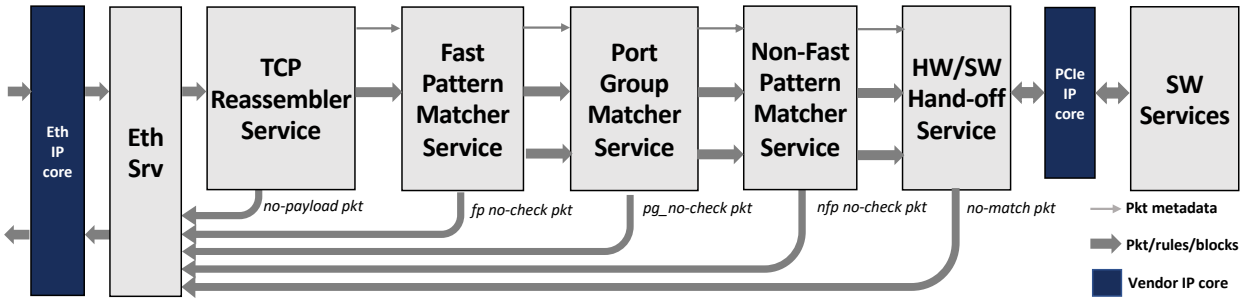


Figure 8.14: Service-oriented Pigasus intrusion detection and prevention system.

Design Configuration	ALM	ALM Percentage	BRAM	BRAM Percentage
Original	207960	29.59	3296	48.14
Service-Oriented	216899	30.87	3357	49.03

Table 8.5: Resource Utilization for Pigasus Designs on the Intel Stratix 10 MX Evaluation Board

As a service-oriented design, the designer is afforded much more flexibility and convenience to make design changes at a high level in Python rather than RTL. The service-oriented Pigasus design first abstracts supporting modules as service modules such as the TCP reassembler, Ethernet, and DMA engines. Then the monolithic MSPM was transformed into a disaggregated set of packet filters: the fast pattern matcher, port group matcher, and non-fast pattern matcher. As a part of this disaggregation, each Pigasus service conforms to Fluid's service interface. Each service is a streaming design providing and requiring three streams for (1) packet metadata, (2) Ethernet packets, and (3) matching rules/blocks.

Through evaluating both designs, we demonstrate that Fluid's design methodology does not compromise performance or produce resource inefficient designs. This is particularly significant in a design like Pigasus that is at the cutting edge for performance in a demanding application. We observe that the service-oriented Pigasus design does not appreciably impact resource utilization when compared to the original design. The resource utilization for the service-oriented Pigasus design is within 1.3% of the original as shown in Table 8.5. In our trace-driven performance evaluation of the service-oriented design, Fluid's abstraction does not impact performance at all. The new design matches the original design's clock frequency, and achieves the same 100Gbps throughput. The service-oriented Pigasus design successfully completes the benchmark traces without requiring any additional processor cores relative to the original design.

After applying the Fluid design methodology, the Pigasus accelerator and scope is transformed from "RTL" or a single design. The service-oriented Pigasus accelerator is now a design space template that lets designers choose the most appropriate configuration to fit their deployment environment. The new Piga-

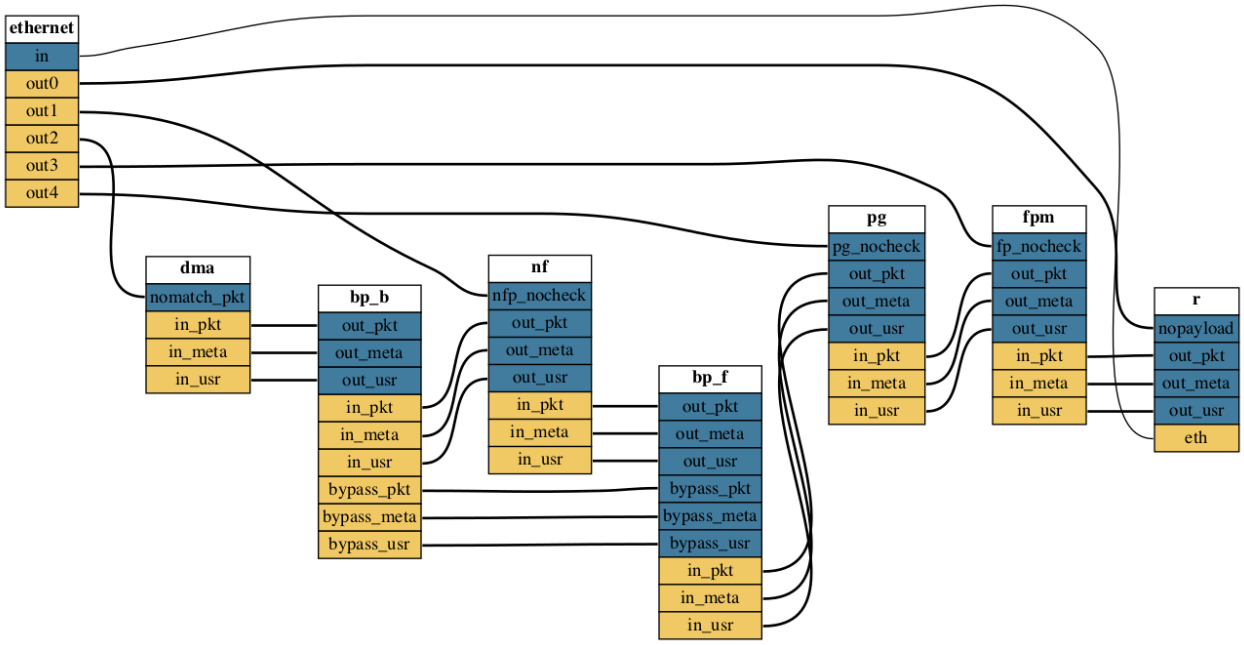


Figure 8.15: Service-oriented Pigasus block diagram from the Fluid design vizualizer.

sus services are "plug-and-play" that allow designers to insert/remove and scale up/down functionalities at compile time. Furthermore, designers develop their accelerator at a high-level without ever touching RTL improving productivity and portability.

The Pigasus services were incorporated into Fluid's service library enabling fully generatable designs from a high-level architectural description in Python. Figure 8.15 shows the output of the Fluid design vizualizer for the service-oriented Pigasus design. In this diagram each main block represents a service or significant infrastructure; the abbreviated names given by the designer in their Python description are in the white header. Channel interfaces are represented in the colored boxes, and connections are represented with solid lines. These lines represent abstracted connections, in this case they are custom FIFOs that match the Pigasus service structure and handle buffering and clock crossing. Not shown in the diagram are clock assignments. The designer specifies clocks with the same syntax (`add_extern_connection`) for unstructured ports.

Within the Fluid design framework, designers can conveniently reorganize and adapt the Pigasus design to fit their deployment environment. Any design change is made at a high level modifying tens of lines of Python code rather than hundreds of lines of RTL. The design framework introduces incredible design flexibility at a low cost allowing the designer to add or remove Pigasus services and scale up/out their designs. For example, the components in the diagram labeled "bp_f" and "bp_b" support bypass capabilities. If a Pigasus service is overloaded the traffic bypasses the service as shown in Figure 8.15.

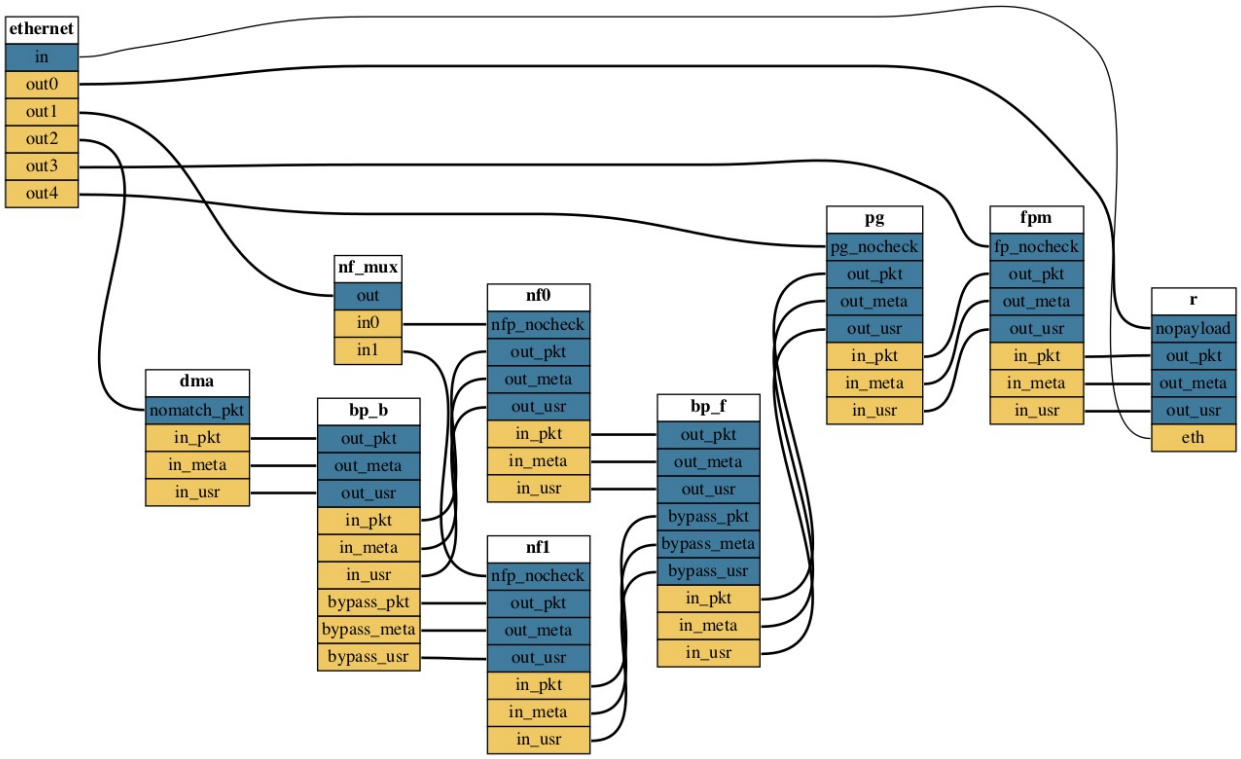


Figure 8.16: Pigasus block diagram demonstrating scale out capabilities in the non-fast pattern matcher service.

The designer could alternatively instantiate another copy of the service in parallel in order to scale out to handle the overflow traffic as shown in Figure 8.16. In fact, this design pattern can be encapsulated in a reusable function and applied to any service in the Pigasus pipeline. Simplifying hardware modules through well-defined abstractions and flexible infrastructure increases the designer's productivity and description capability. These types of design changes are simple to make working within Fluid's design framework where the accelerator is described logically and generated. This provides a significant improvement over explicit wires and infrastructural modules in RTL.

8.4.3 Communication Abstraction

The Fluid design framework and communication abstraction enables new design opportunities outside of the scope of the original Pigasus design. Partitioning the Pigasus design across multiple FPGAs, is possible and convenient through the Fluid design framework. Exploring design opportunities at a high-level is afforded by Fluid's methodology and convenient descriptive capabilities. Designers capitalize on the additional flexibility afforded by abstraction and flexible generatable infrastructure. Fluid's framework makes each Pigasus service both abstract and placeable—each packet-rule matching service is available in

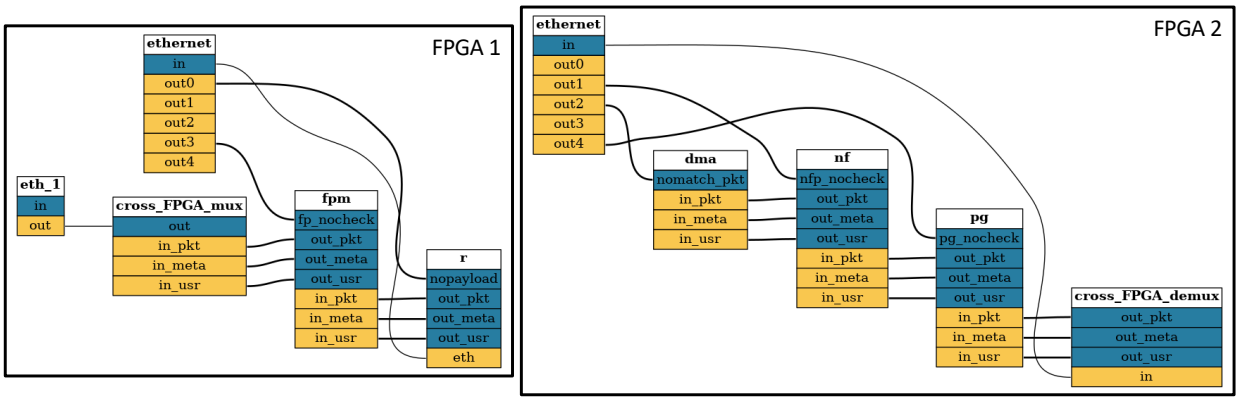


Figure 8.17: Dual FPGA Pigasus block diagram from the Fluid design vizualizer.

the service library. Designers describe their Pigasus service pipeline logically connecting each service at a high level. The design framework allows the designer to insert infrastructure into the streaming pipeline so that it can span multiple FPGAs. Figure 8.17 provides an example of a multi-FPGA Pigasus design generated in the Fluid framework.

As network line rates continue to scale, the Pigasus design can scale up to meet the required throughput. However, each service's resource consumption will scale commensurate with performance. Multiple FPGAs may be required to host the entire Pigasus service pipeline in this future scenario. Designers, through the Fluid design framework, can scale the design and place services across multiple FPGAs without touching the RTL. Fluid provides the inter-FPGA infrastructure and description capabilities which boost the designer's productivity. Modifying the placement of Pigasus services requires the designer to change about 17 lines of Python code compared to hundreds of lines of RTL code. Reusable functions that encapsulate multiple connections are one key to unlock productivity. The function below connects a source and destination service, connecting the inter-FPGA infrastructure in a single line of code:

```
1 connect_cross_fpga(app0, app1, fpm2pg_cf, "fpm2pg",
2                    cross_FPGA_mux, cross_FPGA_demux, fpm, pg)
```

The function requires multiple arguments which are references to the services and required infrastructure instantiated by the designer in their design description. The first two arguments are references to the FPGA applications the designer is building for each FPGA device. The next two arguments are a reference and name for the FIFO buffer between the services; this is replicated in the FPGA applications by the function. The next two arguments are references to the cross-FPGA infrastructure that combines packet, metadata, and rule channels to a single stream. References to the source and destination services, respectively, make up the last two arguments to the function. Once the design is partitioned across multiple

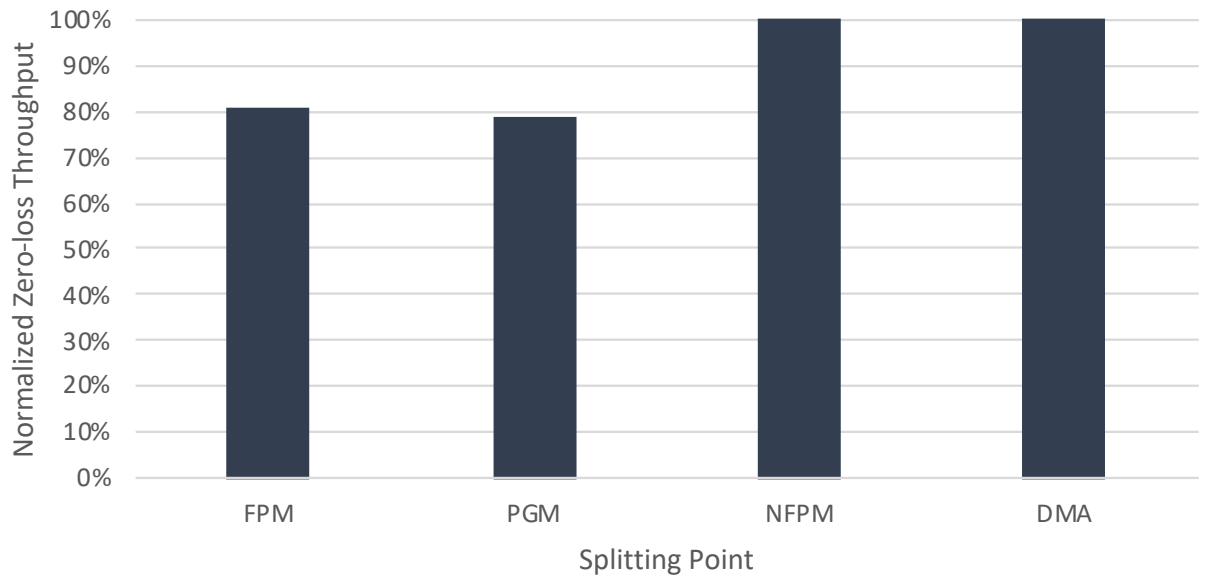


Figure 8.18: Normalized throughput of Pigasus for cross-FPGA split points before the labeled service.

FPGAs the designer can use parameters to tune the performance utilizing additional available resources.

The throughput of the Pigasus design was evaluated at various splitting points to demonstrate design productivity and infrastructure flexibility. In these experiments an additional Intel Stratix 10 MX FPGA is added to the host and connected to the first FPGA though an additional 100Gbps Ethernet port. We evaluate the throughput of each design normalized to the throughput for a single FPGA design. The service parameterization is held constant so that the experiment evaluates the impact of the splitting points and infrastructure. Figure 8.18 shows the results of the experiment for each splitting point. The splitting point is defined as inserting the multi-FPGA infrastructure before the named service module. Each splitting point has different performance, only the final two designs match the performance of the single FPGA design. Splitting before the fast pattern matcher and port group matcher results in reduced performance compared to the single FPGA design. The performance drop is the result of increased traffic that has to cross the inter-FPGA Ethernet connection. The increase is associated with matching rules and metadata that must accompany the packets to downstream services. However, in the non-fast pattern matcher and DMA split points the bulk of the traffic has been filtered out by the previous services leaving headroom for the additional traffic. It is important to note however that this impact is input-dependent. A designer must choose a split point that fits their deployment scenario. Fluid's design framework and Pigasus services allow the designer to generate new designs with minimal effort compared to the conventional RTL development process. Fluid's design methodology increases productivity and design flexibility enabling the designer to quickly generate their design to meet their use case.

Chapter 9

Conclusions

Specialized hardware accelerators are increasingly popular as alternatives to general purpose processors in order to satisfy performance requirements in tight power envelopes. FPGAs offer attractive advantages as compute devices to build specialized compute circuits, while retaining flexibility to adapt to new applications over time. Furthermore, FPGAs are more capable with every generation featuring larger logic fabrics, hardened compute resources, high-performance memories and network interfaces. However, a major obstacle for FPGA's widespread adoption is their programming model. Developing custom hardware accelerators is tedious working with low level abstractions, like RTL, at the bit and cycle level. Compounding this problem, designers are faced with increasingly complex applications, larger FPGA fabrics, heterogeneous FPGA architectures, diverse memory interfaces, and elaborate datapath infrastructures. Furthermore, FPGAs are still programmed like static, ad-hoc, single-purpose designs. FPGA's are inherently more flexible than ASICs and their programming model should reflect this. In order for FPGAs to absorb applications from software, hardware developers require the same level of support software developers have come to expect through abstraction.

This thesis presents Fluid, a methodology for hardware development that provides a service-oriented abstraction and high level design framework to increase FPGA programmability without compromising performance. Fluid develops a modular abstraction that factors the complexity of memory and communication logic out of an accelerator designer's kernel modules. Memory services encapsulate high-level semantic rich operations on data in memory and provide designers starting points at an abstraction level higher than loads and stores. The communication as a service abstraction enables logical connections between modules in an accelerator design, and allows designers to specify these connections abstractly. Fluid provides a working catalog of services and communication infrastructure library that realize the abstract services presented to designers. Through Fluid's design framework, designers specify a service-

oriented accelerator, at a high level in Python, comprised of their kernel modules and service modules. The design framework provides a convenient extensible programming abstraction that allows designers to architect their accelerators and control implementation details without touching RTL. The design framework generates a top level system design in SystemVerilog that instantiates the modules specified by the designer and elaborates necessary communication infrastructure. Fluid (1) absorbs the complexity of memory and communication infrastructure from designers' modules without increasing overhead or compromising performance, (2) enables higher levels of abstraction for hardware design by decoupling functionality from implementation, and (3) supports designers' to conveniently architect specialized accelerator systems on FPGAs at a high level with transparent control over implementation details.

The Fluid design methodology is an important step towards increasing FPGA programmability by absorbing complexity in accelerator design and supporting the widespread adoption of programmable hardware in the computing landscape. Through abstraction, designers can rely on expertly crafted modules to access data in memory, conveniently compose accelerators at a high level, and transparently employ a variety of communication substrates in their datapaths. Currently, rapid development and convenient experimentation is relatively simple in software whereas hardware development is reserved for experts. Fluid's methodology mitigates this disparity by increasing FPGA programmability through abstractions that reduce complexity and increase design flexibility without compromising performance.

9.1 Future Directions

Fluid demonstrates that abstractions increase programmability introducing flexibility to optimize and adapt to new design implementations, and structure simplifies design generation. The implementation of the abstractions presented by Fluid for memory and communication demonstrate its value, however the key principles identified extend beyond the hardware and tools that were developed for this thesis. Fluid's approach can be extended toward many exciting research directions and has currently influenced ongoing work to increase FPGA programmability.

- **Abstract Communication Interfaces.** Fluid currently restricts services to a physical, RTL compatible, channel interface. While this design choice demonstrates Fluid's paradigm, it limits the abstraction to logic outside of the designer's modules. Extending the abstraction into the designer's modules would be an interesting future direction. It would be interesting to investigate abstract interfaces without defined timing or structure to enable automatic interface elaboration.

- **Service Generation.** Fluid provides a library of services supporting some typical operations on

data useful for FPGA accelerator design. This set is enough to demonstrate the types of operations services provide and were developed in RTL by hand. The library of available services could be extended to more operations from a high level language. It would be useful to generate service modules from high level descriptions so that Fluid is less reliant on hardware experts to produce the service library. Later on experts can introduce highly tuned versions of generated services. Additionally incorporating support for IP generators that produce designs competitive with expert RTL would increase Fluid's capability. Some initial work has been investigated incorporating HPIPE [38] designs as services for machine learning inference.

- **Software Services.** Fluid currently supports software implementations of services however their practical use could be extended. In particular, software services running on multiple architectures such as embedded processors on the FPGA fabric, AI processors, or even GPUs. The service abstraction doesn't inherently restrict the implementation of a service's function, and incorporating hardened and embedded processing units available on some modern FPGA platforms would be an appealing extension. Furthermore, services on the FPGA logic fabric can increase the efficiency of downstream compute units by reorganizing and pre-processing data as it flows through the system.

- **Dynamic Partial Reconfiguration.** This thesis focused on abstractions to increase programmability in a static FPGA design. Dynamic partial reconfiguration is an emerging direction to increase FPGA programmability by substituting portions of the design at runtime. The principles and lessons learned from Fluid can apply directly to dynamically reconfigured FPGA designs. Specifically, various service module implementations can be substituted at runtime optimizing the design for the current power, performance, and area constraints. Additionally, service alternatives targeting different data structures, applications and/or performance optimizations would be interesting to investigate through dynamic partial reconfiguration. Research investigating this topic is currently in progress exploring services for different graph representations (sparse vs dense) and various cache sizes in the BFS application discussed in the evaluation.

- **Hardened Resources.** Fluid's communication as a service abstraction encourages the incremental adoption of new communication technologies. Fluid's design framework provides support for a highly configurable network on chip as a connection type. This infrastructure could be replaced by a hardened network on chip, available on some cutting edge FPGA devices. It would be interesting to investigate how these resources introduce efficiency and performance gains, and/or unlock additional design possibilities. Support for a hardened network on chip, or other communication interfaces could unlock larger and even more scalable service-oriented accelerator designs.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association. 17

[2] M. S. Abdelfattah and V. Betz. The case for embedded networks on chip on field-programmable gate arrays. *IEEE Micro*, 34(1):80–89, Jan 2014. 2, 18

[3] Achronix. Speedster7t Network on Chip User Guide (UG089). https://www.achronix.com/sites/default/files/docs/Speedster7t_Network_on_Chip_User_Guide_UG089_2.pdf, June 2020. 2, 18

[4] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 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, pages 25–28, New York, NY, USA, 2011. ACM. 19

[5] Altera. Quartus Platform Designer. https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-platform-designer.html. 2, 18

[6] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131, 2009. 15

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, June 2012. 16

[8] Susanne M. Balle, Mark Tetreault, and Roberto Dicecco. Inter-Kernel Links for Direct Inter-FPGA Communication. https://www.intel.com/content/dam/www/programmable/us/en/others/literature/wp/wp-01305-inter-kernel-links-for-direct-inter-fpga-communication.pdf, 2020. White Paper: Intel Corporation. 58, 98

[9] Jon Beare. Data streaming with arka runtime apis, Jan 2021. 21

[10] Bluespec, Inc. Bluespec System Verilog. http://www.bluespec.com/products/bsc.htm. 16, 28, 52, 66

[11] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: A web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, page 125–137, GBR, 1997. Chapman Hall, Ltd. 119

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. 17

[13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM. 16, 17

[14] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 9 2001. 5, 31, 43, 49

[15] Luca P. Carloni. From latency-insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151, 2015. 5, 8

[16] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016. 23

[17] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Proceedings of the 2008 Symposium on Application Specific Processors*, SASP '08, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society. 1

[18] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. 17

[19] Shaoyi Cheng and John Wawrzynek. Architectural synthesis of computational pipelines with decoupled memory access. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 83–90, 2014. 17

[20] Yuze Chi, Licheng Guo, Jason Lau, Young kyu Choi, Jie Wang, and Jason Cong. Hls-compatible embedded-processor stream links. In *Proceedings of the 2021 IEEE International Symposium On Field-Programmable Custom Computing Machines*, FCCM 2021. IEEE, 2021. 16

[21] Jeffrey Chromczak, Mark Wheeler, Charles Chiasson, Dana How, Martin Langhammer, Tim Vanderhoek, Grace Zgheib, and Ilya Ganusov. Architectural enhancements in intel® agilex™ fpgas. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 140–149, New York, NY, USA, 2020. Association for Computing Machinery. 2, 14

[22] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM. 20

[23] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society. 1

[24] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and evaluation of the coram memory architecture for fpga-based computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 139–142, New York, NY, USA, 2012. ACM. 20

[25] Intel Corporation. Intel Annual Report (2020 Form 10-K). https://www.intc.com/filings-reports/all-sec-filings/content/0000050863-21-000010/0000050863-21-000010.pdf, January 2021. 15

[26] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. 109

[27] John Demme. Elastic silicon interconnects. *Workshop on Languages, Tools, and Techniques for Accelerator Design*, 2021. 21

[28] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. 1, 14

[29] Nariman Eskandari, Naif Tarafdar, Daniel Ly-Ma, and Paul Chow. A modular heterogeneous stack for deploying fpgas and cpus in the data center. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 262–271, New York, NY, USA, 2019. Association for Computing Machinery. 21

[30] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association. 1, 15

[31] K. Fleming, H. J. Yang, M. Adler, and J. Emer. The leap fpga operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014. 8, 19, 20

[32] Shane T. Fleming and David B. Thomas. Using runahead execution to hide memory latency in high level synthesis. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–116, 2017. 17

[33] 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, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale DNN processor for real-time AI. In

Proceedings of the 45th Annual International Symposium on Computer Architecture, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press. 1, 15

[34] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. Spiral: Extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018. 17

[35] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 84–93, New York, NY, USA, 2019. Association for Computing Machinery. 1, 14, 18

[36] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019. 22

[37] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner. Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA. In *Proceedings of 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 46–56, 2016. 1

[38] Mathew Hall and Vaughn Betz. Hpipe: Heterogeneous layer-pipelined and sparse-aware cnn inference for fpgas. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 320, New York, NY, USA, 2020. Association for Computing Machinery. 135

[39] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001. 2, 43

[40] B.L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 111–120, 2002. 17

[41] Intel. Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. 2, 17

[42] Intel. Intel HLS Compiler. https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html. 2, 16

[43] Intel. Avalon Interface Specifications. https://www.altera.com/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf, 2018. 18, 62

[44] Intel Corporation. Intel Stratix 10 DX FPGAs. https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-dx.html. 43

[45] Intel Corporation. Intel Stratix 10 MX FPGA Development Kit. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-s10-mx.html. 1, 125

[46] Intel Corporation. Open Programmable Acceleration Engine - Documentation. https://opae.github.io/latest/index.html. version 1.4.0. 58, 109

[47] Intel Corporation. Intel FPGA Programmable Acceleration Card D5005 Data Sheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-d5005.pdf, 11 2019. DS-1058. 2, 109

[48] Valeriu Manuel Ionescu. The analysis of the performance of rabbitmq and activemq. In *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, pages 132–137, 2015. 22

[49] Nachiket Kapre and Samuel Bayliss. Survey of domain-specific languages for fpga computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–12, 2016. 17

[50] Olof Kindgren. Invited Paper: A Scalable Approach to IP Management with FuseSoC. In *Workshop on Open Source Design Automation*, OSDA '19, 2019. 18

[51] R. Kirchgessner, A. D. George, and H. Lam. Reconfigurable computing middleware for application portability and productivity. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 211–218, June 2013. 19

[52] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020. 20

[53] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. Stratix 10 nx architecture and applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 57–67, New York, NY, USA, 2021. Association for Computing Machinery. 1, 14

[54] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The stratix™ 10 highly pipelined fpga architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 159–168, New York, NY, USA, 2016. Association for Computing Machinery. 43

[55] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. Aggressive pipelining of irregular applications on reconfigurable hardware. *SIGARCH Comput. Archit. News*, 45(2):575–586, June 2017. 1

[56] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 417–433, New York, NY, USA, 2020. Association for Computing Machinery. 22

[57] Gabriel H Loh, Nuwan Jayasena, M Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*, pages 1–4, 2013. 22

[58] Paolo Mantovani. *Scalable System-on-Chip Design*. PhD thesis, Columbia University, 2017. 18

[59] Joseph Melber and James Hoe. A Service-Oriented Memory Architecture for FPGA Computing. In *The International Conference on Field-Programmable Logic and Applications*, FPL '20, pages 91–97, 08 2020. 26

[60] Mentor Graphics. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls. 16

[61] Eric Micallef, Yuanlong Xiao, and Andre DeHon. Extending high-level synthesis for task-parallel programs. In *Proceedings of the 2021 IEEE International Symposium On Field-Programmable Custom Computing Machines*, FCCM 2021. IEEE, 2021. 16

[62] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, page 127–136, New York, NY, USA, 2007. Association for Computing Machinery. 17

[63] Timothy Prickett Morgan. The Inevitability of FPGAs in the Datacenter. https://www.nextplatform.com/2020/01/14/the-inevitability-of-fpgas-in-the-datacenter/, January 2020. 15

[64] MyHDL. The MyHDL Manual. http://docs.myhdl.org/en/stable/. 16

[65] G. Nordin, P.A. Milder, J.C. Hoe, and M. Puschel. Automatic generation of customized discrete fourier transform ips. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 471–474, 2005. 17

[66] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84, 2016. 1

[67] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 5–14, New York, NY, USA, 2017. Association for Computing Machinery. 1, 15

[68] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An FPGA framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28. IEEE, 2014. 17

[69] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85, Nov 2011. 1, 14

[70] Robert Palumbo, Hossein Saiedian, and Mansour Zand. The operational semantics of an active message system. In *Proceedings of the 1992 ACM Annual Conference on Communications*, CSC '92, pages 367–375, New York, NY, USA, 1992. ACM. 22

[71] Michael K. Papamichael. *Pandora: Facilitating IP Development for Hardware Specialization*. PhD thesis, Carnegie Mellon University, 2015. 3, 39

[72] Michael K Papamichael and James C Hoe. CONNECT: re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 37–46. ACM, 2012. 18, 94

[73] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. 17

[74] Michael Pellauer, Michael Adler, Derek Chiou, and Joel Emer. Soft connections: Addressing the hardware-design modularity problem. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 276–281, New York, NY, USA, 2009. ACM. 21

[75] Prabhat K. Gupta. Xeon+FPGA Platform for the Data Center. In *ISCA/CARL 2015*, Portland, OR, USA, 2015. 14

[76] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014. 1, 15

[77] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. *A Taxonomy, Survey, and Issues of Cloud Computing Ecosystems*, pages 21–46. Springer London, London, 2010. 23

[78] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Paul Chow, Ralph Wittig, Henry Styles, and Andrew Putnam. Mpi as a programming model for high-performance reconfigurable computers. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):22:1–22:29, November 2010. 21

[79] M. Saldana and P. Chow. Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, Aug 2006. 21

[80] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct universal access: Making data center resources available to FPGA. In

16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 127–140, Boston, MA, February 2019. USENIX Association. 21

[81] SNORT 3. Snort 3 Intrusion Prevention System. https://www.snort.org/snort3. 17, 125

[82] SpinalHDL. SpinalHDL User Guide. https://spinalhdl.github.io/SpinalDoc. 16

[83] Synopsys. Synphony C Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html. 16

[84] O. Terzo, P. Ruiu, E. Bucci, and F. Xhafa. Data as a service (daas) for sharing and processing of large data collections in the cloud. In *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 475–480, July 2013. 23

[85] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 63–72, New York, NY, USA, 2009. ACM. 1

[86] D. Tsichritzis, F. Rabitti, S. Gibbs, O. Nierstrasz, and J. Hogg. A system for managing structured messages. *IEEE Transactions on Communications*, 30(1):66–73, January 1982. 22

[87] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM. 22

[88] C. Wang, X. Li, Y. Chen, Y. Zhang, O. Diessel, and X. Zhou. Service-oriented architecture on fpga-based mpsoc. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2993–3006, Oct 2017. 23

[89] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. A survey of FPGA based deep learning accelerators: Challenges and opportunities. *CoRR*, abs/1901.04988, 2019. Withdrawn. 15

[90] Y. Wang, J. C. Hoe, and E. Nurvitadhi. Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–144, 4 2019. 11, 107

[91] G. Weisz and J. C. Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015. 20

[92] Ruediger Willenberg and Paul Chow. A Remote Memory Access Infrastructure for Global Address Space Programming Models in FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 211–220, New York, NY, USA, 2013. ACM. 21

[93] Xilinx. SDAccel. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. 16

[94] Xilinx. SDSoC. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html. 17

[95] Xilinx. Vivado HLx. https://www.xilinx.com/products/design-tools/vivado.html. 2, 18

[96] Xilinx. AXI Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, 2017. 18, 62

[97] Xilinx. Xilinx Annual Report (2020 Form 10-K). https://investor.xilinx.com/static-files/a7429604-5e98-4fae-8904-bd2c041eb476, May 2021. 15

[98] H. J. Yang, K. Fleming, M. Adler, and J. Emer. Leap shared memories: Automating the construction of fpga coherent memories. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 117–124, May 2014. 19

[99] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, New York, NY, USA, 2017. Association for Computing Machinery. 20

[100] S. Zhang, H. Angepat, and D. Chiou. Hgum: Messaging framework for hardware accelerators. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2017. 21

[101] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020. xvii, 11, 15, 17, 90, 125, 126

Appendix A

Source Code for the BFS Case Study

A.1 Python Service Catalog

```
class HostMsgSvc(Service):
    def __init__(self, name=None):
        super().__init__("csr2srv", SourceType.SYSTEM_VERILOG, name)
        self.add_client("memcpy")
        self.add_control("start", 1, Direction.INPUT)
        self.add_control("done", 1, Direction.OUTPUT)
        self.add_control("clear", 1, Direction.INPUT)
        self.add_control("destination", 64, Direction.INPUT)
        self.add_control("source", 64, Direction.INPUT)
        self.add_control("mc_num", 64, Direction.INPUT)

class MemCopy(Service):
    def __init__(self, name=None):
        super().__init__("mkMemCopyDualExternal",
        SourceType.BLUESPEC_SYSTEM_VERILOG, name)
        self.add_Service("cmdQ", 64, 512)
        self.add_client("readA")
        self.add_client("writeA")
        self.add_client("readB")
        self.add_client("writeB")
```

```
21
22 class BfsAfuPipeline(Service):
23     def __init__(self, name=None):
24         super().__init__("mkBFSafuPipelineExternal",
25         SourceType.BLUESPEC_SYSTEM_VERILOG, name)
26         self.add_client("worklist_chan")
27         self.add_client("neighbor_chan")
28         self.add_client("update_chan")
29         self.add_control("start_strt", 1, Direction.INPUT)
30         self.add_control("finish", 1, Direction.OUTPUT)
31         self.add_control("getNodesTchd", 64, Direction.OUTPUT)
32
33 class Worklist(Service):
34     def __init__(self, name=None):
35         super().__init__("mkWorklistExternal",
36         SourceType.BLUESPEC_SYSTEM_VERILOG, name)
37         self.add_Service("workQ", 32, 32,)
38         self.add_client("listRd")
39         self.add_client("listWr")
40         self.add_control("start", 1, Direction.INPUT)
41         self.add_control("setCapacity", 32, Direction.INPUT)
42
43 class Graph(Service):
44     def __init__(self, name=None):
45         super().__init__("mkGraphExternal",
46         SourceType.BLUESPEC_SYSTEM_VERILOG, name)
47         self.add_Service("neighborCmdQ", 32, 32)
48         self.add_Service("updateCmdQ", 32, 32)
49         self.add_client("readNodes")
50         self.add_client("readEdges")
51         self.add_client("readDistance")
52         self.add_client("writeDistance")
```

```
53
54 class ReadExternal32(Service):
55     def __init__(self, name=None):
56         super().__init__("mkReadExternal32",
57         SourceType.BLUESPEC_SYSTEM_VERILOG, name)
58         self.add_Service("cmdQ", 32, 512)
59         self.add_client("readQ")
60         self.add_control("offset", 64, Direction.INPUT)
61
62 class WriteExternal32(Service):
63     def __init__(self, name=None):
64         super().__init__("mkWriteExternal32",
65         SourceType.BLUESPEC_SYSTEM_VERILOG, name)
66         self.add_Service("cmdQ", 32, 512)
67         self.add_client("writeQ")
68         self.add_control("offset", 64, Direction.INPUT)
69
70 class Cache(Service):
71     def __init__(self, name=None):
72         super().__init__("direct_mapped_better",
73         SourceType.SYSTEM_VERILOG, name)
74         self.add_Service("read_channel_in", 32, 512)
75         self.add_Service("write_channel_in", 32, 512)
76         self.add_client("read_channel_out")
77         self.add_client("write_channel_out")
78         self.add_control("start_req", 1, Direction.INPUT)
79         self.add_control("stop_req", 1, Direction.INPUT)
80         self.add_control("start_ack", 1, Direction.OUTPUT)
81         self.add_control("stop_ack", 1, Direction.OUTPUT)
82         self.add_property("additional-files", ["fifo_fifo_191_mrvfezi.sv",
83                                                "pr_freeze_bridge.sv",
84                                                "ram2_ram_2port_2000_w4fp7wq.sv"])
```

```
85
86 class CacheBSV(Service):
87     def __init__(self, name=None):
88         super().__init__("mkReadWriteCacheExternal_32_64_1M",
89         SourceType.BLUESPEC_SYSTEM_VERILOG, name)
90         self.add_Service("readCache", 32, 512)
91         self.add_Service("writeCache", 32, 512)
92         self.add_client("read")
93         self.add_client("write")
94         self.add_control("setBase_addr", 64, Direction.INPUT)
```

A.2 Python System Design

```
1  import sys
2  import os
3
4  from header import *
5  from opaeplatform import *
6
7  USE_CACHE = False
8
9  registers = OPAERegisters("soma_csr", SourceType.SYSTEM_VERILOG)
10 platform = IntelOPAEPlatform(registers)
11 app = Application(platform, path_prefix="examples/bfs_avalon/")
12
13 afu = BfsAfuPipeline("bfs")
14 worklist = Worklist("worklist")
15 graph = Graph("graph")
16 rd_work = ReadExternal32("rd_work")
17 wr_work = WriteExternal32("wr_work")
18 rd_source = ReadExternal32("rd_source")
19 rd_edge = ReadExternal32("rd_edge")
20 rd_dist = ReadExternal32("rd_dist")
```

```
21 wr_dist = WriteExternal32("wr_dist")
22 if USE_CACHE:
23     AvalonCCIPService, avlccip_src = make_rw_service({platform.ccip:1,
24                                                       platform.avalon[0]:1})
25 else:
26     AvalonCCIPService, avlccip_src = make_rw_service({platform.ccip:1,
27                                                       platform.avalon[0]:1},
28                                                       lite=True)
29 rw = AvalonCCIPService()
30 mcp = MemCopy("mcp")
31 csr2ca = HostMsgSvc("csr2ca")
32
33 app.direct(csr2ca.clients["memcpy"], mcp.Services["cmdQ"])
34
35 app.direct(afu.clients["worklist_chan"], worklist.Services["workQ"])
36 app.direct(afu.clients["neighbor_chan"], graph.Services["neighborCmdQ"])
37 app.direct(afu.clients["update_chan"], graph.Services["updateCmdQ"])
38
39 app.direct(worklist.clients["listRd"], rd_work.Services["cmdQ"])
40 app.direct(worklist.clients["listWr"], wr_work.Services["cmdQ"])
41
42 app.direct(graph.clients["readNodes"], rd_source.Services["cmdQ"])
43 app.direct(graph.clients["readEdges"], rd_edge.Services["cmdQ"])
44 app.direct(graph.clients["readDistance"], rd_dist.Services["cmdQ"])
45 app.direct(graph.clients["writeDistance"], wr_dist.Services["cmdQ"])
46
47 app.noc(rd_work.clients["readQ"], rw.Services["read_avalon0"])
48 app.noc(wr_work.clients["writeQ"], rw.Services["write_avalon0"])
49 app.noc(rd_source.clients["readQ"], rw.Services["read_avalon0"])
50 app.noc(rd_edge.clients["readQ"], rw.Services["read_avalon0"])
51
52 if USE_CACHE:
```

```
53      cache = Cache("cache")
54      app.direct(cache.clients["read_channel_out"], rw.Services["read_avalon0"])
55      app.direct(cache.clients["write_channel_out"], rw.Services["write_avalon0"])
56      app.noc(rd_dist.clients["readQ"], cache.Services["read_channel_in"])
57      app.noc(wr_dist.clients["writeQ"], cache.Services["write_channel_in"])
58  else:
59      app.noc(rd_dist.clients["readQ"], rw.Services["read_avalon0"])
60      app.noc(wr_dist.clients["writeQ"], rw.Services["write_avalon0"])
61
62  app.noc(mcp.clients["readB"], rw.Services["read_avalon0"])
63  app.noc(mcp.clients["writeB"], rw.Services["write_avalon0"])
64  app.noc(mcp.clients["readA"], rw.Services["read_ccip"])
65  app.noc(mcp.clients["writeA"], rw.Services["write_ccip"])
66
67  pass_manager = PassManager(app)
68  pass_manager.load_data("bsv-files", {AvalonCCIPService.__name__:avlccip_src})
69  pass_manager.add_pass(OPAEAnalysisPass)
70  pass_manager.add_pass(OPAETopPass)
71  pass_manager.add_pass(OPAEBSVWrapperPass)
72  pass_manager.add_pass(OPAENOCWrapperPass)
73  pass_manager.add_pass(OPAEFinalPass)
74  pass_manager.schedule_and_run_passes()
```

A.3 Bluespec User-Level Kernel

```
1  package BFSafuPipeline;
2  import MessagePack::*;
3
4  interface BFSafuPipeline;
5      (* always_ready, always_enabled *) method Action start(Bool strt);
6      (* always_ready, always_enabled *) method Bool finish();
7      (* always_ready, always_enabled *) method Bit#(64) getNodesTchd();
8  endinterface
```

```
9
10 interface Control;
11         method Action start;
12         method Bool   idle;
13         method Action halt;
14 endinterface
15
16 //
17 // BFS application kernel pipeline.
18 //
19 // This elastic pipeline implementation has three simple stages as services
20 // that absorb complexity and simplify the user-level kernel logic.
21 //
22 // The design relys on three services:
23 //  - worklist service (operates like a queue with 'unlimited' capacity)
24 //  - node index to neighbors (traverses CSR graph returning a node's
25 //    neighbors and edge weights)
26 //  - atomic node update (updates a node's distance with proper locking
27 //    at the node granularity)
28 //
29 // The Bluespec code is compiled to Verilog before integration into the
30 // top level design.
31 //
32 module mkBFSafuPipeline#(
33               Service#(AM_FULL#(Bit#(32),Bit#(32)),
34                       AM_FULL#(Bit#(32),Bit#(32)),32) worklist_chan,
35               Service#(AM_FULL#(Bit#(32),Bit#(32)),
36                       AM_FULL#(Bit#(32),Bit#(32)),32) neighbor_chan,
37               Service#(AM_FULL#(Bit#(32),Bit#(32)),
38                       AM_FULL#(Bit#(32),Bit#(32)),32) update_chan)
39                  (BFSafuPipeline);
40
```

```
41   Reg#(Bit#(64)) nds_t <- mkReg(0);
42
43   Control stg1 <- mkFetchSourceNodeStage(worklist_chan.rxPort,
44                                          neighbor_chan.txPort);
45   Control stg2 <- mkUpdateNeighborStage(neighbor_chan.rxPort,
46                                         update_chan.txPort);
47   Control stg3 <- mkUpdateWorklistStage(update_chan.rxPort,
48                                         worklist_chan.txPort,32'd0,nds_t);
49
50   Reg#(Bool) start_in <- mkReg(False);
51   Reg#(Bool) fin <- mkReg(False);
52   Reg#(Bit#(64)) idleCnt <- mkReg(0);
53
54   rule starter;
55     if (start_in) begin
56       stg1.start();
57       stg2.start();
58       stg3.start();
59     end
60   endrule
61
62   rule ender;
63       if(stg2.idle&&stg3.idle) begin
64         idleCnt <= idleCnt + 1;
65       end else begin
66         idleCnt <= 0;
67       end
68       if (idleCnt>500) fin<=True;
69       else fin<=False;
70   endrule
71
72   method Action start(Bool strt);
```

```
73        start_in <= strt;
74    endmethod
75
76    method Bool finish();
77        return fin;
78    endmethod
79
80    method Bit#(64) getNodesTchd();
81        return nds_t;
82    endmethod
83  endmodule
84
85  //
86  // First pipeline stage.
87  //
88  // Simple combinational logic that issues requests to the "node index to
89  // neighbors" service whenever a new node is available for processing in
90  // the worklist queue.
91  //
92  module mkFetchSourceNodeStage#(
93                  RxMsgChannel#(AM_FULL#(sdarg,work)) wklist,
94                  TxMsgChannel#(AM_FULL#(sdarg,data)) pernode)
95              (Control) provisos(Bits#(sdarg,a_),
96                                 Bits#(data,c_),
97                                 Bits#(work,d_),
98                                 Literal#(sdarg),
99                                 Add#(a__, a_, d_));
100
101         rule fetch_src if ((!wklist.rxEmpty)&&(!pernode.txFull));
102                 sdarg argIdx = unpack(truncate(pack(wklist.rx.data.payload)));
103                 let hd = AM_HEAD { srcid:? , dstid:? , arg0:argIdx , arg1:?
104                                                      , arg2:?      , arg3:? };
```

```
105                         let req = AM_FULL { head: hd, data: ? };
106                         pernode.tx(req);
107
108                         wklist.rxPop;
109         endrule
110
111         method Action start;
112                 noAction;
113         endmethod
114
115         method Bool idle;
116                 return wklist.rxEmpty;
117         endmethod
118
119         method Action halt;
120                 noAction;
121         endmethod
122 endmodule
123
124 //
125 // Second pipeline stage.
126 //
127 // Combinational logic to update a neighbor node's distance with its
128 // parent node index if it has not been visited before. The atomic update
129 // service completes the update if arg2==True and handles locking
130 // of the neighbor's cacheline.
131 //
132 module mkUpdateNeighborStage#(
133                 RxMsgChannel#(AM_FULL#(sdarg,Bit#(32))) neighborRd,
134                 TxMsgChannel#(AM_FULL#(sdarg,Bit#(32))) neighborUd)
135             (Control) provisos(Bits#(sdarg,a_),
136                                Literal#(sdarg),
```

```
137                                 Arith#(sdarg),
138                                 Add#(a__,a_,32),
139                                 Add#(b__, 1, a_));
140
141         rule inner_loop if ((!neighborRd.rxEmpty) && (!neighborUd.txFull));
142                 let parent = neighborRd.rx.data.payload;
143                 let srcIdx = neighborRd.rx.head.arg0;
144                 Bool hasParent = parent!=32'hFFFFFFFF;
145                 Bit#(32) weight;
146                 if (hasParent) begin
147                         weight = parent;
148                 end else begin
149                         weight = extend(pack(srcIdx));
150                 end
151                 let hd = neighborRd.rx.head;
152                 hd.arg2 = unpack(extend(pack(hasParent)));
153                 let dd = AM_DATA { payload: weight };
154                 let rsp = AM_FULL { head: hd, data:dd };
155                 neighborUd.tx(rsp);
156                 neighborRd.rxPop;
157         endrule
158
159         method Action start;
160                 noAction;
161         endmethod
162
163         method Bool idle;
164                 return neighborRd.rxEmpty;
165         endmethod
166
167         method Action halt;
168                 noAction;
```

```
169         endmethod
170 endmodule
171
172 //
173 // Third pipeline stage.
174 //
175 // Sends a node index to the worklist for processing once it has been updated.
176 // As in the first stage the worklist consumes and provides work items like a
177 // queue at the work item granularity.
178 //
179 module mkUpdateWorklistStage#(
180                 RxMsgChannel#(AM_FULL#(sdarg,data)) nodepropW,
181                 TxMsgChannel#(AM_FULL#(sdarg,work)) wklist,
182                             Bit#(32) num_nodes, Reg#(Bit#(64)) numDone)
183             (Control) provisos(Bits#(sdarg,a_),
184                                Bits#(data,b_),
185                                Bits#(work,c_),
186                                Add#(a__,a_,c_));
187
188         Reg#(Bit#(32)) nodeCnt     <- mkReg(0);
189         Reg#(Bool) started <- mkReg(False);
190
191         rule consume_update_rsp if ((!nodepropW.rxEmpty));
192                 if ((!wklist.txFull)) begin
193                         let dd = AM_DATA { payload:
194                                  unpack(extend(pack(nodepropW.rx.head.arg2)))
195                                         };
196                         let req = AM_FULL { head: ?, data: dd };
197                         wklist.tx(req);
198                         nodepropW.rxPop;
199
200                         nodeCnt <= nodeCnt + 1;
```

```
201                         numDone <= extend(nodeCnt + 1);
202                 end
203         endrule
204
205         method Action start;
206                 started <= True;
207         endmethod
208
209         method Bool idle;
210                 return nodepropW.rxEmpty;
211         endmethod
212
213         method Action halt;
214                 noAction;
215         endmethod
216 endmodule
217
218 endpackage
```

Appendix B

Python Description for the Service-Oriented Pigasus Design

B.1 Python Service Catalog

```
class Reassembler(Service):
    def __init__(self, name):
        super().__init__("reassembler_service",
        SourceType.SYSTEM_VERILOG, name)
        self.add_extern_connection("Clk", "clk")
        self.add_extern_connection("Rst_n", "rst_n")

        self.add_extern_connection("pkt_buffer_writeaddress", "pkt_buf_wraddress")
        self.add_extern_connection("pkt_buffer_write", "pkt_buf_wren")
        self.add_extern_connection("pkt_buffer_writedata", "pkt_buf_wrdata")

        self.add_extern_connection("pkt_buffer_readaddress", "pkt_buf_rdaddress")
        self.add_extern_connection("pkt_buffer_read", "pkt_buf_rden")
        self.add_extern_connection("pkt_buffer_readvalid", "pkt_buf_rd_valid")
        self.add_extern_connection("pkt_buffer_readdata", "pkt_buf_rddata")

        self.add_control("parser_meta_csr_readdata", 32, Direction.OUTPUT)
        self.add_control("stats_incomp_out_meta", 32, Direction.OUTPUT)
```

```
19          self.add_control("stats_parser_out_meta", 32, Direction.OUTPUT)
20          self.add_control("stats_ft_in_meta", 32, Direction.OUTPUT)
21          self.add_control("stats_ft_out_meta", 32, Direction.OUTPUT)
22          self.add_control("stats_emptylist_in", 32, Direction.OUTPUT)
23          self.add_control("stats_emptylist_out", 32, Direction.OUTPUT)
24          self.add_control("stats_dm_in_meta", 32, Direction.OUTPUT)
25          self.add_control("stats_dm_out_meta", 32, Direction.OUTPUT)
26          self.add_control("stats_dm_in_forward_meta", 32, Direction.OUTPUT)
27          self.add_control("stats_dm_in_drop_meta", 32, Direction.OUTPUT)
28          self.add_control("stats_dm_in_check_meta", 32, Direction.OUTPUT)
29          self.add_control("stats_dm_in_ooo_meta", 32, Direction.OUTPUT)
30          self.add_control("stats_dm_in_forward_ooo_meta", 32, Direction.OUTPUT)
31          self.add_control("stats_nopayload_pkt", 32, Direction.OUTPUT)
32          self.add_control("stats_dm_check_pkt", 32, Direction.OUTPUT)
33
34          self.add_provides("eth", 32, 512, channel_type="Ethernet")
35          self.add_requires("nopayload", channel_type="Nopayload")
36          self.add_requires("out_pkt", channel_type="Packet")
37          self.add_requires("out_meta", channel_type="Meta")
38          self.add_requires("out_usr", channel_type="Usr")
39
40  count = 0
41
42  def make_channel_fifo(clk_i, rst_n_i, counter, dual_clock=False,
43                        clk_o=None, rst_n_o=None):
44      class CF(Service):
45          def __init__(self, name):
46              super().__init__("channel_fifo_service",
47              SourceType.SYSTEM_VERILOG, name)
48              if dual_clock == False:
49                  self.add_extern_connection("Clk_i", clk_i)
50                  self.add_extern_connection("Rst_n_i", rst_n_i)
```

```
51                 self.add_property("parameters", {"DUAL_CLOCK":"0"})
52             else:
53                 self.add_extern_connection("Clk_i", clk_i)
54                 self.add_extern_connection("Rst_n_i", rst_n_i)
55                 self.add_extern_connection("Clk_o", clk_o)
56                 self.add_extern_connection("Rst_n_o", rst_n_o)
57                 self.add_property("parameters", {"DUAL_CLOCK":"1"})
58
59             self.add_control("in_pkt_fill_level", 32, Direction.OUTPUT)
60
61             self.add_provides("in_pkt", 32, 512, channel_type="Packet")
62             self.add_provides("in_meta", 32, 512, channel_type="Meta")
63             self.add_provides("in_usr", 32, 512, channel_type="Usr")
64
65             self.add_requires("out_pkt", channel_type="Packet")
66             self.add_requires("out_meta", channel_type="Meta")
67             self.add_requires("out_usr", channel_type="Usr")
68
69             self.add_control("stats_in_pkt", 32, Direction.OUTPUT);
70             self.add_control("stats_in_pkt_sop", 32, Direction.OUTPUT);
71             self.add_control("stats_in_meta", 32, Direction.OUTPUT);
72             self.add_control("stats_in_rule", 32, Direction.OUTPUT);
73
74             self.add_bypass(self.requires["out_pkt"], self.provides["in_pkt"])
75             self.add_bypass(self.requires["out_meta"], self.provides["in_meta"])
76             self.add_bypass(self.requires["out_usr"], self.provides["in_usr"])
77
78             self.add_property("hide-visualization", "bold")
79     global count
80     CF.__name__ = "CF_" + str(count)
81     CF.__qualname__ = "CF_" + str(count)
82     count += 1
```

```
83      return CF
84
85  def connect_channel_fifo(app, fifo_class, name, source, sink):
86      fifo = fifo_class(name)
87      app.direct(source.requires["out_pkt"], fifo.provides["in_pkt"])
88      app.direct(source.requires["out_meta"], fifo.provides["in_meta"])
89      app.direct(source.requires["out_usr"], fifo.provides["in_usr"])
90
91      app.direct(fifo.requires["out_pkt"], sink.provides["in_pkt"])
92      app.direct(fifo.requires["out_meta"], sink.provides["in_meta"])
93      app.direct(fifo.requires["out_usr"], sink.provides["in_usr"])
94
95  class FastPatternMatcher(Service):
96      def __init__(self, name):
97          super().__init__("fast_pm_service",
98          SourceType.SYSTEM_VERILOG, name)
99          self.add_extern_connection("Clk", "clk")
100         self.add_extern_connection("Rst_n", "rst_n")
101         self.add_extern_connection("Clk_front", "clk_high")
102         self.add_extern_connection("Rst_front_n", "rst_n")
103         self.add_extern_connection("Clk_back", "clk_pcie")
104         self.add_extern_connection("Rst_back_n", "rst_n_pcie")
105
106         self.add_control("sm_bypass_af", 32, Direction.OUTPUT)
107         self.add_control("sm_cdc_af", 32, Direction.OUTPUT)
108
109         self.add_control("stats_out_pkt", 32, Direction.OUTPUT)
110         self.add_control("stats_out_meta", 32, Direction.OUTPUT)
111         self.add_control("stats_out_rule", 32, Direction.OUTPUT)
112         self.add_control("stats_nocheck_pkt", 32, Direction.OUTPUT)
113         self.add_control("stats_check_pkt", 32, Direction.OUTPUT)
114         self.add_control("stats_check_pkt_s", 32, Direction.OUTPUT)
```

```
115
116         self.add_provides("in_pkt", 32, 512, channel_type="Packet")
117         self.add_provides("in_meta", 32, 512, channel_type="Meta")
118         self.add_provides("in_usr", 32, 512, channel_type="Usr")
119         self.add_requires("fp_nocheck", channel_type="FPNocheck")
120         self.add_requires("out_pkt", channel_type="Packet")
121         self.add_requires("out_meta", channel_type="Meta")
122         self.add_requires("out_usr", channel_type="Usr")
123
124 class PortGroupMatcher(Service):
125     def __init__(self, name):
126         super().__init__("port_group_matcher_service",
127         SourceType.SYSTEM_VERILOG, name)
128         self.add_extern_connection("Clk", "clk_pcie")
129         self.add_extern_connection("Rst_n", "rst_n_pcie")
130
131         self.add_control("stats_out_pkt", 32, Direction.OUTPUT)
132         self.add_control("stats_out_meta", 32, Direction.OUTPUT)
133         self.add_control("stats_out_rule", 32, Direction.OUTPUT)
134         self.add_control("stats_nocheck_pkt", 32, Direction.OUTPUT)
135         self.add_control("stats_check_pkt", 32, Direction.OUTPUT)
136         self.add_control("stats_check_pkt_s", 32, Direction.OUTPUT)
137         self.add_control("pg_no_pg_rule_cnt", 32, Direction.OUTPUT)
138         self.add_control("pg_int_rule_cnt", 32, Direction.OUTPUT)
139
140         self.add_provides("in_pkt", 32, 512, channel_type="Packet")
141         self.add_provides("in_meta", 32, 512, channel_type="Meta")
142         self.add_provides("in_usr", 32, 512, channel_type="Usr")
143         self.add_requires("pg_nocheck", channel_type="PGNocheck")
144         self.add_requires("out_pkt", channel_type="Packet")
145         self.add_requires("out_meta", channel_type="Meta")
146         self.add_requires("out_usr", channel_type="Usr")
```

```
147
148 class NonFastPatternMatcher(Service):
149     def __init__(self, name):
150         super().__init__("non_fast_pm_service",
151         SourceType.SYSTEM_VERILOG, name)
152         self.add_extern_connection("Clk", "clk_pcie")
153         self.add_extern_connection("Rst_n", "rst_n_pcie")
154         self.add_extern_connection("Clk_high", "clk_high")
155         self.add_extern_connection("Rst_high_n", "rst_n_high")
156
157         self.add_control("stats_out_pkt", 32, Direction.OUTPUT)
158         self.add_control("stats_out_meta", 32, Direction.OUTPUT)
159         self.add_control("stats_out_rule", 32, Direction.OUTPUT)
160         self.add_control("stats_nocheck_pkt", 32, Direction.OUTPUT)
161         self.add_control("stats_check_pkt", 32, Direction.OUTPUT)
162         self.add_control("stats_check_pkt_s", 32, Direction.OUTPUT)
163         self.add_control("stats_bypass_pkt", 32, Direction.OUTPUT)
164         self.add_control("stats_bypass_pkt_s", 32, Direction.OUTPUT)
165         self.add_control("stats_bypass_meta", 32, Direction.OUTPUT)
166         self.add_control("stats_bypass_rule", 32, Direction.OUTPUT)
167
168         self.add_control("bypass_fill_level", 32, Direction.OUTPUT)
169         self.add_control("bypass2nf_fill_level", 32, Direction.OUTPUT)
170         self.add_control("nf2bypass_fill_level", 32, Direction.OUTPUT)
171         self.add_control("nf_max_raw_pkt_fifo", 32, Direction.OUTPUT)
172         self.add_control("nf_max_pkt_fifo", 32, Direction.OUTPUT)
173         self.add_control("nf_max_rule_fifo", 32, Direction.OUTPUT)
174
175         self.add_provides("in_pkt", 32, 512)
176         self.add_provides("in_meta", 32, 512)
177         self.add_provides("in_usr", 32, 512)
178         self.add_requires("nfp_nocheck")
```

```
179         self.add_requires("out_pkt")
180         self.add_requires("out_meta")
181         self.add_requires("out_usr")
182
183 def make_unified_fifo(clk_i, rst_n_i, uf_name, fill_level, dual_clock=False,
184                       clk_o=None, rst_n_o=None):
185     class UF(Service):
186         def __init__(self, name):
187             super().__init__("unified_pkt_fifo_service",
188             SourceType.SYSTEM_VERILOG, name)
189             if dual_clock == False:
190                 self.add_extern_connection("Clk_i", clk_i)
191                 self.add_extern_connection("Rst_n_i", rst_n_i)
192                 self.add_property("parameters", {"FIFO_NAME":'"' + uf_name + '"',
193                                                  "MEM_TYPE":'"M20K"',
194                                                  "DUAL_CLOCK":"0",
195                                                  "USE_ALMOST_FULL":"1",
196                                                  "FULL_LEVEL":"450",
197                                                  "SYMBOLS_PER_BEAT":"64",
198                                                  "BITS_PER_SYMBOL":"8",
199                                                  "FIFO_DEPTH":"512"})
200             else:
201                 self.add_extern_connection("Clk_i", clk_i)
202                 self.add_extern_connection("Rst_n_i", rst_n_i)
203                 self.add_extern_connection("Clk_o", clk_o)
204                 self.add_extern_connection("Rst_n_o", rst_n_o)
205                 self.add_property("parameters", {"FIFO_NAME":'"' + uf_name + '"',
206                                                  "MEM_TYPE":'"M20K"',
207                                                  "DUAL_CLOCK":"1",
208                                                  "USE_ALMOST_FULL":"1",
209                                                  "FULL_LEVEL":"450",
210                                                  "SYMBOLS_PER_BEAT":"64",
```

```
211                                                 "BITS_PER_SYMBOL":"8",
212                                                 "FIFO_DEPTH":"512"})
213
214             self.add_provides("in", 32, 512)
215             self.add_requires("out")
216             self.add_extern_connection("fill_level", fill_level)
217             self.add_bypass(self.requires["out"], self.provides["in"])
218             self.add_property("hide-visualization", "bold")
219     global count
220     UF.__name__ = "UF_" + str(count)
221     UF.__qualname__ = "UF_" + str(count)
222     return UF
223
224 class DMA(Service):
225     def __init__(self, name):
226         super().__init__("dma_service",
227         SourceType.SYSTEM_VERILOG, name)
228         self.add_extern_connection("Clk", "clk_pcie")
229         self.add_extern_connection("Rst_n", "rst_n_pcie")
230         self.add_extern_connection("pcie_rb_wr_data", "pcie_rb_wr_data")
231         self.add_extern_connection("pcie_rb_wr_addr", "pcie_rb_wr_addr")
232         self.add_extern_connection("pcie_rb_wr_en", "pcie_rb_wr_en")
233         self.add_extern_connection("pcie_rb_wr_base_addr",
234                                    "pcie_rb_wr_base_addr")
235         self.add_extern_connection("pcie_rb_almost_full", "pcie_rb_almost_full")
236         self.add_extern_connection("pcie_rb_update_valid",
237                                    "internal_rb_update_valid")
238         self.add_extern_connection("pcie_rb_update_size", "pcie_rb_update_size")
239         self.add_extern_connection("disable_pcie", "disable_pcie")
240         self.add_extern_connection("pdumeta_cpu_data", "pdumeta_cpu_data")
241         self.add_extern_connection("pdumeta_cpu_valid", "pdumeta_cpu_valid")
242         self.add_extern_connection("pdumeta_cpu_ready",
```

```
243                                     "pdumeta_cpu_ready")
244         self.add_extern_connection("pdumeta_cpu_csr_readdata",
245                                     "pdumeta_cpu_csr_readdata")
246         self.add_extern_connection("ddr_wr_req_data", "ddr_wr_req_data")
247         self.add_extern_connection("ddr_wr_req_valid", "ddr_wr_req_valid")
248         self.add_extern_connection("ddr_wr_req_almost_full",
249                                     "ddr_wr_req_almost_full")
250         self.add_extern_connection("ddr_rd_req_data", "ddr_rd_req_data")
251         self.add_extern_connection("ddr_rd_req_valid", "ddr_rd_req_valid")
252         self.add_extern_connection("ddr_rd_req_almost_full",
253                                     "ddr_rd_req_almost_full")
254         self.add_extern_connection("ddr_rd_resp_data", "ddr_rd_resp_out_data")
255         self.add_extern_connection("ddr_rd_resp_valid", "ddr_rd_resp_out_valid")
256         self.add_extern_connection("ddr_rd_resp_almost_full",
257                                     "ddr_rd_resp_out_ready")
258         self.add_provides("in_pkt", 32, 512)
259         self.add_provides("in_meta", 32, 512)
260         self.add_provides("in_usr", 32, 512)
261         self.add_requires("nomatch_pkt")
262
263 class Ethernet(Service):
264     def __init__(self, name):
265         super().__init__("ethernet_service_multi_out",
266         SourceType.SYSTEM_VERILOG, name)
267         self.add_extern_connection("Clk", "clk")
268         self.add_extern_connection("Rst_n", "rst_n")
269
270         self.add_extern_connection("out_data", "out_data")
271         self.add_extern_connection("out_valid", "out_valid")
272         self.add_extern_connection("out_ready", "out_ready")
273         self.add_extern_connection("out_startofpacket", "out_sop")
274         self.add_extern_connection("out_endofpacket", "out_eop")
```

```
275         self.add_extern_connection("out_empty", "out_empty")
276         self.add_extern_connection("in_sop", "in_sop")
277         self.add_extern_connection("in_eop", "in_eop")
278         self.add_extern_connection("in_data", "in_data")
279         self.add_extern_connection("in_empty", "in_empty")
280         self.add_extern_connection("in_valid", "in_valid")
281
282         self.add_requires("in")
283         self.add_provides("out0", 32, 512)
284         self.add_provides("out1", 32, 512)
285         self.add_provides("out2", 32, 512)
286         self.add_provides("out3", 32, 512)
287         self.add_provides("out4", 32, 512)
```

B.2 Python System Design

```
1  import sys
2  import os
3
4  from header import *
5  from vtl import *
6  from genericplatform import *
7  from visualizer import *
8
9  registers = GenericRegisters("soma_csr", SourceType.SYSTEM_VERILOG)
10 platform = GenericPlatform("clk", "rst_n", registers)
11 app = Application(platform, path_prefix="examples/pigasus/",
12                   top_name="top", top_includes=["./src/struct_s.sv",
13                   "./src/stats_reg.sv"], paste_files=["top_base.sv"],
14                   hide_controls=True)
15
16 #externs, generated by parser.py
17 app.add_extern("clk", 1, Direction.INPUT)
```

```
18 app.add_extern("rst", 1, Direction.INPUT)
19 app.add_extern("clk_high", 1, Direction.INPUT)
20 app.add_extern("rst_high", 1, Direction.INPUT)
21 app.add_extern("clk_pcie", 1, Direction.INPUT)
22 app.add_extern("rst_pcie", 1, Direction.INPUT)
23 app.add_extern("in_sop", 1, Direction.INPUT)
24 app.add_extern("in_eop", 1, Direction.INPUT)
25 app.add_extern("in_data", 512, Direction.INPUT)
26 app.add_extern("in_empty", 6, Direction.INPUT)
27 app.add_extern("in_valid", 1, Direction.INPUT)
28 app.add_extern("out_data", 512, Direction.OUTPUT)
29 app.add_extern("out_valid", 1, Direction.OUTPUT)
30 app.add_extern("out_sop", 1, Direction.OUTPUT)
31 app.add_extern("out_eop", 1, Direction.OUTPUT)
32 app.add_extern("out_empty", 6, Direction.OUTPUT)
33 app.add_extern("out_ready", 1, Direction.INPUT)
34 app.add_extern("pkt_buf_wren", 1, Direction.OUTPUT)
35 app.add_extern("pkt_buf_wraddress", 17, Direction.OUTPUT)
36 app.add_extern("pkt_buf_rdaddress", 17, Direction.OUTPUT)
37 app.add_extern("pkt_buf_wrdata", 520, Direction.OUTPUT)
38 app.add_extern("pkt_buf_rden", 1, Direction.OUTPUT)
39 app.add_extern("pkt_buf_rd_valid", 1, Direction.INPUT)
40 app.add_extern("pkt_buf_rddata", 520, Direction.INPUT)
41 app.add_extern("pcie_rb_wr_data", 514, Direction.OUTPUT)
42 app.add_extern("pcie_rb_wr_addr", 12, Direction.OUTPUT)
43 app.add_extern("pcie_rb_wr_en", 1, Direction.OUTPUT)
44 app.add_extern("pcie_rb_wr_base_addr", 12, Direction.INPUT)
45 app.add_extern("pcie_rb_almost_full", 1, Direction.INPUT)
46 app.add_extern("pcie_rb_update_valid", 1, Direction.OUTPUT)
47 app.add_extern("pcie_rb_update_size", 12, Direction.OUTPUT)
48 app.add_extern("disable_pcie", 1, Direction.INPUT)
49 app.add_extern("pdumeta_cpu_data", 28, Direction.INPUT)
```

```
50 app.add_extern("pdumeta_cpu_valid", 1, Direction.INPUT)
51 app.add_extern("pdumeta_cnt", 10, Direction.OUTPUT)
52 app.add_extern("ddr_wr_req_data", 541, Direction.OUTPUT)
53 app.add_extern("ddr_wr_req_valid", 1, Direction.OUTPUT)
54 app.add_extern("ddr_wr_req_almost_full", 1, Direction.INPUT)
55 app.add_extern("ddr_rd_req_data", 29, Direction.OUTPUT)
56 app.add_extern("ddr_rd_req_valid", 1, Direction.OUTPUT)
57 app.add_extern("ddr_rd_req_almost_full", 1, Direction.INPUT)
58 app.add_extern("ddr_rd_resp_data", 512, Direction.INPUT)
59 app.add_extern("ddr_rd_resp_valid", 1, Direction.INPUT)
60 app.add_extern("ddr_rd_resp_almost_full", 1, Direction.OUTPUT)
61 app.add_extern("clk_status", 1, Direction.INPUT)
62 app.add_extern("status_addr", 30, Direction.INPUT)
63 app.add_extern("status_read", 1, Direction.INPUT)
64 app.add_extern("status_write", 1, Direction.INPUT)
65 app.add_extern("status_writedata", 32, Direction.INPUT)
66 app.add_extern("status_readdata", 32, Direction.OUTPUT)
67 app.add_extern("status_readdata_valid", 1, Direction.OUTPUT)
68
69 #### SERVICES ####
70
71 reassembler = Reassembler("r")
72 dm2sm_cf = make_channel_fifo("clk", "rst_n", "dm2sm_fill_level")
73 fpm = FastPatternMatcher("fpm")
74 sm2pg_cf = make_channel_fifo("clk_pcie", "rst_n_pcie",
75                              "sm2pg_fill_level", dual_clock=True,
76                              clk_o="clk_pcie", rst_n_o="rst_n_pcie")
77 pgm = PortGroupMatcher("pg")
78 pg2nf_cf = make_channel_fifo("clk_pcie", "rst_n_pcie",
79                              "pg2nf_fill_level")
80 nfp = NonFastPatternMatcher("nf")
81 bypassback2pdu_cf = make_channel_fifo("clk_pcie", "rst_n_pcie",
```

```
82                                          "nf2pdu_fill_level")
83 dma = DMA("dma")
84 nopayload  = make_unified_fifo("clk", "rst_n", "[top] fifo0",
85                                "dm_nopayload_pkt_csr_readdata")("fifo0")
86 sm_nocheck = make_unified_fifo("clk_pcie", "rst_n_pcie", "[top] fifo1",
87                                "sm_nocheck_pkt_csr_readdata",
88                                dual_clock=True, clk_o="clk",
89                                rst_n_o="rst_n")("fifo1")
90 pg_nocheck = make_unified_fifo("clk_pcie", "rst_n_pcie", "[top] fifo2",
91                                "pg_nocheck_pkt_csr_readdata",
92                                dual_clock=True, clk_o="clk",
93                                rst_n_o="rst_n")("fifo2")
94 nf_nocheck = make_unified_fifo("clk_pcie", "rst_n_pcie", "[top] fifo3",
95                                "nf_nocheck_pkt_csr_readdata",
96                                dual_clock=True, clk_o="clk",
97                                rst_n_o="rst_n")("fifo3")
98 nomatch    = make_unified_fifo("clk_pcie", "rst_n_pcie", "[top] fifo4",
99                                "nomatch_pkt_csr_readdata", dual_clock=True,
100                               clk_o="clk", rst_n_o="rst_n")("fifo4")
101 eth = Ethernet("ethernet")
102
103 #### CONNECTIONS ####
104
105 app.direct(eth.requires["in"], reassembler.provides["eth"])
106 app.direct(nopayload.requires["out"], eth.provides["out0"])
107 app.direct(nf_nocheck.requires["out"], eth.provides["out1"])
108 app.direct(nomatch.requires["out"], eth.provides["out2"])
109 app.direct(sm_nocheck.requires["out"], eth.provides["out3"])
110
111 # Example: to remove pgm, comment this line
112 app.direct(pg_nocheck.requires["out"], eth.provides["out4"])
113
```

```
114 connect_channel_fifo(app, dm2sm_cf, "dm2sm", reassembler, fpm)
115
116 # Example: to remove pgm, uncomment this line
117 #connect_channel_fifo(app, sm2pg_cf, "sm2pg", fpm, nfp)
118
119 # Example: to remove pgm, comment these two lines
120 connect_channel_fifo(app, sm2pg_cf, "sm2pg", fpm, pgm)
121 connect_channel_fifo(app, pg2nf_cf, "pg2nf", pgm, nfp)
122
123 connect_channel_fifo(app, bypassback2pdu_cf, "by2pd", nfp, dma)
124
125 app.direct(reassembler.requires["nopayload"], nopayload.provides["in"])
126 app.direct(fpm.requires["fp_nocheck"], sm_nocheck.provides["in"])
127
128 # Example: to remove pgm, comment this line
129 app.direct(pgm.requires["pg_nocheck"], pg_nocheck.provides["in"])
130
131 app.direct(nfp.requires["nfp_nocheck"], nf_nocheck.provides["in"])
132 app.direct(dma.requires["nomatch_pkt"], nomatch.provides["in"])
133
134 #### Generator Passes ####
135
136 pass_manager = PassManager(app)
137 pass_manager.add_pass(GenericAnalysisPass)
138 pass_manager.add_pass(GenericTopPass)
139 pass_manager.add_pass(generate_visualizer_pass(VisualizerType.DEFAULT))
140 pass_manager.schedule_and_run_passes()
```