Accelerating Input-Dependent Streaming Applications on FPGAs

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Shashank Obla

B.Tech, Electrical Engineering, Indian Institute of Technology, Bombay

M.Tech, Electrical Engineering, Indian Institute of Technology, Bombay

Carnegie Mellon University

Pittsburgh, PA

August 2026

Acknowledgments

Thesis Committee:

- Prof. James C. Hoe (Chair)
- Prof. Akshitha Sriraman
- Prof. Kevin Skadron
- Dr. Bin Li

Every PhD journey is unique, defined not only by the work but also by the people you meet. I would like to acknowledge those who had an impact on mine: celebrating the good times, guiding me along the way, and supporting me when things didn't work out as expected.

James, my advisor, has been a constant source of support throughout my time here. His advice and guidance in the research process, especially in formulating research questions and presenting my work, have been invaluable. I couldn't have asked for a kinder or more considerate advisor. He was understanding when life got in the way, yet he pushed me when necessary. The opportunities and flexibility he provided enriched my PhD experience immensely, from letting me explore my own ideas and mentor interns, to having unfiltered conversations and remaining completely approachable even while on sabbatical.

My thesis committee members have gone above and beyond their roles, guiding me for years even before my thesis came together and serving as close collaborators on various parts of my dissertation. I have had the pleasure of knowing

Akshitha since she joined the faculty at CMU, and I have thoroughly enjoyed all our interactions, from discussing research to sharing advice about networking and academia. I also deeply appreciate the trust she placed in me, both by having me meet with her prospective students when she was forming her research group, and by giving me the opportunity to chair the student council. *Bin* has been a constant and close collaborator since the very early years of the RapidQ project, sticking with us through thick and thin. Her insights from a computer systems perspective were crucial in steering the framework into its final form, and her feedback on the various iterations of the RapidQ paper was extremely helpful in improving its clarity and readability. Ever since I first joined the RapidDetect project, I have been constantly amazed by *Kevin*'s sharp insights and deep expertise across domains from PIM to FPGAs and automata processing. His thoughtful consideration of everyone's input and full engagement in our weekly meetings, despite his incredibly busy schedule, have been a tremendous source of motivation. Finally, I am very grateful for the collective feedback from my committee during my proposal, which profoundly shaped my dissertation, and for the engaging discussions during my defense.

I have had the privilege of working with great collaborators whose insights and contributions were essential to this dissertation. I would like to thank *Tommy Tracy II (Tom)* for being the beta tester of the FPGA designs in RapidDetect, and for working alongside me to debug the software and meet paper deadlines. I would also like to acknowledge the rest of the RapidDetect team, *Matthew Beck* and *Wajih Ul Hassan*, for their contributions and discussions during our meetings. *Isha Garg* and *Xiying Deng*, who were interns in our research group, were an integral part of the RapidQ project during its early stages. They tackled very open-ended problem statements as we worked to define the core research questions. Although we did not directly collaborate, the interactions I had with *Andrew Boutros* and *Vaughn*

Betz through the Crossroads project taught me a great deal about FPGAs and being a researcher. Meeting them and their groups at FCCM earlier this year was one of the highlights of my trip, and I truly appreciate them for being so approachable, generously letting me pick their brains, and engaging in such open conversations. Furthermore, the Crossroads and Pigasus projects provided the invaluable opportunity to interact with brilliant minds, *Justine Sherry*, *Hugo Sadok*, *Nirav Atre*, and *Derek Chiou*, whose insights profoundly shaped my view of systems research. A special shoutout to *Scott Weber*, who was also my mentor during my internship at Intel PSG (now Altera). He has been with us since the beginning of the Crossroads project, providing feedback and guidance from an industry perspective and, more importantly, always being so kind. I am incredibly excited to soon be his colleague at Altera and to keep the collaboration going!

I would like to thank the members of the James Hoe research group. I am grateful to my seniors: *Zhipeng*, *Joe*, *Marie*, and *Guanglin*. Even though our in-person overlap was cut short by COVID, they helped initiate me into the group, got me caught up on the state-of-the-art, and patiently answered all my questions. I ended up becoming the senior-most PhD student in the group quite early in my third year. Because of this, the students who came after me, *Eric*, *Siddharth*, *Chengyue*, *Jason*, *Will*, *Simon*, *Edwin*, and *Sanjay*, became wonderful friends, making the time we shared in the office and group meetings feel much less formal and stressful. Over the years, we also had the pleasure of working with many interns: *Yang*, *Ni*, *Plava*, *Akshay*, *Isha*, *Xiying*, *Yuhe*, *Michelle*, *Haoyang*, *Jiajun*, and *Rock*. Working with them was a rewarding experience, and I want to thank them for their patience as I navigated the nuances of mentorship. I would also like to acknowledge the other A-levelers, *Upasana*, *Larry* (honorary), *Het*, *Elliott*, *Naifeng*, *Sanil*, *Ishank*, *Quentin*, *Tianyun*, *Oğuzhan*, *Yufei*, *Maia*, and *Anuva*, for creating such a welcoming environment. Thank you to the ECE department staff, particularly *Nathan*, *Greta*,

Kimmy, *Jessica*, and all the admins, for their assistance and for organizing fun, department-wide activities like ECE Day and the open house. Finally, I want to thank EGO (the ECE Graduate Organization) for providing a space for all of us in the department to get together.

Among the many courses I took at CMU, two stood out as having a significant impact on my research and growth. Taking a math-heavy course on queueing theory was daunting as a computer architect, but *Prof. Mor Harchol-Balter* made it incredibly fun. I deeply appreciated the enthusiasm she brought to the classroom, as well as her ability to always provide relatable examples and strong intuition for complex mathematical proofs. Her course is what got me excited about applying queueing theory to FPGA designs, ultimately laying the foundation for RapidQ and this dissertation. Serving as a TA for her new course, which helps systems PhD students map their problems to queueing theory using intuition rather than rigorous math, was truly rewarding. That experience was instrumental in honing my ability to bridge disparate concepts to engineer novel solutions. The second was *Prof. Tze Meng Low*'s course on "How to Write Fast Code", which served as an eye-opening introduction to systematically thinking about performance on modern processors. His class beautifully bridged my understanding of performance across custom hardware, FPGAs, and traditional processors. Beyond the classroom, I had the pleasure of interacting with him regularly as a fellow A-leveler. I am deeply grateful for his informal mentorship, as he constantly challenged me to rethink my assumptions and rigorously clarify my arguments.

I have been extremely fortunate to make some good friends along the way who kept me sane through my PhD. From Friday (or really, any day) dinners and rides in the Tangmobile to random hangs at Solway and the office, the Hooligans, *Eric*, *Larry*, and *Jason*, have been a constant anchor for most of my time here in Pittsburgh. *Upasana*, the cult leader, is the reason I picked up so many new hobbies,

from biking to yarn work, and I loved having someone I could talk to about research, teaching, and random topics, no matter how edgy. I truly value *Elliott* for being a wonderfully predictable friend amidst the chaos. *Het* was only at CMU for a short time, but as one of my first friends right after COVID, I always felt supported by her caring nature. Hanging out at *Chris* and *Jen*'s right down the street was like being with family who genuinely cared about all of us. We had some great times cooking together, playing Switch games, and just eating all the incredible food Chris whipped up for us! While I have known *Siddharth* since he joined our research group during COVID, we really bonded over games of AoE, and I always appreciated having someone I could spontaneously hit up for a game or a meal. *Tuhin* was my flatmate and likely my very first friend in Pittsburgh, and I enjoyed navigating my transition to the states and weathering the COVID lockdowns at home with him. Together with *Amritesh*, we shared wonderful times biking, going on hikes, and simply being there for one another. I am also so grateful to *Naifeng*, *Icey*, *Pragna*, *Deepanjali*, and *Miguel* for sharing in the movie nights, craft picnics, and house parties I always looked forward to. Special thanks to *Naifeng* for trusting me with Ray-Ray; no one can ever top that birthday present! Even across time and distance, even though I have been terrible at keeping in touch, I am lucky to have lasting friendships from my school and undergrad days. Reaching out to them always feels effortless, no matter how long we have been apart.

Lastly, I would not be here without the unwavering support and motivation of my mom and dad. I owe so much to the sacrifices they made to help me succeed and their constant investment in my future. They never allowed the distance to make me feel far from home, whether they were patiently listening to me vent daily about broken code, or just quietly sitting on a call with me while I worked. I was also extremely fortunate that they traveled to visit me multiple times when I could not make the trip back, allowing me to stay focused on my work. And finally, to

Ray-Ray, my furry companion[1] and best friend: thank you for always being by my side. I can't imagine these past three years without him. Between staying awake with me while I worked late, melting my stress away by being so affectionate, and always being thrilled to see me, he is the best companion I could have asked for.

This work was supported in part by CIT Dean's Fellowship, SRC/JUMP (CONIX), Intel/VMware Crossroads 3D-FPGA Academic Research Center, CyLab's Future Enterprise Security Initiative, and generous hardware donations from AMD, through the Heterogeneous Accelerated Compute Clusters (HACC) Program, and Intel.

Shashank Obla

Carnegie Mellon University

August 2026

[1] he's a cat for those wondering

Abstract

The limited flexibility of domain-specific accelerators renders them inefficient for the important class of streaming applications, such as in security and database analytics, whose optimal resource provisioning is workload dependent. Because provisioning hardware for the worst-case can be prohibitively expensive, these applications are often relegated to software execution. Field-Programmable Gate Arrays (FPGAs) are uniquely positioned to accelerate such input-dependent streaming applications, combining inherent programmability with hardware-class performance and efficiency. However, their adoption for customized accelerator deployments is hampered by the slow and rigid hardware-centric toolchains. At design time, RTL-based implementations offer limited customizability, while High-Level Synthesis (HLS) frameworks struggle to generate efficient solutions for input-dependent dataflows. Furthermore, at deployment time, design-space exploration (DSE) approaches rely on detailed simulations, infeasible for complex, real-world designs and workloads. Accelerating input-dependent streaming pipelines on FPGAs requires configurable implementations that expose a broad, performant design space, coupled with high-level performance abstractions to rapidly generate workload-specialized instantiations.

To address the design challenge, the first part of this dissertation identifies common input-dependent design patterns and presents a systematic methodology for representing them via the streaming paradigm in statically scheduled HLS. We showcase this methodology in the development of highly parameteriz-

able HLS design templates for two primary application case studies. We introduce RapidScan, an HLS-based string matching library designed to enhance the customizability and portability of an existing RTL-based network intrusion detection accelerator. To demonstrate its cross-domain applicability, we use RapidScan to deploy RapidDetect, a 200 Gbps streaming log monitoring accelerator on a single FPGA-enabled server, achieving a 4x cost reduction compared to software-based solutions. We further explore the limitations in describing input-dependence in HLS by investigating packet routing within a database aggregation accelerator, compared against ReConnect, a novel RTL-native Network-on-Chip (NoC) generator. ReConnect harnesses the regularity in NoC structures to unlock a vast design space directly in RTL while reducing resource consumption by over 35% and seamlessly integrating with existing HLS workflows.

Leveraging the structure imparted by the streaming paradigm to input-dependent pipelines in HLS, in the second part of the thesis, we present RapidQ, a queuing-inspired performance modeling workflow for rapid workload-driven design-space exploration (DSE) at deployment time. RapidQ distills a workload's input-dependent behavior into a workload model, capable of predicting performance across various design parameters, using a single functional simulation. RapidQ's lightweight queueing simulator exercises this workload model to achieve a 7x speedup over state-of-the-art approaches and is over 100x faster than RTL simulation, without significant accuracy loss. Driven by this fast performance estimation, we designed an automated DSE flow that co-tunes module throughputs and buffer sizes, achieving up to 42% resource savings for real-world workloads.

By pairing HLS-powered, highly parameterized design templates with rapid queueing-based performance estimation, this thesis provides a scalable approach to designing and tuning input-dependent streaming pipelines, enabling hardware acceleration with customized deployments on FPGAs.

Contents

List of Figures

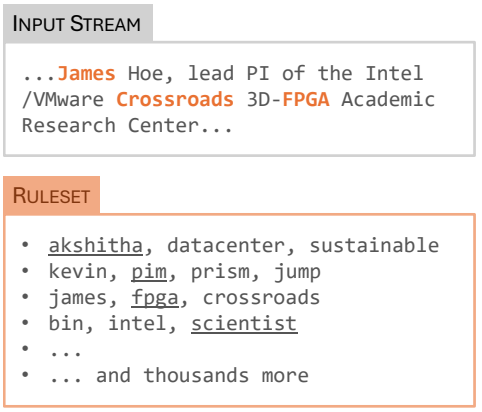
List of Tables

Chapter 1

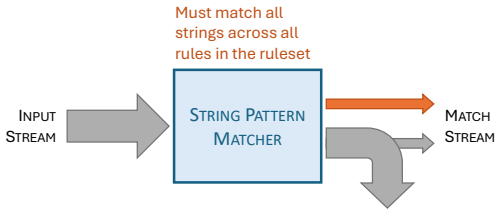
Introduction

1.1 What is Input-Dependence?

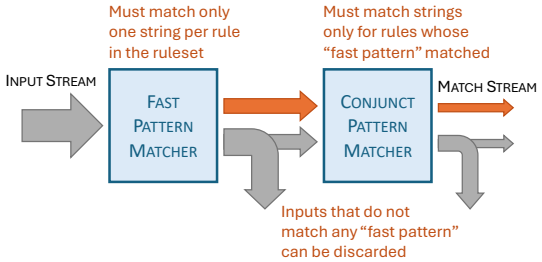
Consider a string matching application that must match a stream of input data packets (e.g., news blurbs) against a set of string-based rules (e.g., keywords) as shown in Figure 1.1a. A naive approach to implementing this application might check for all strings in the ruleset against every input. We can immediately see that this approach is very wasteful if, most of the time, inputs do not match any of the rules. Taking note of the fact that even if one string in a rule does not match, the input will not match the entire rule, a smarter implementation first filters using only one string per rule, as shown in Figure 1.1c. This common-case optimization strategy can produce a much more resource- and cost-effective solution by saving full string matching only for the subset of inputs that match a "fast pattern" and against the subset of rules the "fast patterns" belong to. This input-dependence is found in many applications, including network intrusion detection systems [4–7], log monitoring [8], database analytics [9, 10], and machine learning [11, 12], with software implementations often making use of common-case optimization to



(a) Data packets in the input stream are matched against a set of rules, each possibly containing more than one string.



(b) A naive implementation matches all strings in the ruleset against every data packet in the input stream.



(c) Common-case optimization enables cheap pre-filtering with a single string per rule followed by matching other strings only for rules flagged by the pre-filter.

Figure 1.1: Common-case optimization is essential for efficient handling of Input-Dependent Streaming Applications.

significantly reduce computational complexity. *However, this same pattern poses a significant challenge to hardware acceleration.*

1.2 Common-Case Optimized Hardware Acceleration

Application-Specific Integrated Circuits (ASICs) or Domain-Specific Accelerators (DSAs) have seen widespread adoption to provide hardware acceleration critical to sustaining generational growth in performance as general-purpose architectures plateau [13, 14]. By trading-off general-purpose flexibility for specificity to a single class of applications, these accelerators can deliver performant and energy-efficient

solutions for a broad range of applications such as LLM training/inference [15, 16], video transcoding [17], and cryptography [18, 19]. Moreover, research in new packaging technologies and high-level development tools (e.g., High-Level Synthesis), has significantly offset the design and fabrication cost of ASICs, garnering interest from more domains, such as High-Performance Computing [14].

Hardware acceleration also provides an outsized benefit to streaming data-on-the-move applications. Custom architectures optimized for the streaming data access pattern deliver orders-of-magnitude efficiency gains by eschewing the overhead from complex control and communication structures, and memory hierarchies of general-purpose processors [20–25].

1.2.1 Changing Common-Case

Despite the proliferation of hardware acceleration and the criticality of input-dependent streaming applications in domains such as security [7, 8, 26], these applications are still largely executed on software systems [5, 27]. The fundamental limitation arises from the incompatibility between the dynamic nature of common-case optimization and the rigidity of ASICs and DSAs.

Continuing with the example in Section 1.1, a string matching system tuned for a specific common-case will lose efficiency as the common-case shifts. For instance, a sudden surge in a news topic's (rule's) popularity will cause an increase in traffic to the Conjunct Pattern Matcher which it might be under-provisioned to handle, dropping end-to-end throughput. Moving to a new domain like network security can introduce entirely new performance targets (e.g., 100 Gbps vs. 200 Gbps network rates) and functional requirements for common-case optimization (e.g., packet-header-based filtering). To maintain reduced computational complex-

ity and keep overall application costs low, the system's common-case provisioning must be updated to reflect these shifts.

While software is programmable by nature, due to their lack of flexibility, DSAs for input-dependent applications must either (a) commit to a common-case provisioning (e.g., input throughput to the Conjunct Pattern Matcher) at design time making the accelerator brittle to changing common-cases or (b) fallback to the naive implementation of assuming the worst-case which might be prohibitively expensive (e.g. matching against full regular-expressions [28, 29]). *Can Field-Programmable Gate Arrays (FPGAs) offer an alternative?*

1.2.2 Harnessing the Programmability of FPGAs

FPGAs are uniquely positioned in the spectrum from ASICs to general-purpose architectures, delivering a middle-ground in terms of both efficiency and flexibility [30]. By emulating digital circuits, FPGAs execute custom hardware designs with efficiencies within an order-of-magnitude of equivalent ASIC implementations [31, 32]. But unlike ASICs, owing to their reconfigurability, FPGAs are programmable. Designers have leveraged this programmability, even for regular *input-independent* computations, to close the efficiency gap with ASICs by building adaptable accelerator generators [33–35], as opposed to one-shot implementations, enabling end-users to deploy instantiations tailored to their needs.

FPGAs' programmability serves an even more critical need for input-dependent applications which have no other alternative for hardware implementations. FPGA-based accelerators can freely utilize common-case optimization techniques, switching to a different customized instantiation for a new common-case, without being locked into the worst-case. Leveraging programmability coupled

with hardware-class efficiency, input-dependent applications can finally reap the benefits of hardware acceleration through FPGAs.

1.3 Research Challenges and Goals

The central goal of this thesis is to enable the acceleration of common-case optimized input-dependent streaming applications on FPGAs. While FPGAs possess the theoretical flexibility, this potential to support customized instantiations is wasted if these instantiations cannot be generated systematically for different common-case scenarios.

Software systems are inherently scalable and runtime adaptable. By tailoring common-case optimization to runtime conditions using parallel implementations, real-time performance monitoring, and autoscaling [36–39], software systems can even offset the limited efficiency gains provided by inflexible hardware acceleration. However, FPGAs have traditionally relied on the same hardware frameworks, including programming languages and design-space exploration (DSE) approaches, used by domain-specific accelerators. Their high engineering costs and long design cycles place an undue emphasis on squeezing performance out of a single implementation. Consequently, hardware-oriented programming languages target fine-grained control over the generated hardware [40–42] and DSE approaches use high-fidelity performance simulations [43], at the expense of configurability and rapid exploration, the key enablers of customized deployments.

Effectively utilizing the programmability of FPGAs hinges on enhancing FPGA workflows with software-like agility in (1) generating varied instances across the performance-cost tradeoff space and (2) finding the right instance for a given use-case. In the remainder of this section, we identify the key challenges in building

configurable designs and design-space exploration for input-dependent streaming applications on FPGAs.

1.3.1 Building Parameterizable Design Templates using HLS

To address the lack of configurability of RTL representations in traditional hardware-description languages (HDLs), higher-level FPGA design frameworks such as High-Level Synthesis (HLS) [44–46] have played a critical role in improving the accessibility of configurable hardware design [47, 48]. HLS allows hardware to be expressed in a high-level language such as C++, using most of the language features for compile-time expressibility such as object-oriented features and template meta-programming, essential to designing parameterizable hardware designs. Furthermore, by automating the mechanical implementation details of low-level control logic such as pipelining, and providing fast functional simulation capabilities, HLS delivers a major productivity boost to hardware designers in developing readable and reusable implementations [49].

However, this productivity gain from HLS breaks down in the face of input-dependent patterns, especially within commercial tools [45, 50]. While input-dependent applications should be optimized for common-case behavior, they must still maintain functional correctness for the uncommon-case that can be triggered in a real-world deployment. For instance, in the string matching application discussed in Section 1.1, the Fast Pattern Matcher typically outputs one rule hit per packet to the Conjunct Pattern Matcher. Yet, the system must also process the occasional packet that generates multiple hits without degrading common-case throughput. Unfortunately, the static scheduling relied upon by commercial HLS tools precludes this type of dynamic, data-dependent execution, producing inefficient hardware

provisioned for the worst-case, undermining the fundamental flexibility that FPGAs provide.

HLS tools have tried to supplement the underlying schedulers' capabilities with data-dependent hardware-friendly design paradigms such as dataflow [51] and stream optimizations [52]. In this dissertation, we show that without a structured framework to reason about and represent input-dependent patterns, HLS implementations can regress into being more mangled and less efficient than the HDLs they were meant to replace. Hence, the first research goal of this thesis is to *study the ability of HLS to represent input-dependent design patterns and to provide guidelines for when and how to fall back to traditional HDLs without losing configurability.*

Towards this goal, we first identify and provide a hierarchy of design patterns that provide structure to designing and reasoning about input-dependence. Based on these input-dependent design patterns, we develop RapidScan, an HLS-based string matching library, derived from an existing FPGA-accelerated implementation for network intrusion detection. Leveraging the stream processing paradigm in HLS, RapidScan isolates the core string matching filters from the infrastructure kernels that encapsulate input-dependence. We further study the implementation of the most resource-intensive infrastructure kernel, comparing a stream-based decomposition against a monolithic single kernel implementation and an optimized RTL implementation. The resource consumption of the stream processing-based implementation is within 10% of the RTL implementation, while being more readable and easily configurable, and is 2x cheaper than the monolithic implementation. Lastly, the parameterizability of RapidScan allows us to deploy a novel 200 Gbps FPGA-accelerated log monitoring system RapidDetect, supported by the rapid development of new HLS kernels essential to handling formatted log traffic and log

monitoring rulesets.

To understand the limits of representing input-dependent design patterns in HLS, we evaluate packet routing, the most complex of these patterns, within a database aggregation accelerator using ReConnect, a novel RTL-native Network-on-Chip (NoC) generator. Packet routing is a highly structural paradigm with data-dependence at its core. While prior work [53] has demonstrated that NoCs can be represented in HLS by strictly mimicking hardware structures, this approach incurs a severe penalty in developer productivity. To establish an iso-effort comparison for database aggregation, we evaluate ReConnect against a custom HLS butterfly topology implemented via stream-based decomposition. Our evaluation demonstrates that ReConnect consumes 35% fewer resources than the equivalent HLS design at iso-throughput. Furthermore, ReConnect provides highly efficient alternative design points with its default Input-Queued configuration achieving a 3x reduction in resource consumption in exchange for a 26% reduction in saturation throughput.

1.3.2 Performance Estimation for Design-Space Exploration

Even with highly parameterizable designs like RapidScan and ReConnect available, deploying them efficiently requires navigating a massive workload-dependent performance-cost design space to find the optimal instance for input-dependent applications. Whether the objective is to maximize performance given a fixed cost budget, meet a performance target while minimizing cost, or a combination of the two, iterative DSE must have the ability to evaluate candidate configurations against their performance and cost estimates. While software can scale across cores and processors from a single binary, hardware designs including FPGA bitstreams need to

be recompiled to instantiate a different tradeoff point. A vast design space makes maintaining a library of all possible design instances infeasible, and the lengthy hardware compile times preclude the use of repeated runtime testing common in software autoscaling [37, 39].

Therefore, deployment-time (or compile-time) tuning relies on performance estimation to guide design-space exploration (DSE) toward optimal configurations [54]. While HLS-based DSE tools for *input-independent* designs successfully utilize static, model-based techniques to capture performance-resource tradeoffs [55–59], these static analyses fail for input-dependent dataflows. Real-world workload behavior is highly dynamic and cannot be readily summarized as parameters in an analytical model. For example, the string matching application from Section 1.1 processing a news stream may experience bursts in traffic when a specific rule surges in popularity due to real-world events. Capturing the full performance impact of such complex, time-varying behaviors on streaming pipelines necessitates simulation.

Traditionally, designers rely on RTL simulation for workload-driven performance estimation. While this provides cycle-accurate data, its detailed, event-driven nature results in impractically slow execution; simulating millions of real-world news blurbs through RTL is fundamentally unscalable. Designs implemented using HLS can leverage software simulation to faithfully verify functionality at high speeds, but lack of timing or clock-cycle awareness makes this approach incapable of accounting for performance bottlenecks or backpressure during FPGA execution. Previous attempts to bridge this gap for input-dependent DSE have fallen short. Runtime execution via incremental compilation [60] offers high accuracy, but does not improve enough on baseline hardware compilation times. Similarly, tools based on schedule-augmented software simulations [2, 61] remain bottlenecked by the repeated exe-

cution of detailed functionality. While these methods are acceptable during initial accelerator design and debugging, they lack the higher-level abstraction required to scale effectively to large real-world designs with traces that span tens of millions of cycles (e.g., intrusion detection workloads in [6]). To address this gap, the second research goal of the thesis is to *develop lightweight and scalable performance simulation techniques that support rapid deployment-time design-space exploration of real-world designs and workloads.*

In this dissertation, we present RapidQ, a systematic approach to modeling and tuning HLS-based implementations of input-dependent streaming pipelines at deployment time for FPGAs. Leveraging the design patterns developed in the first part of this thesis, RapidQ abstracts core compute kernels into their static schedules, modeling workload-driven dynamic performance through a queueing system of the kernel modules interconnected by stream buffers [62]. By cleanly decoupling functionality from timing, this approach allows a single functional trace to be reused across multiple design-space exploration (DSE) iterations to evaluate different module throughputs and buffer sizes. To obtain end-to-end performance estimates, RapidQ employs a SystemC-based queueing simulator that translates the interaction between the functional trace and HLS-extracted timing models into queueing behavior across the streaming graph. Operating exclusively at the queue-server granularity allows this simulation process to bypass detailed application functionality, achieving a 7x speedup over state-of-the-art HLS simulators [2, 3] and a 100x speedup compared to RTL simulation, significantly reducing DSE turnaround time.

Lastly, to showcase the effectiveness of RapidQ's performance modeling, we developed an automated tuning flow to identify resource-efficient configurations for input-dependent streaming pipelines that meet specific throughput targets for a

given input workload. The agility of RapidQ's simulation framework allows us to explore the vast design space encompassing the cross-product of buffer sizes and modules' parallelism/unroll parameters. This enables the tuning loop to efficiently tradeoff module resources against buffering capacity, minimizing the total resource footprint of the pipeline. Using our tuning flow, we show that co-optimizing module throughputs and buffer sizes saves up to 42% in resource footprint compared to baselines [60, 61] that tune these parameters in isolation.

1.4 Thesis Statement

> *Although FPGAs are uniquely suited to accelerate input-dependent streaming applications, their programmability remains underutilized due to the rigidity of current hardware toolchains. Leveraging the stream processing paradigm to systematically represent input-dependent patterns as configurable templates in high-level synthesis, coupled with a lightweight queueing abstraction to model performance, can enable the rapid design-space exploration necessary for efficient acceleration using customized deployments for input-dependent streaming applications.*

1.5 Dissertation Contributions and Outline

In support of the thesis statement, this dissertation makes the following contributions that showcase the systematic design and tuning of input-dependent streaming applications on FPGAs.

- **Systematization of input-dependent design patterns in streaming pipelines (Chapter 2).** As a foundation for representing and modeling

input-dependent streaming pipelines, this dissertation outlines the hierarchy of input-dependent patterns. Using these patterns, complex application pipelines can be broken down into core, parallelizable compute kernels connected by infrastructure kernels that encapsulate the data-dependence. These patterns also provide the framework for developing higher-level performance abstractions.

- **Methodology for designing input-dependent streaming pipelines as configurable design templates using HLS (Chapter 3).** We use string matching as a case-study to implement RapidScan, an HLS-based Streaming String Matching Library. RapidScan implements string matching filters as parallelized and deeply pipelined kernels with extensive parameterization both for tuning functionality and performance-resource tradeoffs. Infrastructure kernels act as shims between the various filter components, acting on the input-dependent dataflow to enable common-case optimization.

- **RapidDetect: 200 Gbps FPGA-accelerated Streaming Log Monitoring (Chapter 3).** Leveraging the configurability of RapidScan and the productivity conferred by HLS, this dissertation presents RapidDetect, a novel streaming Log Monitoring accelerator operating on Sigma rules. We implement new kernels specifically designed for parsing JSON-formatted logs and enhancing existing filters to robustly process Sigma rules.

- **ReCONNECT: RTL-Native NoC Generator (Chapter 4).** Understanding the limits of representing input-dependent patterns in HLS requires a state-of-the-art RTL baseline. We present ReConnect, a NoC generator written entirely in SystemVerilog based on FPGA-friendly hardware design

principles for modern architectures.

- **Characterization of HLS vs RTL implementations for input-dependent design patterns (Chapters 3 and 4).** This dissertation uses RapidScan and an FPGA-accelerated database aggregation pipeline to study the compaction and routing design pattern implementations. We implement equivalent RTL and HLS-based implementations for both design patterns, showing that while compaction can be represented in HLS with minimal loss in efficiency, Network-on-Chips are better suited for RTL-native implementations.

- **Queuing-Based Performance Modeling and Simulation of Input-Dependent Streaming Pipelines (Chapter 5).** We present RapidQ, a lightweight queueing-based performance modeling and simulation framework. RapidQ's high-level abstraction and fast queueing simulation implemented in SystemC, significantly reduce simulation load, making it 7x faster than state-of-the-art approaches with minimal loss in accuracy.

- **Automated flow for co-tuning module throughput and buffer sizes for resource efficiency in a streaming pipeline (Chapter 5).** Empowered by the simulation agility delivered by RapidQ, we present an automated tuning strategy with the objective of minimizing resource utilization while meeting a performance target. The strategy explores the vast design space of co-tuning both module throughputs and buffer sizes achieving over 42% reduction in resource utilization compared to tuning either in isolation.

Lastly, Chapter 6 presents concluding remarks, discussing the impact and limitations of the dissertation, and future research directions.

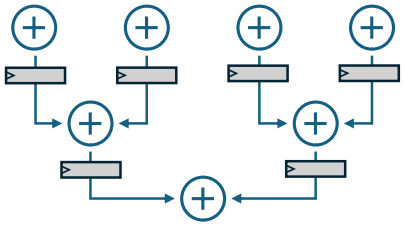
Chapter 2

Input-Dependent Design Patterns in Streaming Pipelines

Configurable design templates that expose a broad design space are critical to enabling common-case optimization for input-dependent streaming applications. Traditional hardware design flows that use low-level languages, such as SystemVerilog or Chisel [42], provide limited parameterizability and hamper development productivity, essential to customizing hardware to changing common-cases. High-level synthesis (HLS) can confer all the benefits of higher-level languages such as template meta-programming and object-oriented programming to automatic hardware generation, bringing software-like development agility to hardware design. However, input-dependent dataflows pose a challenge to the regular pipeline and data parallelism that HLS depends on to generate efficient and performant hardware using static scheduling. In this chapter, we identify key input-dependent design patterns that arise due to common-case optimization and how the streaming data paradigm in HLS can help separate core parallelizable computation from this input-dependence,

```
using data_t = int;
#define N 8
void reduction(data_t A[N], data_t B[0]) {
  B[0] = 0;
  #pragma unroll
  for (int i = 0; i < N; i++) {
    B[0] += A[i];
  }
}
```

(a) HLS Implementation of a reduction function written in C++. The unroll pragma specifies that the loop must be fully unrolled and performed in parallel using multiple adder units if possible.



(b) Generated hardware for the 8 inputs of 32 bit integers. HLS generates an efficient pipelined reduction tree to achieve high throughput at the targeted maximum frequency.

Figure 2.1: Example of Automatic Hardware Generation using High-Level Synthesis (HLS) for an Adder Reduction Tree

enabling performant but configurable hardware designs.

2.1 Background and Motivation

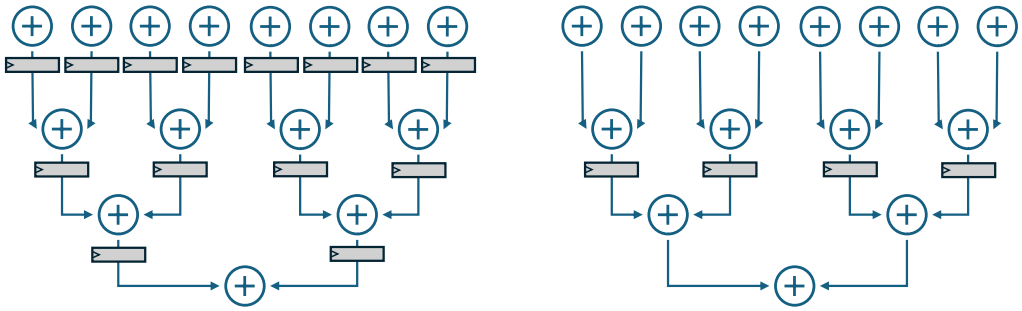
2.1.1 High-Level Synthesis for Parameterizable Designs

High-Level Synthesis (HLS) toolchains are specialized compilers that translate programs written in high-level languages, such as C/C++, directly into hardware implementations. By performing dataflow analysis on sequential code, HLS automatically extracts parallelism while allowing designers to control the performance-resource tradeoff through pragmas that dictate memory and compute layouts. Fig-

ure 2.1 illustrates how HLS tools generate efficient hardware for an adder reduction kernel represented in C++ and annotated with these performance pragmas. For a specified target frequency, HLS automates the generation of both the control and data paths of the reduction tree, ensuring the resulting hardware functionality strictly matches the software representation. Paired with rapid functional testing via software execution, this agile, correct-by-construction approach to hardware generation significantly enhances designer productivity and broadens the accessibility of hardware design.

Effectively exposing a design space, however, relies on parameterizable templates capable of generating a diverse range of efficient hardware implementations rather than one high-performance instantiation. While parameterization in traditional Hardware Description Languages (HDLs) like SystemVerilog allows for compile-time modifications, their rigid, cycle-level structural encoding restricts flexibility. This rigidity makes it difficult to maintain high performance and efficiency across parameterizations that require fundamental shifts in the hardware's control and data paths. In the adder reduction example, transitioning to a different data type, such as an 8-bit integer or a floating-point number, requires more than a simple data-width parameter adjustment. 8-bit adders have a much smaller timing delay, reducing the need to pipeline every adder unit. Floating-point adders, on the other hand, are more complex and typically require mapping to hardened DSP slices on an FPGA to remain efficient. Consequently, parameterizable designs in HDLs demand substantial manual effort to specify customized optimizations for each parameter instance, rendering a broadly compile-time tunable implementation unwieldy.

The true strength of automatic hardware generation via HLS lies in bridg-



(a) Reduction tree for N = 16 and 32-bit integer data type adds a layer to accommodate the wider input.

(b) Reduction tree for N = 16 and 8-bit integer data type uses fewer pipeline stages.

Figure 2.2: HLS can generate efficient hardware for a variety of configurations from a single implementation through parameterization in the higher-level software representation

ing this gap. By leveraging software-level parameterization, such as type systems and metaprogramming features, designers can easily modify functional specifications that are then automatically translated into efficient hardware implementations at compile time. For instance, altering the data type or the number of inputs in the adder reduction kernel (Figure 2.1a) yields a variety of distinct data paths. As demonstrated in Figure 2.2, each resulting architecture is automatically optimized to efficiently achieve the targeted frequency and balance performance-resource trade-offs. This ability of HLS to map a single parameterizable implementation into a broad design space of efficient instantiations makes it ideal for developing customizable design templates for input-dependent applications.

2.1.2 Streaming Data Paradigm in HLS

HLS tools also feature stream-based communication primitives in C/C++ such as the Vitis HLS Stream library [63] and SYCL pipes [64] to support streaming

```
template <typename T>
void kernel(hls::stream<T> in, hls::stream<T> out) {
  "architectural state" variables;
  while (true) {
    "transient state" variables;

    // Could read multiple pipes
    T in_flit = in.read();

    // Loop body (can do anything that statically elaborates into a
    dataflow)

    // Could write multiple pipes
    out.write(out_flit);
  }
}
```

Listing 2.1: Stream processing paradigm based implementation template for an HLS kernel that reads from one input stream and writes to one output stream

pipelines. Hardware accelerators for streaming applications such as networking and databases, that fit the stream processing/dataflow paradigm [65], can leverage this direct module-to-module streaming to bypass expensive memory accesses [22, 66, 67].

Listing 2.1 shows an example of a free-running stream-based kernel implementation in Vitis HLS [45]. In each iteration of the loop, the kernel reads data (a flit) from the input pipe/stream, performs some computation on it, and writes into the output stream. Architectural state variables can optionally be defined outside the loop, for persistence across loop iterations, allowing loop-carried dependence. Within the loop body, designers can use the rich set of features present in the HLS toolflow including unrolling to extract parallelism. In general, streaming kernels can read from and write to multiple streams or produce and consume multiple stream flits per loop iteration.

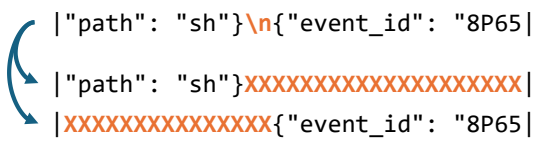


Figure 2.3: The input file stream must be split into packets using the newline character as the delimiter. Padding bytes are inserted to fill to the stream width.

2.1.3 Input-Dependence and HLS

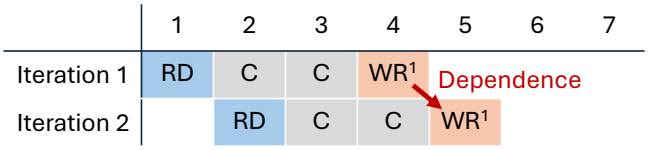
Custom hardware implementations achieve acceleration by leveraging the spatial parallelism present in algorithms. For regular algorithms such as dense matrix multiplication, this parallelism is independent of the data being processed and known at design time. This allows static specialization to spatial hardware structures that extract this parallelism. High-Level Synthesis (HLS) frameworks [46, 68] excel in generating efficient FPGA implementations for such data-*independent* algorithms. Using pipelining and unrolling of computational loop structures, HLS tools can statically interpret and extract parallelism even from sequential representations in C/C++.

Input-dependent applications, on the other hand, exhibit irregular parallelism which can vary with each input. While hardware can be tuned to support a certain common-case parallelism, uncommon-cases, as the name suggests, are not impossible and must be handled for functional completeness. However, most commercial HLS tools are statically scheduled [69], catering to computations that can make use of spatial parallelism in custom hardware. Since input-dependence is not statically determinable, these schedulers assume the worst-case, possibly limiting the throughput even for the common-case.

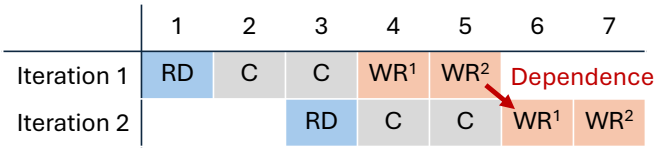
Illustrative Example

```
void splitter(hls::stream<flit_t> in, hls::stream<flit_t> out) {
  while (true) {
    flit_t in_flit = in.read();

    flit_t out_flit_before_newline;
    flit_t out_flit_after_newline;

    bool found_newline = false;
    // Search for the newline
    // Compute the padding for the output flits
    // Could take multiple cycles

    out.write(out_flit_before_newline);
    if (found_newline) {
      out.write(out_flit_after_newline);
    }
  }
}
```

Listing 2.2: HLS implementation of a splitter kernel that splits input flits based on the presence of the newline character.



(a) Schedule when no newline is found and only one flit needs to be written. The write-after-write dependence forces a delay of only 1 cycle between loop iterations and input reads.



(b) Schedule when newline is found and two flits need to be written to the output stream. The write-after-write dependence forces a delay of 2 cycles between loop iterations.

Figure 2.4: Depending on the presence of the newline character in the input flit, the splitter kernel must follow different schedules at runtime to achieve maximum performance.

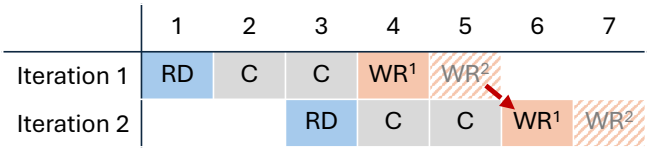


Figure 2.5: Static Scheduling assumes the worst-case leading to a two cycle delay between loop iterations even in the absence of a newline character in the input flit, halving the kernel throughput for long, multi-flit packets.

To illustrate, consider a kernel that processes a file stream by splitting individual lines into separate packets, as shown in Figure 2.3. Listing 2.2 provides an example HLS implementation. During each iteration, the kernel reads from the input stream and outputs either one or two flits, depending on whether a newline character is detected. For maximum performance, the hardware implementation of the kernel must schedule the next iteration of the loop based on this input-dependent newline detection as shown in Figure 2.4. However, for correctness, static scheduling must assume the worst-case scenario of two writes per iteration, with the generated hardware scheduled for a fixed two clock cycle delay between loop iterations to ensure it can accommodate the potential second write as shown in Figure 2.5.

If the file being processed comprises short lines that fit within a flit, this worst-case schedule matches the common-case. However, for a workload consisting of files with long lines (e.g., system logs), this implementation operates at half the maximum possible throughput, with the output stream idle for close to 50% of the time. This example illustrates the fundamental limitation of statically scheduled HLS in representing input-dependent streaming kernels as monolithic units.

2.1.4 Stream-Dependent Backpressure

The streaming data paradigm offers a relaxation to the strict static schedul-

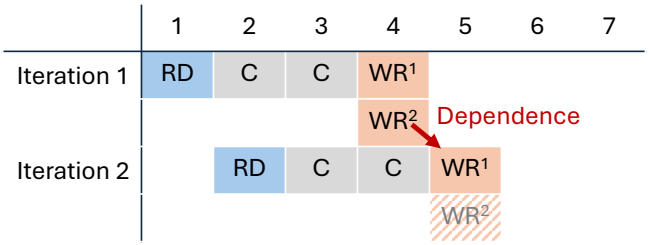
```
out_0.write(out_flit_before_newline);
if (found_endline) {
  // Scheduled in the same cycle but occurs conditionally
  out_1.write(out_flit_after_newline);
}
```

Listing 2.3: Modified HLS implementation of a splitter kernel using two output streams to allow fully-pipelined execution

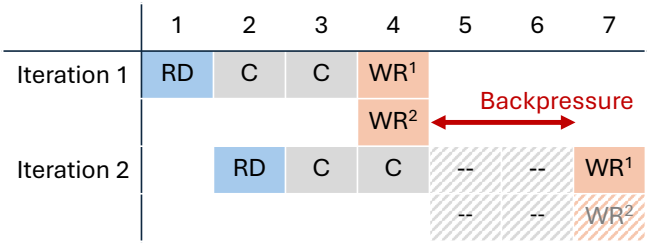
```
void merge(hls::stream<flit_t> &in_0, hls::stream<flit_t> &in_1,
           hls::stream<flit_t> &out) {
  flit_t flit_0, flit_1;
  bool valid_0 = false, valid_1 = false;

  while (true) {
    if (!valid_0) valid_0 = in_0.read_nb(flit_0);
    if (!valid_1) valid_1 = in_1.read_nb(flit_1);

    flit_t flit_out = valid_0 ? flit_0 : flit_1;

    if (valid_0 || valid_1) {
      out.write(flit_out);
    }

    if (valid_0) valid_0 = false;
    else valid_1 = false;
  }
}
```

Listing 2.4: HLS implementation of the merge kernel that merges two input streams using non-blocking reads. The kernel prioritizes the common-case stream that must support a higher throughput.

ing in HLS. While stream reads and writes must remain strictly timed relative to one another, HLS permits conditional execution and non-blocking stream accesses. Provided that these conditional operations do not waste cycles in the common-case, such as by conditionally writing to a parallel second stream (Listing 2.3), streaming kernels can still achieve fully pipelined execution. A downstream merge module can then conditionally read from these parallel streams to produce a single unified output stream, as shown in Listing 2.4.



(a) Output streams maintain low occupancy when newlines are rare allowing loop iterations to execute every cycle irrespective of the presence of newline.



(b) When the output streams eventually fill up due to lower downstream throughput, stream backpressure stalls the kernel independent of the presence of newline.

Figure 2.6: Stream backpressure allows statically scheduled kernels to be stalled independent of the input being processed currently, effectively shifting the input-dependence into the queueing behavior of stream buffers.

Due to input-dependent behavior, some iterations still require the handling of multiple flits. While the splitter kernel can potentially execute every cycle despite its higher output throughput, the merge kernel cannot push that same throughput into a single output stream. Instead, this input-dependence is dynamically managed via backpressure: the merge kernel conditionally reads its input pipes and stalls the upstream splitter only when necessary as shown in Figure 2.6, avoiding the pessimistic, every-cycle stall assumed by a static schedule.

Modern HLS toolchains manage this backpressure efficiently, scaling resource usage according to kernel complexity using stall-free or free-running pipeline optimizations [70, 71]. For designs relying on streams or dynamically managed dataflow

regions, these implementations use exit FIFOs to mitigate fanout on the stall path. By systematically leveraging the streaming paradigm and the associated optimizations, designers can achieve high-performance implementations of input-dependent stream kernels in HLS.

2.2 Design Patterns

The stream processing paradigm is essential for representing input-dependent dataflows within an otherwise statically scheduled HLS toolchain. Common-case optimizations, however, take several distinct forms, each giving rise to a unique design pattern which, if mishandled, can lead to inefficient HLS kernels. In this section, we identify four types of common-case optimizations and, for each, present strategies for separating input-dependence from the core parallelizable computation, yielding kernels that are more amenable to efficient HLS acceleration.

2.2.1 Early-Exit

The simplest common-case optimization is the early-exit pattern in which some packets in the stream are able to exit the pipeline early if they satisfy a filtering condition, and skip the rest of the computation. The string matching pipeline from Section 1.1 is a clear example of input-dependent early-exit where packets that do not match any "fast-pattern" exit the pipeline early. If in the common-case a significant fraction of packets exit the pipeline, the reduced load on the downstream stages can allow for more resource efficient, lower throughput implementations. This pattern is seen in applications such as networking [26] and early-exit neural networks [11, 12].

Stream-based Implementation Strategy. The simplicity of this pattern allows

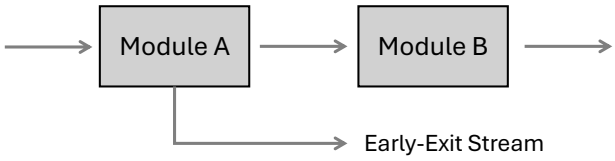


Figure 2.7: Combined implementation of the early-exit pattern relies on the kernel computing on the packets to make the filtering decision.

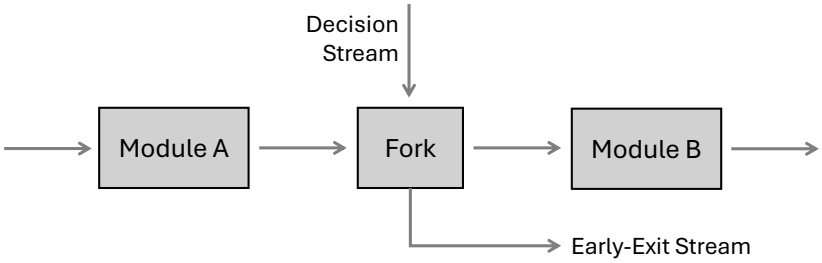


Figure 2.8: Dedicated fork kernel-based implementation of the early-exit pattern uses a separate per-packet decision stream to make the filtering decision.

it to be integrated into the decision-making HLS kernel without impact on performance as illustrated in Figure 2.7. But, for modularity and separation of the data and decision stream, this pattern can also be encapsulated in a separate steering module that takes as input the data stream and a decision stream and steers the data into one of two output streams (Figure 2.8).

2.2.2 Fast-Slow Path or Fork-Join

The fork-join pattern emerges in applications where different packets require different functions to be applied to them or produce variable amounts of input-dependent work, such as in graph-based accelerators [72]. By applying common-case optimizations, the hardware implementation of each function can be specialized according to its computational complexity and the expected volume of traffic it will serve. A classic example of this is the fast-slow path architecture, widely used in

```
template <typename T>
void kernel(hls::stream<T> in, hls::stream<T> out) {
  while (true) {
    T in_flit = in.read();

    if (condition(in_flit) == FAST_PATH) {
        out_flit = fast_path(in_flit);
    } else {
        out_flit = slow_path(in_flit);
    }

    out.write(out_flit);
  }
}
```

Listing 2.5: Naive software-like implementation of the fork-join design pattern in statically scheduled HLS severely restricts per-function customization

system designs such as Software-Defined Networking (SDN) [73]. This represents a specialized class of the fork-join pattern where a distinct common case requires high-throughput, low-cost processing, while a separate "slow path" handles uncommon, computationally expensive packets.

While a software implementation can easily handle this pattern using standard conditional structures like if-else or switch-case blocks, a direct translation to HLS (Listing 2.5) will produce an inefficient hardware design. Because HLS relies on static scheduling, both execution branches are forced to adhere to identical hardware timing constraints, severely limiting the ability to specialize the common-case function. Therefore, even if the slow path is rarely triggered, it must be provisioned to support the full throughput of the fast path to avoid becoming a system bottleneck. Although a modified structure that explicitly prioritizes the common case (Listing 2.6) can provide a reasonably efficient workaround when the slow path is exceptionally rare, this approach does not scale to generalized fork-join architectures.

```
template <typename T>
void kernel(hls::stream<T> in, hls::stream<T> out)
  while (true) {
    bool fast = true;
    while (fast) {
      T in_flit = in.read();

      fast = condition(in_flit);

      // Fast path processing
      out_flit = fast_path(in_flit);

      if (fast) out.write(out_flit);
    }

    // Slow path processing
    out_flit = slow_path(in_flit);
    out.write(out_flit);
  }
}
```

Listing 2.6: Heavily skewed common-case allows for dedicated fast-path handling with the slow-path implemented for functional coverage. Switching between the fast and slow-path in this implementation comes with a performance penalty.

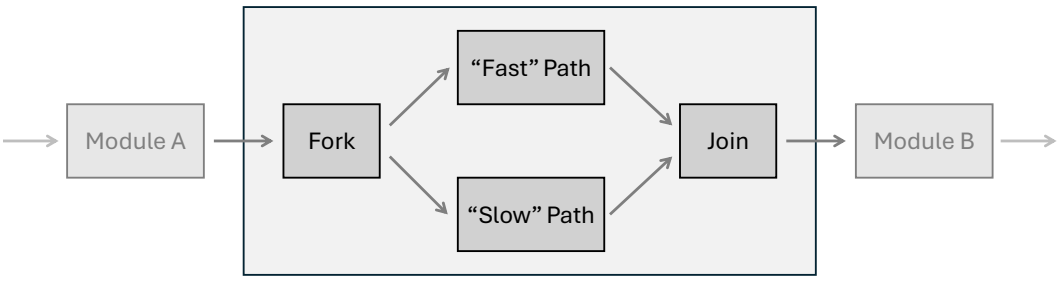


Figure 2.9: Fast-Slow path allows dedicated implementation and resource allocations for common-case packets requiring cheaper, higher throughput compute and rare-case packets that can be handled resource efficiently at a lower throughput.

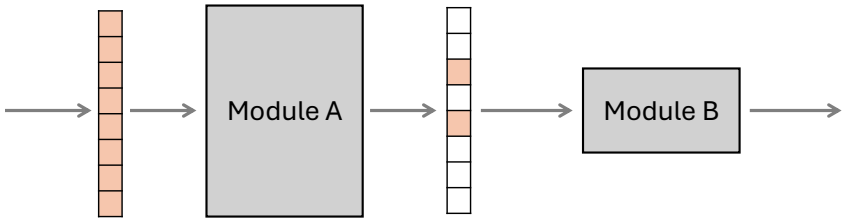


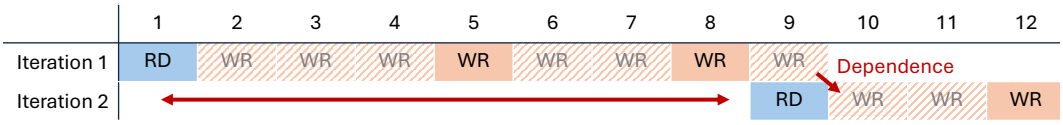
Figure 2.10: Module A produces 2 elements per flit on average which can be handled by a narrower Module B. A parallel implementation of Module A produces a sparse output which wastes processing cycles in B creating unnecessary backpressure.

Stream-based Implementation Strategy. To overcome the limitations, the fork-join design pattern can be explicitly decomposed into a packet steering kernel (fork), individual per-function kernels, followed by a stream join kernel to combine the individual function streams into one, as shown in Figure 2.9. Individual kernels can be tuned independently to the expected common-case traffic. In fact, in the degenerate case, both the fast and slow path could be simple passthrough kernels, similar to the proposed solution for the line splitter kernel in Section 2.1.3.

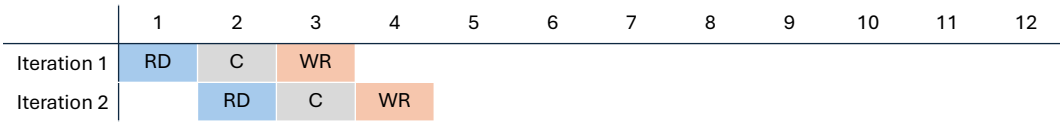
2.2.3 Sparsity

Consider the streaming pipeline shown in Figure 2.10. Module A consumes 8 elements per flit from the input stream every cycle and, depending on the input values, produces 0 to 8 output elements for each flit. The downstream module B processes the outputs from A. Assuming the worst-case, B would need to be provisioned to handle 8 elements per cycle. However, we can instantiate a cheaper and narrower module B if it is known at design time that, for the expected inputs of a certain deployment, module A outputs an average of 2 elements per cycle.

However, hardware implementations achieve high throughput by exploiting algorithmic parallelism. To scale performance, Module A must scan all elements of



(a) Static schedule for scanning a wide flit in smaller chunks tuned for the common-case leads to worst-case inter-iteration delay.

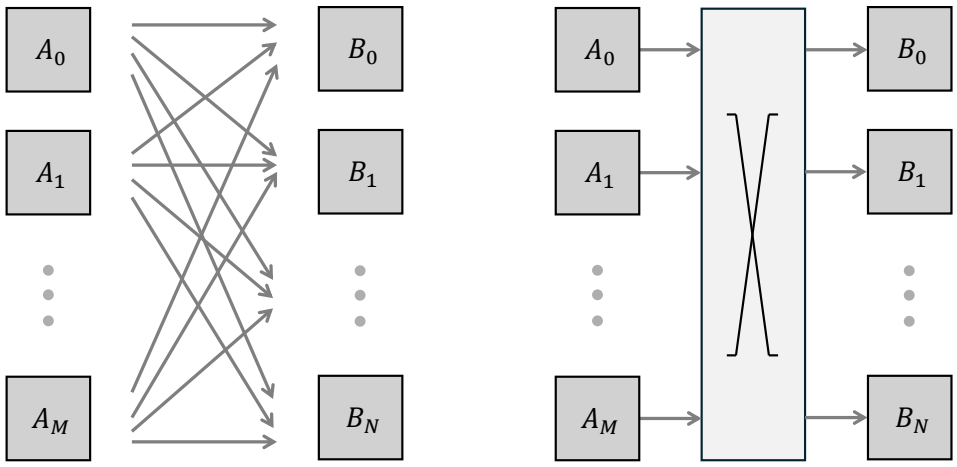


(b) When the output streams eventually fill up due to lower downstream throughput, stream backpressure stalls the kernel independent of the presence of newline.

Figure 2.11: Stream backpressure allows statically scheduled kernels to be stalled independent of the input being processed currently, effectively shifting the input-dependence into the queueing behavior of stream buffers.

an input flit simultaneously. Because most elements do not produce an output in the common case, this parallel scanning introduces input-dependent sparsity into the intermediate streams. If Module B were to process these sparse flits directly, it would either need to spend multiple cycles sequentially scanning the wide input (Figure 2.11a) or provision for the worst-case by processing all eight elements in parallel. Both approaches are sub-optimal: the former wastes clock cycles parsing empty inputs, while the latter is over-provisioned for the common-case.

Stream-based Implementation Strategy. Compaction, or packing the sparse flit into a narrower, denser flit, tailored to the common-case, is essential for streaming pipelines with the sparsity design pattern to efficiently utilize downstream parallelism. Because this compaction process is highly input-dependent, it is best implemented as a dedicated standalone module between Modules A and B to filter out empty elements within stream flits. The compaction will naturally propagate backpressure back to the source (Module A) in case of lower sparsity in the uncommon-



(a) Shuffle using point-to-point streams similar to the fork-join design pattern.

(b) Shuffle using a central crossbar/routers abstracting the implementation.

Figure 2.12: The shuffle design pattern connects M source kernels with N sink kernels enabling arbitrary input-dependent communication patterns.

case, allowing module B to be implemented with a more efficient schedule as shown in Figure 2.11b. Additionally, this technique can be viewed as a generalized stream join, bridging wide and narrow streams rather than multiplexing multiple distinct streams.

2.2.4 Shuffle

The most comprehensive design pattern, which can be viewed as a generalization of the patterns discussed, is the shuffle pattern (also called all-to-all). This pattern is commonly found in parallel processing architectures, such as stream processing for database analytics [74]. In streaming applications, the shuffle pattern arises when M source modules must route data to N sinks based on input-dependent conditions, such as a database table key. Depending on the workload, the actual communication traffic that the shuffle operation must support can vary significantly. In the

common case, the required bandwidth across links may fluctuate or become heavily imbalanced toward specific sink kernels. Efficiently supporting this bandwidth heterogeneity demands a scalable and configurable implementation of the shuffle design pattern.

Stream-based Implementation Strategy. A point-to-point implementation, as illustrated in Figure 2.12a, delivers the highest performance and can utilize HLS implementation strategies similar to the fork-join pattern. However, its high resource consumption and the complexity of managing very wide joins, particularly when ensuring fairness guarantees, make it impractical for large-scale shuffles. Like the compaction strategy, the shuffle pattern requires a dedicated, custom network implementation tailored to the common-case communication patterns of the specific input-dependent streaming application.

2.3 Summary

In this chapter, we presented four common input-dependent design patterns that arise when applying common-case optimizations to spatial hardware implementations. While static scheduling in commercial HLS tools is resource-efficient, it fails to extract parallelism from common-case optimized kernels, placing the burden on the designer to manually restructure the code on a case-by-case basis. We leverage the stream processing model provided by commercial HLS tools [63, 64] to impart structure to the problem of representing input-dependence in HLS. Using stream-based backpressure, we successfully decouple this input-dependence from the core parallelizable computations. The isolated computations are more amenable to static scheduling, enabling input-dependent streaming pipelines to be systematically realized as highly parameterizable HLS design templates.

2.4 Implications

2.4.1 Design

While the strategies presented in this chapter simplify the representation of core computations in input-dependent streaming pipelines, the structures managing the input-dependence itself, such as compaction and shuffle kernels, are highly complex. To construct a truly configurable design capable of adapting to varying common-case scenarios, these input-dependent kernels must also be highly parameterizable. In subsequent chapters, we use case studies to demonstrate that complex input-dependent structures, like the shuffle pattern, are better suited for RTL-native implementations. These RTL designs deliver higher performance and resource efficiency while remaining reusable across diverse deployment scenarios. Furthermore, the decoupling of the compute kernels allows them to remain in HLS, while the RTL-native input-dependent intermediaries simply expose streaming interfaces for seamless integration into the broader HLS framework.

2.4.2 Performance Abstraction

The stream processing paradigm facilitates scalability by exploiting parallelism and concurrency in the application, while efficiently utilizing the compute resources available on the device [75]. Input-dependent dataflows, presented in this chapter, depend on this structure to decompose algorithms into smaller, acceleratable kernels, characterized by high computational intensity and simplified control flow, conducive to high-performance HLS representation [66]. Performance abstractions can exploit this inherent structure, predicated on the insight that stream processing kernels function as simple and parallelizable units with buffers exhibiting queueing behavior [76]. A streaming pipeline can be modeled at the granularity of streaming

kernels connected by buffers, abstracting away the functional implementation details. This allows the model to focus on their parallelization parameters to enable rapid simulation and efficient tuning.

Chapter 3

RapidScan: Parameterized HLS-based Streaming String Matching Library

In this chapter, we present a case-study of developing a parameterizable design template for string matching and deploying it for Log Monitoring over Sigma rules [77]. We introduce RapidScan[1] , a novel High-Level Synthesis (HLS)-based streaming string-matching library that treats parameterizability and development agility as first-class design goals alongside performance and efficiency. Based on the design patterns and strategies laid out in Chapter 2, we designed the data-dependent computations of string matching into a library of fully-pipelined HLS kernels. Written in HLS, RapidScan exposes a broad design space of throughput and resource tradeoffs for individual string matching filters. In addition, the ability to compose existing library components with new domain-specific kernels allows RapidScan to pro-

[1] A short paper about this work was published as part of the Reconfigurable Computing Challenge at FCCM 2026 [78]

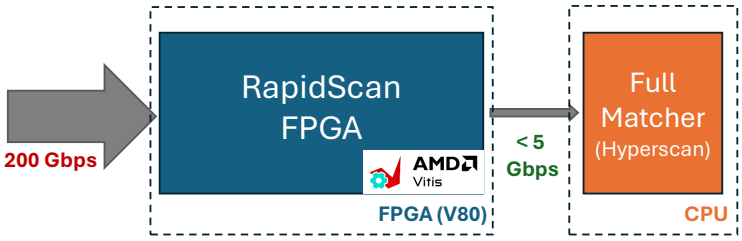


Figure 3.1: RapidScan String Matching operates at over 200Gbps using less than 30% of the Versal V80. Along with Hyperscan on the host CPU, the system, RapidDetect, can process incoming logs at 200 Gbps on a single server

vide high-throughput FPGA-based string matching solutions across workloads and domains, while minimizing energy and resource costs.

We use RapidScan to develop RapidDetect[2], a high-performance streaming log monitoring pipeline (shown in Figure 3.1). To achieve this, we designed novel kernels specifically for parsing JSON-formatted logs and enhanced existing filters to robustly process Sigma rules [77]. Our solution achieves near 200 Gbps throughput on real-world log traces, evaluating over 4,000 rules and approximately 18,000 string literals, delivering a 30x speedup and an over 12x reduction in infrastructure cost compared to a CPU-only Hyperscan baseline.

3.1 Motivation

As hyperscalers push the limits of data centers, the scalability of log monitoring has become critical for delivering efficient security solutions. However, existing log-based systems remain bottlenecked by batch-oriented architectures that require database ingestion prior to scheduled rule evaluation [8]. Streaming implementations using software-based regular-expression libraries such as Hyperscan [79] can

[2]This work was performed in collaboration with Tommy Tracy II, Matthew Beck, Wajih Ul Hassan and Kevin Skadron from the University of Virginia and is open-sourced at https://github.com/pigasus-hls/RapidDetect-RCC

reduce detection latency and provide orders-of-magnitude higher bandwidth than database query-based solutions. Nevertheless, the fundamental operation underlying regex matching, string matching, is ill-suited for general-purpose hardware, requiring hundreds of CPU cores to saturate network line rates of 200 Gbps and beyond [6].

Pigasus [6], a regex engine designed for network security, employs FPGA-based pre-filtering stages operating at line rates of 100 Gbps to rapidly discard traffic that fails to meet baseline rule conditions. This reduces the dependency on expensive regular expression matching engines in software, providing an extremely fast and energy- and cost-efficient solution on a single server. However, this common-case optimization strategy renders the RTL solution fragile to changes in input traffic composition and detection rulesets. Moreover, porting the design to the new domain of log monitoring introduces further inefficiencies that cannot be solved purely by retuning the existing design.

3.2 Background

3.2.1 Pigasus Intrusion Prevention System

Pigasus [6] is an FPGA-first regular-expression matching engine for network intrusion detection/prevention. Using common-case optimization strategies, Pigasus implements hierarchical filtering stages on the FPGA ending with full regular-expression matching on the host CPU. With each successive filter, although the compute complexity rises, the throughput required drops precipitously as traffic and the set of potential rule matches get filtered in the common-case. Using these successive filtering stages on the FPGA reduces the incoming network traffic by 6-100x and uses the host CPU to perform full regular expression matching on the few

remaining packets.

Pigasus implements two string matching filters, Multi-String Pattern Matcher (MSPM; also called the Fast Pattern Matcher) and the Conjunct Pattern Matcher (CPM; also called the Non-Fast Pattern Matcher), along with a network traffic specific filter, the Header Matcher (HM). The MSPM scans packet payloads at line rate for many thousands of string literals (called "fast patterns"), extracted from the regex-based rules, concurrently. For SNORT [4] rules implemented by Pigasus, the literal for each rule is hand picked by domain experts to be the most selective. This fast and cheap filtering is implemented using hash-table and shift-or-based scanning, removing a large fraction of incoming traffic. The CPM performs more selective checks, but only for rules that match in the MSPM, significantly reducing the complexity and speed of scanning needed. Using a bloom-filter-like fingerprint, the CPM combines literal matches to check for conjunctions (ANDs) of literals present in each regular expression, significantly reducing false positives.

3.2.2 Log Monitoring with Sigma Rules

Log Monitoring is another application domain which involves using Security Information and Event Management (SIEM) systems [8, 80, 81] to detect malicious activity in enterprise systems. Similarly to network IDS, SIEMs use rule-based detection by matching logs with threat signatures as in Sigma rules [77]. Designed to be SIEM-agnostic, Sigma rules use YAML syntax with detection conditions composed of field-value predicates joined by logical operators (`AND`, `OR`, `NOT`).

Unlike Snort rules that match byte patterns in network payloads, Sigma rules operate on semantic fields within structured log data, enabling detection of host-level threats invisible to network monitoring. Specifically, the logs and their

associated rules are JSON-formatted, consisting of `"field": "value"` pairs with a small set of unique field types. Furthermore, most Sigma rules [77] are of the Conjunctive Normal Form (ANDs of ORs). Because the CPM can only process strict conjunctions (ANDs of literals), these rules must be decomposed into individual conjunction rules.

3.2.3 Input-Dependent Patterns in String Matching

Between Filtering Stages

The main source of input-dependent behavior in Pigasus comes from the reduction in packet traffic between filtering stages and the reduction in the number of rules that must be checked in the downstream stages. The filtering efficiency of each filtering stage is directly dependent on the input packets, and even for the packets that do not get filtered, the number of rules they match on is input-dependent.

Early-exit patterns are found at the output of each stage, where packets that do not match any rules are forked out of the pipeline, similar to the example from Section 1.1. The reduction in the number of rule matches through a filtering stage appears as increased sparsity in the output rule flits. Pigasus uses compacting stages to remove empty rule hits while at the same time reducing the rule flit width to allow for a lower processing rate of downstream kernels.

Multi-String Pattern Matcher (MSPM)

The Multi-String Pattern Matcher (Figure 3.2), being the first stage, needs to check all the rules in the input stream. It uses multiple input-dependent structures to achieve high throughput in the common-case that packets will not match most of the rules.

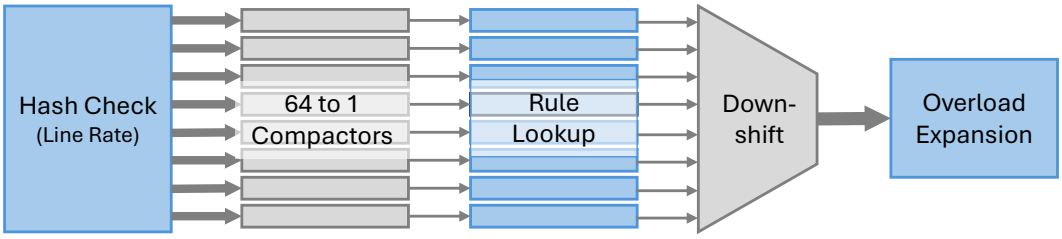


Figure 3.2: Multi-String Pattern Matcher Pipeline

The frontend of the MSPM uses hash-table and shift-or-based string matching in the Hash Check stage to check for all rule literals in every incoming flit. The tables encode only the existence of a match and the stage produces one element per byte in the input stream over eight channels for different length literals. To reduce the throughput needed for the rule lookup, MSPM uses compaction stages to bring down the extremely sparse per-byte match mask to only the elements that need a lookup. This structure parallels the sparsity example from Figure 2.10, where Module A is the Hash Check stage and Module B performs the rule lookup on the output of Module A.

The final stage, overload expansion, handles hash collisions in the Hash Check stage and expands the colliding rule to all the component rules. This stage exhibits a fast-slow path input-dependent pattern: in the common-case most rule hits can skip this step while only the rare colliding rules need to be expanded for correctness in the downstream stages.

Conjunct Pattern Matcher (CPM)

The Conjunct Pattern Matcher (Figure 3.3) checks multiple literals per rule in every packet that it processes. Similar to MSPM, it first uses a Hash Check to match all the literals in parallel. But since each rule can have a variable number of literals

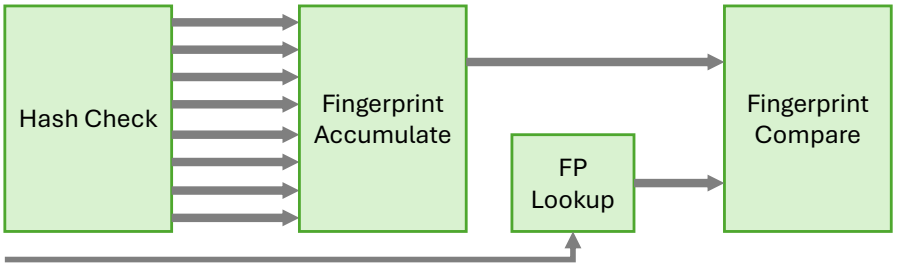


Figure 3.3: Conjunct Pattern Matcher Pipeline

to check for, using the raw matches is very expensive. Instead, the CPM uses the hash-table hits to generate a Bloom-filter-like fingerprint which is compared against precomputed per-rule fingerprints for only the rules the packet is tagged with from the previous stages. Although this structure is not input-dependent in itself, the throughput of the CPM is a function of both its input bitrate and the number of rules it can process per cycle, both of which are input-dependent.

3.3 RapidScan

The primary compute kernels in RapidScan comprise the MSPM and CPM, which are derived from Pigasus [6] (Figures 3.2 and 3.3). We completely redesigned these kernels and the streaming pipeline using the design patterns detailed in Chapter 2, with specialized infrastructure kernels to decouple the input-dependence from the main string matching modules. Furthermore, RapidScan treats parameterizability and development agility as first-class design goals alongside performance and efficiency. Using C++ and HLS features, the kernels incorporate extensive functional and performance parameterization, while being modular allowing customizable pipeline composition.

In the rest of this section, we describe the implementation of core string

matching kernels and infrastructure kernels along with the parameterization options available in RapidScan. We further analyze HLS and RTL implementation options for the 64-to-1 compactors within MSPM to provide guidance on the effective use of the stream processing model in HLS to achieve RTL-like efficiency.

3.3.1 String Matching Kernels

After the stream-based restructuring of the design, the core string matching kernels in the MSPM and CPM are easily parallelizable and simple feed-forward computations amenable to efficient hardware mapping using statically scheduled HLS. All kernels are implemented as fully pipelined free-running kernels with parameterization to control their parallelism factor, and hence the supported throughput. Being feed-forward, the implementations are similar to how these computations would be written for software, with the HLS toolchain appropriately pipelining the complex logic to meet given clock timing constraints.

To ensure modularity and ease of future updates, the hash-table and shift-or-based string matching functions used in the hash check stage are implemented as helper C/C++ functions with the actual HLS kernels as wrappers following the basic streaming structure in Listing 2.1. Read-only memories used to store the processed rules for the hash-table, shift-or bitmap and rule and fingerprint lookups are implemented as arrays in header files and can be independently updated for new rulesets.

Since the string matching algorithm is the same as described in the Pigasus author's thesis [82] and RapidScan's codebase is open source[3], we do not repeat the descriptions here.

[3] https://github.com/pigasus-hls/RapidDetect-RCC

Field Tagging for Log Monitoring

As described in Section 3.2.2, Log Monitoring, being a different domain from network security, has unique characteristics that the original Pigasus MSPM and CPM functionalities could not adequately accommodate. Field-aware matching is essential for Sigma rules because a rule match literal is specific to a designated field, but that literal can appear across multiple fields in a log event. The original Pigasus string matcher without field awareness would generate excessive false matches if used directly for Sigma rule checking. In practice, this field-aware matching reduces the number of false positives from the FPGA by 2x.

To enable field-aware matching in RapidScan, we analyzed the logs and Sigma rules and found the number of unique field types to be small, and out of those, only two or three were very common among log events and rules. Based on this observation, we introduced a new processing stage: The *Field Tagger*. The Field Tagger parses JSON-formatted logs for predetermined field literals and appends metadata to the logs, marking the position of bytes belonging to the particular field's value. MSPM and CPM match logic is updated with the required field type for each value literal, which can be checked against the field tags in the data flits for field-aware scanning.

As the first stage in the pipeline, the Field Tagger must sustain line-rate traffic. Leveraging the productivity gained from using HLS, we implemented the Field Tagger as a fully unrolled, deeply pipelined kernel within two days of one PhD student's effort. This development agility is critical to broadening the applicability of RapidScan to new domains which might have unique constraints and common-cases which cannot be handled by existing library functionality.

3.3.2 Input-Dependent Infrastructure Kernels

The Pigasus string matching pipeline consists of several common-case optimizations, as outlined in Section 3.2.3, and the majority of input-dependent structures fall into the early-exit and fork-join categories. We implement these kernels as common primitives parameterized using C++ templates, allowing reuse across various stages in the string matching pipeline. As an example, Listing 3.1 illustrates the templated packet steering kernel. We use the dedicated fork-based implementation of the early-exit pattern in Figure 2.8 to steer the packets marked safe by each filtering stage out of the pipeline.

Compaction

Sparsity occurs in the string matching pipeline within the MSPM pipeline in Figure 3.2 and also between stages to remove empty rule hits. In a 200 Gbps instantiation, the compactors after the Hash Check stage in MSPM, must compact input widths of 64 and in the common-case the number of hits might be as low as 1 per cycle. This reduction, starting with extremely wide inputs, consumes significant resources. Even in the Pigasus's RTL implementation, all the compactors put together consume close to as many logic resources as the Hash Check stage itself. Furthermore, this compaction is essential to keep the resources of downstream logic such as the rule lookup tables minimal.

Since scanning a sparse flit requires multiple clock cycles, preventing pipelining and limiting throughput, compaction must be implemented as a tree of $2 \rightarrow 1$ that can be parameterized to tune the compaction factor. However, implementing this tree as a monolithic kernel is unwieldy and resource intensive. The extremely structural and input-dependent design cannot be readily written in sequential C++

```
template <typename TFlit>
void steerPayload(stream<bool> &Steer, stream<TFlit> &In,
                  stream<TFlit> &Match, stream<TFlit> &Safe) {

  bool eop = false;
  bool flagged = false;
  bool first = true;

  while (true) {
    if (first) {
      // Read one steering decision per packet
      bool valid = Steer.read_nb(flagged);
      first = !valid;
    }

    if (!first) {
      TFlit inFlit;
      bool valid = In.read_nb(inFlit);
      if (valid) {
        if (flagged) Match.write(inFlit);
        else Safe.write(inFlit);
      }
      eop = inPayload.eop && valid;

      // Mark next as the first flit of the next packet
      if (eop) first = true;
    }
  }
}
```

Listing 3.1: Packet steering kernel based on the dedicated fork-based implementation of the early-exit design pattern.

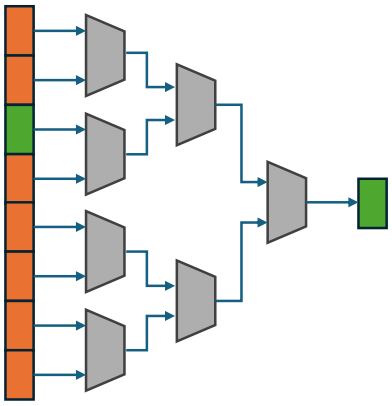


Figure 3.4: Reduction tree structure for implementing 8 → 1 Compaction

```
template <int FanIn>
struct compactor_core_t<FanIn, 1> {
  template <typename TFlit>
  static void compactor_core(stream<TFlit> *InStreams,
                             Cstream<TFlit> *OutStream) {
    stream<TFlit, COMPACTOR_PIPE_DEPTH> inter_0, inter_1;

    // Split the input streams into two halves
    stream<TFlit> *InStreams_0, *InStreams_1;
    InStreams_0 = InStreams;
    InStreams_1 = &InStreams[FanIn / 2];

    // Recursively instantiate sub-trees
    task core_task_0(compactor_core_t<FanIn / 2, 1>::template
                     compactor_core<TFlit>, InStreams_0, &inter_0);
    task core_task_1(compactor_core_t<FanIn / 2, 1>::template
                     compactor_core<TFlit>, InStreams_1, &inter_1);

    // Last 2-to-1 compactor
    task merge_task(compact_2to1<TFlit>, inter_0, inter_1, OutStream
    [0]);
  }
};
```

Listing 3.2: Template-based recursion to instantiate a compaction reduction tree with a parameterizable compaction factor

code, needing precise management of intermediate signals and timing to achieve full throughput. The control exercised over the code further restricts HLS compiler optimizations, forgoing the benefits of using HLS over direct RTL implementation.

Using the stream-based decomposition described in Chapter 2, we break the compaction structure into individual $2 \rightarrow 1$ streaming reduction kernels. With stream-carried backpressure that captures the input-dependence, each of these reduction kernels can be simply implemented as a stream-join similar to the merge kernel in Listing 2.4. At compile time, these kernels can be composed together to create a parameterized end-to-end compaction kernel using template recursion as shown in Listing 3.2 showcasing the configurability gained from using HLS.

Table 3.1 compares the Pigasus-derived RTL implementation of the com-

Table 3.1: Resource Utilization and Fmax Comparison of Compaction on Versal V80

Resource	Reduction Tree (RTL)	Reduction Tree (HLS)	Monolithic (HLS)
SLICEs	884	(+6.2%) 939	(+118.9%) 1,935
LUTs	4,460	5,100	9,754
— *Logic*	2,924	3,588	9,754
— *Memory*	1,536	1,512	0
$F_{\max}$	740 MHz	544 MHz	501 MHz

pactor with two Vitis HLS [45] variants: a reduction tree and a monolithic version, on the Versal V80 [83]. The monolithic compactor consumes twice the resources of the RTL baseline due to inefficient CLB LUT utilization for intermediate buffering, rendering it infeasible. However, the HLS-based reduction tree's resource footprint is within 10% of the RTL version. While it does not reach the peak frequency of the RTL implementation, the overall string-matching pipeline is bottlenecked at 400 MHz, making the difference inconsequential. Therefore, the enhanced configurability of the HLS reduction tree, coupled with its minimal impact on resource efficiency, makes it the optimal choice for RapidScan.

Overload Expansion

Overload Expansion resolves conflicts caused by collisions in the hash table or multiple rules sharing the same fast pattern. While hash collisions in Pigasus were extremely rare, the decomposition of the Conjunctive Normal Form used in Sigma rules generates rules that often share multiple literals, creating unavoidable conflicts. By employing a fast-slow path common-case design pattern with dedicated fork and join kernels in HLS (Figure 2.9), we implement Overload Expansion to maintain high throughput for non-conflicting rules while processing conflicted rules without

Table 3.2: Multi-String Pattern Matcher (MSPM) Parameters

Parameter Name	Description
MSPM_UNROLL	Unroll factor controls the packet payload throughput (8 = 200 Gbps)
MSPM_RESULT_WIDTH	Number of rules per output flit per cycle
MSPM_RESULT_BYCOMPACT	Use compaction to reduce final rules output width
MSPM_RESOLVE_CONFLICT	Enable Overload Expansion stage
MSPM_EXPAND_OVERLOADED	Choose between single kernel fast-slow path vs fork-join-based implementation
MSPM_USE_SHIFTOR	Enable the Shift-OR pre-filter
MSPM_FRWD_SUPER	Enable the Shift-OR supercharacter look ahead
MSPM_CHECKTAG	Enable tag checking in the MSPM.
MSPM_LOOKUP_WIDTH	Number of parallel rule lookup tables (controls compactor output width)
MSPM_CHECKFIELD	Enable field tagging in the MSPM

stalling.

3.3.3 Parameterization

Table 3.2 and Table 3.3 present details about the extensive parameterization available in RapidScan. All the parameters are implemented as preprocessor macros in C++ which control the appropriate functionality and performance of various library components.

3.3.4 RapidDetect

Using the kernels in RapidScan, we deployed a fully functional prototype of RapidDetect, a heterogeneous FPGA and CPU architecture for high-throughput, low-latency threat detection in streaming system logs. Sigma rules used for Log Monitoring contain literals and constructs that do not map entirely to the MSPM and CPM structure leading to false positives from the FPGA. Hyperscan runs as a separate, multi-threaded process on the host to complete the checking. Similar to

Table 3.3: Conjunct Pattern Matcher (CPM) Parameters

Parameter Name	Description
CPM_UNROLL	Unroll factor controls the packet payload throughput (4 = 100 Gbps)
CPM_USE_SHIFTOR	Enable Shift-OR pre-filter
CPM_FRWD_SUPER	Enable the Shift-OR supercharacter-based look ahead
CPM_CHECKFIELD	Enable field tagging in the CPM
CPM_CHECK_FPDWIDE	Enable an alternate wide 128-bit primary fingerprint bitmap shared across different literal lengths
CPM_CHECK_FPALT	Enable a second alternate fingerprint crosscheck using different hash bits to increase match strictness
CPM_METAIN_WIDTH	Number of rules per input flit per cycle
CPM_RESULT_WIDTH	Number of rules per output flit per cycle
CPM_RESULT_BYCOMPACT	Use compaction to reduce final rules output width

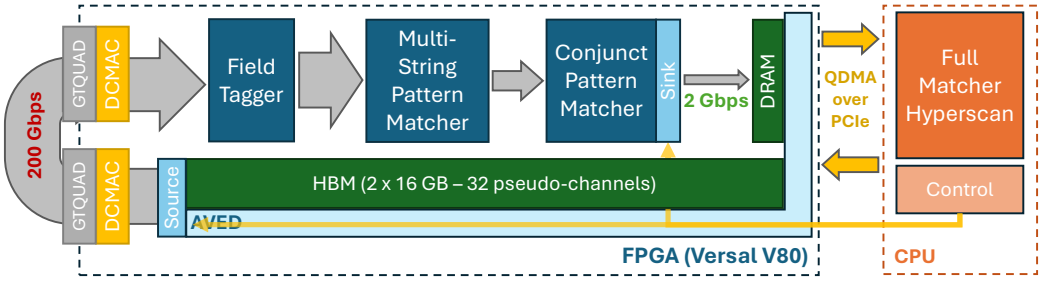


Figure 3.5: RapidDetect System Prototype implemented on the Versal V80 with 200G Ethernet

Pigasus, RapidDetect's FPGA component significantly filters the incoming log traffic reducing the pressure on Hyperscan. Table 3.4 details the system's resource utilization.

3.4 Evaluation

We evaluated RapidScan (1) as part of RapidDetect against existing software-based log monitoring engines[4] and (2) against Pigasus using SNORT rules for Intrusion Prevention.

3.4.1 Experiment Setup

All RapidDetect experiments were conducted on a workstation equipped with a Versal V80 Accelerator card [83]. We compare RapidDetect against Elasticsearch [8], Hyperscan [79] and SigmaLite [27] deployed on a single-node server with an Intel i9-13900K CPU with 8 performance cores and 16 efficiency cores and 128GB of DDR4 memory and a Samsung 990 Pro 2TB SSD. To compare against Pigasus, we replaced the memory IPs in the RTL implementation of Pigasus with Vivado equivalents while leaving the rest of the design unchanged.

3.4.2 Dataset and Rule Set for RapidDetect

We use publicly available system logs from DARPA's Transparent Computing (TC) program [84]. The data originates from the CADETS, TRACE, and THEIA performers during Engagement 5 (E5). CADETS includes over 1.2 billion log entries collected over ten days, totaling 1 TB in size. The datasets simulate real-world attacks interleaved with benign activities, making it representative of enterprise-scale

[4]This work was carried out in collaboration with Tommy Tracy II, Matthew Beck, Wajih Ul Hassan and Kevin Skadron from the University of Virginia

operational environments under heavy cyber threat. Since the DARPA CADETS, TRACE, and THEIA logs represent Linux-based systems, we restricted our evaluation to only Linux Sigma rules; at the time we downloaded the rules there were 200. We compiled the 200 Linux Sigma rules using the Sigma compiler [85] into Elasticsearch's Domain Specific Language (DSL) [86], and evaluated RapidDetect, Hyperscan, and Elasticsearch using this rule set.

3.4.3 Evaluation Metrics for RapidDetect

We evaluated two key metrics: throughput and Mean Time To Detect (MTTD). Throughput measures the volume of log data processed per second by RapidDetect. MTTD captures the time elapsed between log ingestion and detection output. For RapidDetect, we measured throughput by reading JSON-formatted logs directly from the FPGA's DRAM. This approach ensured that performance reflects the detection pipeline rather than I/O constraints. For MTTD, we computed the time from ingestion to alert generation in the hardware pipeline; we also collected MTTD time for the full pipeline including the Hyperscan regular expression engine. For Elasticsearch, we read logs from disk, applied preprocessing, and forwarded them to the Elasticsearch engine. We calculated throughput based on how much data Elasticsearch ingests per second. Detection latency was computed by comparing log timestamps with the ingestion timestamps embedded by Elasticsearch.

3.4.4 RapidDetect Results

Figure 3.6 plots the number of servers required to sustain 200 Gbps of traffic against threat detection latency. RapidDetect achieves 200 Gbps on a single server, demonstrating orders of magnitude lower latency than the state-of-the-art com-

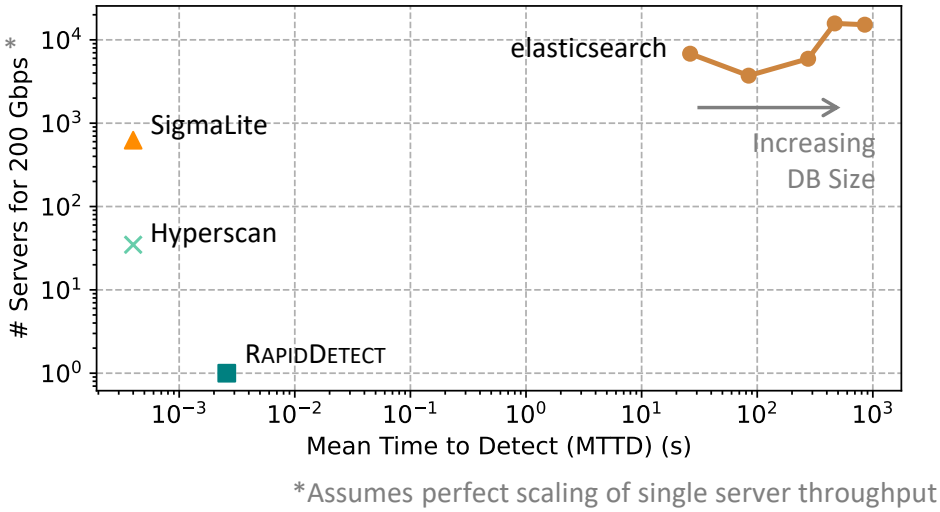


Figure 3.6: Number of servers needed to scale to 200 Gbps vs Mean Time to Detect for different Log Monitoring solutions. RapidDetect achieves near 200 Gbps throughput on a single FPGA-enabled server.

mercial solution, Elasticsearch [8]. While Hyperscan achieves a lower Mean Time to Detect (MTTD), scaling it to 200 Gbps would require approximately 40 servers. Consequently, even though the V80 FPGA costs ten times as much as the server, RapidDetect remains over 4x cheaper than an ideally scaled Hyperscan. Moreover, since RapidDetect utilizes only 30% of the V80, a smaller FPGA could reduce costs further. Elasticsearch, limited by its sub-1 Gbps throughput, would require over 2,000 servers, making it highly cost-prohibitive to scale to 200 Gbps.

3.4.5 Resource Comparison against Pigasus

Table 3.4 compares the resource utilization of the 100 Gbps RTL implementation of Pigasus against RapidScan, which is configured for Log Monitoring at 200 Gbps and Intrusion Prevention (IPS) at both 100 and 200 Gbps line rates. Pigasus' reliance on hardcoded clock frequencies optimized for older FPGAs restricts its ability

Table 3.4: Resource Utilization Comparison on the Versal V80 Accelerator

System	Component	LUTs	FF	BRAM	DSP
RapidDetect	Field Tagger	5,394	8,477	0	0
(200 Gbps	MSPM	125,245	185,757	577	0
Log Monitoring)	CPM	25,111	33,871	271	0
RapidScan	MSPM	133,667	180,327	590	0
(200 Gbps IPS)	CPM	34,141	48,924	479	0
RapidScan	MSPM	75,582	98,060	334	0
(100 Gbps IPS)	CPM	19,323	26,764	271	0
Pigasus	MSPM	128,871	178,185	348	144
(100 Gbps IPS)	CPM	25,478	41,802	340	80

to map efficiently to modern platforms. This limitation is clearly demonstrated when compared against the 100 Gbps IPS variant of RapidScan, which consumes significantly fewer resources than Pigasus at the same speed. At twice the throughput of Pigasus, RapidScan requires only 55% more BRAM (FPGA on-chip SRAM) to support increased hash table parallelism, while its logic consumption remains comparable (within 10%). Furthermore, RapidScan not only scales effectively, but with the addition of field tagging, it successfully supports the novel Log Monitoring workload at a full 200 Gbps line rate.

3.5 Summary

We introduced RapidScan, a High-Level Synthesis (HLS)-based library of high-throughput parameterizable and composable input-dependent string matching kernels. Based on the input-dependent design patterns identified in Chapter 2, RapidScan, effectively leverages the stream processing paradigm in HLS to implement string matching as a parameterizable design template, enabling tuning to

a variety of common-cases and target FPGA platforms. An analysis of the compactor, the most resource-intensive input-dependent structure, demonstrates that through stream-based decomposition and template recursion, HLS achieves RTL-equivalent resource efficiency while offering superior configurability and readability. Consequently, RapidScan seamlessly scales to a 200 Gbps line rate for Intrusion Prevention, consuming significantly fewer resources than an equivalent Pigasus RTL implementation. Furthermore, the inherent productivity of HLS empowers designers to rapidly integrate new functionality, extending RapidScan to provide cost-effective solutions for novel domains like Log Monitoring. RapidDetect, a heterogeneous FPGA-CPU system for Log Monitoring based on RapidScan, delivers near-200 Gbps throughput at a quarter of the cost of Hyperscan, the state-of-the-art software alternative.

Chapter 4

RTL-Native Network-On-Chip Generator

The shuffle design pattern is the most comprehensive input-dependent communication model identified in Chapter 2. Packet-switched Networks-on-Chip (NoCs) provide resource-efficient solutions for this pattern by tailoring the network topology to specific communication characteristics of a given workload [87]. For FPGA accelerators, soft NoCs effectively manage this heterogeneity, enabling customized routing on a per-application or per-workload basis [88–91].

However, implementing NoC routers using High-Level Synthesis (HLS) is not trivial, requiring substantial effort to accurately capture their highly input-dependent dataflows [53]. While Chapter 3 demonstrated that stream-based decomposition can efficiently implement the compaction pattern in HLS without sacrificing developer productivity, NoC-based routing presents a distinct challenge. Moreover, the structure and generality in NoC topologies demand well-known configuration options across application use-cases, placing higher emphasis on performance and

resource efficiency.

To establish a robust baseline for evaluating stream processing-based HLS networks, this chapter introduces ReConnect[1], a novel, state-of-the-art, RTL-native NoC generator. Built entirely in SystemVerilog, ReConnect utilizes a single highly parameterized router module to support an extensive array of configurations and topologies tailored to specific application needs. Furthermore, the modularity of the core router elements, AXI-Stream protocol adapters, and interconnect links enables users to seamlessly integrate existing topologies or build entirely new custom networks.

When evaluating ReConnect-based implementation for a target, common-case optimized database aggregation accelerator, an equivalent HLS implementation consumes over 67% more resources, exposing the practical limits of HLS for representing complex input-dependent patterns.

4.1 Introduction

A state-of-the-art FPGA-optimized soft NoC generator is critical to making a meaningful comparison with hardware optimizations used by HLS toolchains and native support for streaming; however, existing solutions fall short. Legacy FPGA NoC generators like CONNECT [93] are outdated and do not leverage new FPGA architectural features, such as hyper-registers. While the Hoplite family of NoCs [94–96] and FastTrack [97] are newer, they are closed-source and lack support for custom topologies, and ASIC-targeted NoCs, such as Constellation [98] and FlooNoC [99], map poorly to FPGAs [93]. Furthermore, supporting intra-fabric communication re-

[1]ReConnect has been extensively validated using testbenches, internal research projects, and by external users including the development of OpenFPGA-NoC [92]. It is available open-source at https://github.com/shashankov/ReCONNECT

Table 4.1: Comparison of NoC Generators

System	Vintage	FPGA Optimized	Multiple Topologies	AXI-Stream	RTL-Native	Open-Source
ReConnect (ours)	Active	✅	✅	✅	✅	✅
CONNECT [93]	2015	✅	✅	❌	❌	✅
Hoplite [94]	2020	✅	❌	❌	✅	❌
Constellation [98]	Active	❌	✅	❌	❌	✅
FlooNoC [99]	Active	❌	✅	❌	❌	✅

quires prioritizing lightweight, streaming protocols like AXI-Stream, vital to network and streaming accelerators [26, 72, 78, 100] and integration with HLS frameworks, over the bulky memory-mapped interfaces characteristic of ASIC-centric NoCs.

4.1.1 ReCONNECT

We developed ReConnect to fill this gap in open-source FPGA NoC generators. Based on updated principles for FPGA-friendly design, ReConnect's fabric-aware router is optimized to support the high bus widths common to fabric communication patterns [101], stall-free pipelined operation to allow register retiming and harness FPGAs' programmability with aggressive deployment specific optimizations. The credit-based router interface further allows the interconnect links between routers to be freely pipelined leveraging fabric routing features common to modern architectures such as hyper-registers in Altera FPGAs [102]. At the user-interface, ReConnect supports features essential to FPGA workloads, including AXI-Stream [103], customizable clock crossing, and data width conversion.

To ensure seamless integration across hardware design frameworks, we de-

signed ReConnect's implementation to be fundamentally RTL-native. Because nearly all workflows, including High-Level Synthesis (HLS), natively support RTL [104, 105], this approach drastically lowers the barrier to integrating ReConnect with existing designs. Leveraging FPGA compiler toolchain optimizations to supplement System Verilog's limited parameterization and module composition capabilities, ReConnect delivers resource-efficient solutions to a broad-range of topologies and FPGA workloads.

4.1.2 Evaluation

We characterize ReConnect's performance and resource utilization across standard NoC topologies and compare against the open-source NoC generator, CONNECT [93], which comes closest to supporting all the same features. ReConnect achieves an over 60% reduction in resource utilization and 35% increase in maximum operating frequency across various commonly used topologies and router configurations. Overall ReConnect achieves a 9x improvement in iso-resource throughput compared to CONNECT.

Compared with an equivalent HLS implementation for the butterfly topology used in a database aggregation accelerator, ReConnect consumes 38% fewer resources while delivering the same maximum throughput. Furthermore, ReConnect's configurability enables the instantiation of a more efficient butterfly network, trading a 26% drop in saturation throughput for a 3x reduction in resources, suitable for the common-case optimized accelerator.

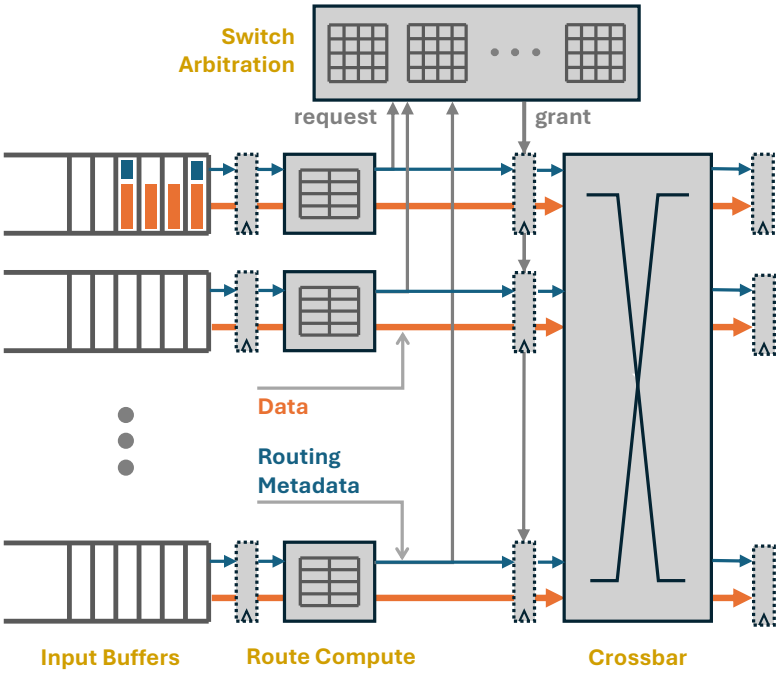


Figure 4.1: Input-buffered router and its various datapath components. The router consists of input buffers, route compute and switch arbitration stages and the data crossbar with optional pipeline stages.

4.2 Background and Motivation

4.2.1 Packet-Switched Network-on-Chips

Packet-switched Networks-on-Chip (simply called NoCs here), as popularized in [106], are on-chip interconnection networks that route packets over shared routing resources. Unlike circuit-switching, packet-switching allows fine-grained sharing of links and routers, leading to resource savings when communication patterns are irregular, bursty, and not statically known ahead of time. Hence, they are extensively used in FPGA-based accelerators that exhibit data-dependent irregular network traffic across spatial compute kernels.

At the heart of NoCs are routers that perform the actual data switching operation as shown in Figure 4.1. Packets first arriving at the router are stored in input buffers which capture backpressure transients from within the router. This is followed by three main stages: (1) Route compute uses the destination information (and possible other internal or NoC state) from the packet to compute which router output port the packet must exit from, (2) Switch arbitration juggles requests from multiple input packets destined for the same output port and fairly allocates the switch resources among the requests, and lastly, the (3) Data crossbar performs the actual switching, routing each input packet to its destination within the router.

These routers are then connected together to form a NoC topology that connects endpoints in the user's design. Complexity in topologies can span from the simplest case of having a single router to more resource-efficient and traffic pattern-specific topologies such as the Butterfly or Torus. The size of the routers and the routing strategy used entirely depend on the needs of the topology, directly affecting the router implementation choices and possible optimizations.

4.2.2 Designing Soft NoCs for FPGAs

Tailoring to FPGAs

Although FPGA fabrics are designed to emulate any custom digital circuit, designs targeting FPGAs must be tailored to the unique features and idiosyncrasies of the fabric to achieve the highest efficiency. As shown in CONNECT [93], FPGA-aware NoC designs can achieve over 50% resource savings when compared to an ASIC-targeted NoC implementation emulated on FPGAs. However, as FPGA architectures evolve [30], it is essential to revisit and update some of the FPGA-oriented design principles put forth by the authors of CONNECT.

With newer process nodes, FPGAs rely on deeper pipelining to enable higher operating frequencies with special routing resources [102] and compiler passes that target pipeline optimization. While not to the extent seen in ASIC designs, modern FPGA NoCs must expose more pipelining options within and between routers to achieve these frequencies, as NoCs span larger die areas. Storage, although still limited on FPGAs compared to the more pervasive logic elements, can now be found in more abundance in certain FPGA families. Although memory-hungry options such as virtual channels remain a bad fit for FPGAs, providing the configurability to choose SRAM-based buffers is essential to balancing utilization across the heterogeneous resource types available in an FPGA fabric [30].

But modern FPGA fabric architectures are not entirely different from their older counterparts. Fabrics continue to be over-provisioned with excess routing resources including wires and support a relatively lower peak frequency compared to peripherals such as memory [30]. Customizable design generators are even more critical to targeting FPGAs to make use of their programmability and stand out from ASICs as the push for specialization has increasingly driven compute to hardware accelerators [107].

We also make a new observation: utilizing vendor-native hard IPs, such as FIFOs, is critical to efficient hardware mapping. Relying on vendor synthesis tools to implicitly infer hard IP from RTL is highly fragile, often failing due to minor code variations or generational changes in the FPGA fabric [108, 109]. In the worst case, uninferred IPs are emulated using soft logic, resulting in orders-of-magnitude increase in resource utilization and severe degradation in maximum operating frequency. Driven by these updated design principles, ReConnect provides a modernized, FPGA-optimized NoC generator equipped with a robust suite of fabric-aware

features.

User Interfacing

The usability of a NoC generator depends on providing and optimizing for interfaces most commonly used by end-user designs. Most ASIC NoCs designed to support SoCs target memory traffic and hence provide memory-mapped interfaces such as AXI-4 [98, 99]. Hard NoCs in modern FPGAs satisfy a similar role, while soft NoCs must carry inter-kernel traffic over latency-insensitive links using streaming interfaces [110]. Existing ASIC NoC generators, such as Constellation [98] do not support a streaming interface such as AXI-Stream out-of-the-box, making them unsuitable for use with such FPGA designs. ReConnect's modular instantiation allows users to select between the bare router interface or use the AXI-Stream adapters for easy integration.

Clock-crossing is another typical requirement for NoC interfaces, allowing the NoC to run at higher frequencies unburdened by external constraints. Although this is common to both ASIC and FPGA NoCs, clock crossing cannot be easily represented in vendor-agnostic RTL [111] and must use vendor-native IPs or templates to achieve clock-crossing and the associated data-width conversion efficiently. In the least, a NoC generator must expose modular clock-crossing buffers to enable users to replace them with their device-specific instances. ReConnect provides behavioral models along with IPs for the major FPGA vendors in a modular FIFO wrapper enabling this customization.

4.2.3 Hardware Design Frameworks

To make hardware design more approachable and modular, frameworks such as Bluespec [41], Chisel [42], and High-Level Synthesis (HLS) [46, 68] offer advanced language features, automatic circuit optimization, and integrated testing for hardware design. HLS exposes the highest level of abstraction, interpreting sequential C/C++ as hardware using code-transformation pragmas and compiler analysis to generate hardware. Delivering a massive productivity boost for input-independent computation kernels and simple dataflows, HLS rapidly loses its advantage for fine-grained kernels such as NoC routers. Careful design as shown in [53], can allow RTL-comparable HLS NoC implementations, but this leaves the code as or even less maintainable than equivalent RTL. Moreover, HLS sources or outputs, finely tuned to a particular HLS toolchain, are not easily portable across vendor offerings due to toolchain idiosyncrasies, license restrictions, vendor-specific IP usage, and target FPGA-specific optimizations.

Bluespec [41] and Chisel [42] provide vendor-agnostic programming abstractions closer to hardware, enabling tighter control over the performance and resource-utilization of the generated RTL. We opt for an RTL-native approach, to allow ReConnect to be truly plug-and-play with any design framework, removing the additional step of wrangling with a new and potentially old RTL generator. RTL languages such as SystemVerilog however do not support as rich a set of features for modular instantiation and parameterization. We exploit the compiler passes in modern FPGA toolchains, such as dead-code elimination and constant propagation [112], to optimize the NoC datapath beyond SystemVerilog's parameterization capabilities. This allows ReConnect to provide a wide variety of customization options, topologies, and hardware optimizations, despite being limited to SystemVerilog.

4.3 One Router to Rule Them All

ReConnect's router is central to fulfilling the goals of being parameterizable, FPGA-friendly and performant. In this section, we describe the router architecture, resource optimizations and parameterization options built into it.

4.3.1 Router Architecture

Flow Control and Backpressure

ReConnect implements wormhole flow control/switching [113] to make efficient use of the limited buffering space available on the FPGA fabric. Unlike CONNECT [93] which features peek flow control, ReConnect's router implements a credit-based interface. While peeking allows routers to directly observe each other's buffer occupancy, it can lead to degradation in operating frequency (fMAX) when the NoC stretches over a large FPGA fabric. Credit-based flow control enables decoupling of connected routers and allows the insertion of a variable number of pipeline stages (upto the buffer depth) over the interconnect links between the routers to improve fMAX.

Flit Buffering

ReConnect uses simple Input Queueing to save storage resources compared to other techniques such as Virtual-Output Queueing [93]. Although ReConnect does not support virtual channels, an accelerator that requires a deadlock-free response channel can easily instantiate a secondary lower bitwidth response network which can reduce storage consumption on the FPGA compared to an over-provisioned virtual channel implementation. Leveraging the programmability of FPGAs, a single NoC instance does not need to support all possible traffic patterns; instantiations

can be tailored to provide resource-efficient solutions on a case-by-case basis.

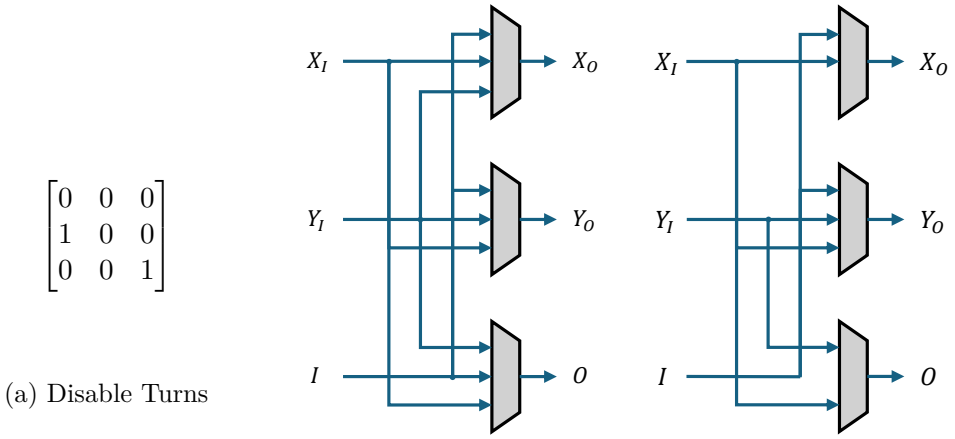
Based on the observation that FPGAs have an over-provisioned routing fabric with an abundance of wires, we chose to implement the packet header as a separate parallel interface instead of occupying its own flit and cycle through the router. This allows us to reduce latency and dedicate tailored resources to the header channel, including narrower buffers.

Routing Strategy

We implement deterministic routing based on read-only routing tables stored in each router, which are directly embedded as FPGA LUTs. In the Route Compute stage, the packet header is used to probe the table for the output port the packet must use to exit the router. Although deterministic routing is more restrictive, adaptive techniques can be exceedingly complex to implement in hardware and primarily benefit worst-case traffic patterns [113]. Since FPGAs allow customization, despite being deterministic, both the routing strategy and NoC topology can be updated per-application or workload, effectively mitigating these worst-case scenarios. Furthermore, deterministic routing guarantees in-order packet delivery between endpoints, eliminating the need for costly reordering logic at the receiver.

Switch Arbitration

Arbitration logic plays an important role in ensuring fairness, which in turn affects the NoCs saturation throughput. Logic blocks in modern FPGAs have grown larger with intra-CLB routing, which is capable of easily handling more complex logic effectively. For ReConnect, we implement the matrix arbiter [113] which maps well to these FPGA logic resources for the relatively small input/output ports seen in NoCs and provides strong fairness guarantees improving the saturation bandwidth



(a) Disable Turns

(b) Full Crossbar

(c) Pruned Crossbar

Figure 4.2: The disable turns matrix is used to achieve pruning in a directional torus topology router using dimension-ordered routing. This disables paths connecting the input directly to the output (I→O; endpoint sending data to itself) and the Y→X turn which is not allowed in dimension-ordered routing.

by close to 20%. The arbitration logic also holds its grants for the entire duration of a packet to avoid interleaving of flits from different packets, eliminating the need for reassembly logic at the receiver.

Crossbar Switch

The crossbar switch is implemented to support full bandwidth between independent input-output pairs using independent muxing logic per output port. Along with the input buffers, the crossbar switch is one of the most resource intensive components of the router.

4.3.2 Micro-architectural Optimizations

Disabling Turns

While in a generic router all pairs of input-output traversals are possible, when used in a topology along with a specific routing strategy, some of these pairs are unused. Without any optimization, the router would be over-provisioned and waste resources, especially in the crossbar, as shown in Hoplite [94] and [114]. However, hard-coding this pruning of crossbar paths is not a viable solution if the router is to be parameterizable and reusable across topologies and routing.

To achieve pruning in a parameterizable manner directly from RTL, we rely on FPGA compiler toolchain's constant propagation and dead-code elimination [112]. The router takes as input a boolean `NUM_INPUTS x NUM_OUTPUTS` matrix `DISABLE_TURNS` which specifies which input-output pair connections should be disabled in the router. This matrix is used to mask the select bits, zeroing out the irrelevant inputs in each output port's crossbar multiplexer logic. The compiler propagates this constant and eliminates any logic affected by it, allowing us to deliver the same reduction in resources without hard-coding the pruning.

An example for the directional torus topology is shown in Figure 4.2. The disable turns matrix produces the pruned crossbar structure after dead-code elimination.

Pipelining and Stall-Free Retiming

Pipelining is crucial to reaching higher operating frequencies in modern FPGAs. Since not all router sizes or situations would require pipelining, in ReConnect we implement pipelining as parameterizable options after each stage of the router, as shown in Figure 4.1.

Register Retiming [115, 116] in the FPGA compilation flow are powerful techniques in balancing combinational paths across registers. But retiming can fail in the presence of feedback paths such as combinational backpressure/stall signals. To harness retiming, we push the credit-based flow control into the router all the way to the route compute stage. This allows flits to flow stall-free through the router once the credit check passes, allowing retiming of the registers within the router critical paths.

Delayed Flit Dequeue

Although pipelining helps improve fMAX, it also leads to increased register usage, especially for wide flit widths. Since only the packet header is needed to drive routing decisions, pipelining the data flit is unnecessary. To reclaim the pipeline registers used for data flits, we delay the dequeue signal to the input buffer until the flit is ready to traverse the switch. This allows the routing and arbitration stages to be pipelined for higher fMAX while data flits simply bypass these stages and directly arrive at the crossbar.

4.3.3 Router Parameterization

Here we summarize the parameterization options available in ReConnect's router module:

- *Flit and Header Width*
- *Flit Buffer Depth:* Minimum depth is decided by the chosen router pipelining options and interconnect pipelining.
- *Number of inputs/outputs:* Each router instantiation can be customized to

have a parameterized number of input and output ports. Crossbar uses the AND-OR structure to generate parameterized one-hot multiplexers.

- *Routing Tables:* Externally specified using memory hex files (generated using provided or custom python scripts).
- *Pipelining Options:* Independent options for pipelining Route Compute, Arbiter and Output stages.
- *Memory Resource Type:* Option to force LUT-based RAM use for flit and header buffers.
- *Disable Turns:* Optimize crossbar paths based on routing strategy and topology structure.

4.4 Building Topologies

With the extensive options provided by ReConnect's router, building a topology is as simple as instantiating and connecting appropriately sized routers using generate constructs available in SystemVerilog. For each topology, ReConnect instantiates routers with the minimum number of input and output ports needed for each router position in the topology. We also provide pipelining options for the links between the routers to allow the NoC to stretch across endpoints placed far apart in the FPGA fabric. The pipeline registers are designed to operate reset-free, enabling the full extent of retiming optimization available in the FPGA toolchains. Figure 4.3 shows a section of a mesh topology, highlighting the variable number of input/output ports across routers and link pipelining.

Agentic AI tools such as Claude Code [117] or Antigravity [118] make custom topology generation even more accessible. Using the templates from topology

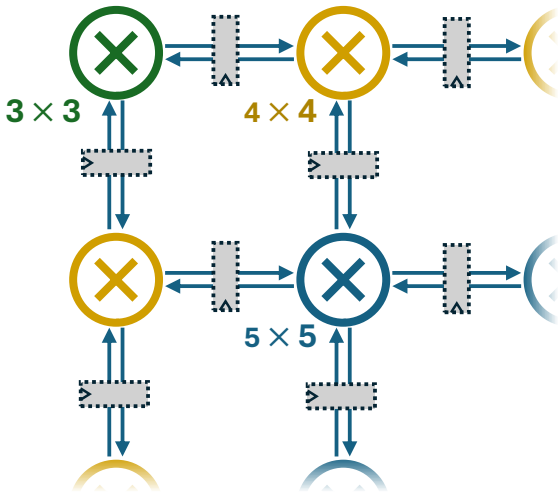


Figure 4.3: Section of a mesh topology showing routers with different number of input/output ports and optional pipeline stages along the links between the routers. Endpoint connections are not shown.

presets available in ReConnect, new topologies and their associated routing tables can be easily generated using natural language prompts. The provided regression testing suite ensures that the generated NoC can be verified for correctness. Two of the topologies in the open-source repository (Fat Tree and Fully Connected), were generated by Antigravity using the Gemini 3.5 Flash model.

4.5 Interfacing with ReConnect

ReConnect provides an optional AXI-Stream wrapper around the credit-based interface of the routers. The wrapper instantiates the necessary buffers to handle credit signaling, converting the interface into a simple ready-valid protocol. Other signals in the AXI-Stream protocol such as TID/TUSER are concatenated into the data flit and unpacked at the receiving end. The AXI-Stream shims also offer optional serialization and deserialization, alongside clock-domain crossing and

simultaneous data-width conversion, implemented using vendor-native IP cores. This comprehensive feature set grants user designs the flexibility to integrate with ReConnect at their optimal operating points, effectively decoupling the constraints of the user application from those of the NoC.

4.6 Evaluation against CONNECT

To showcase the effectiveness of our RTL implementation and parameterization, we evaluate ReConnect across various topology and configuration options against CONNECT [93] over latency vs throughput performance, fMAX (maximum operating frequency) and resource utilization.

4.6.1 Methodology

Default Parameterization

Unless mentioned otherwise, configurations for both CONNECT and ReConnect set number of endpoints to 16, flit buffer depth to 8, flit width of 256 bits, and use no virtual channels (set to 1 for CONNECT). When testing pipelining, we always set all available options within the routers.

Load-Latency Curve

We implemented an open-loop traffic generator in cycle-accurate RTL simulation using Verilator [119] to drive the AXI-Stream wrapped NoC with varying traffic patterns. Uniform random sends packets to any endpoint with equal probability, while the unbalanced neighbor traffic pattern sends a specified fraction (unbalance factor) of packets to nearest neighbors and distributes the rest randomly to other endpoints. The average packet latency is measured from the time the first flit is

enqueued in the source buffer until it arrives at the receiver. For all measurements, we run the simulation over the same number of total packets to ensure a uniform basis for averaging with all pipeline options disabled for a fair comparison with CONNECT.

Synthesis

We synthesize ReConnect using Altera/Intel Quartus 23.2 for an Agilex 7 Speed Grade 1 device (AGIB027R29A1E1VB) which represents the latest and fastest FPGA device available to us. The tool is set to its default compilation options, virtual pins on all its I/Os to ensure appropriate timing analysis, and a target clock frequency of 1 GHz. We also set both ReConnect and CONNECT to use only LUT-based RAM (called MLABs in Altera FPGAs) if possible for the flit buffers and measure resource utilization as the number of ALMs (unit of soft logic in Altera FPGAs). In the synthesis runs, both CONNECT and ReConnect are compiled in isolation without any floorplanning constraints to study them under their best case scenario.

4.6.2 NoC Latency/Throughput Performance

To evaluate the performance of ReConnect, we plot the load-latency curve over four 16-endpoint topologies used in the CONNECT paper [93] and two traffic patterns, Uniform Random (Figure 4.4) and 90% Neighbor Unbalanced (Figure 4.5). The Fully Connected topology uses high-radix routers to provide one-hop links between any two endpoints equivalent to the High Radix topology in CONNECT.

For Uniform Random traffic in Figure 4.4, the Fully Connected topology achieves the lowest average latency while supporting the highest saturation through-

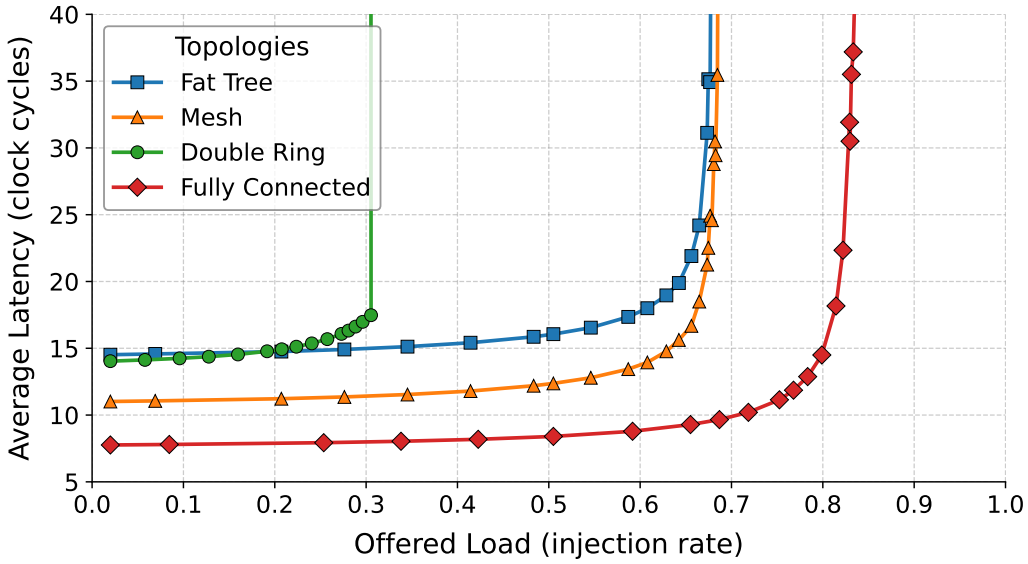


Figure 4.4: Load-Latency Plot for ReConnect under Uniform Random traffic pattern across various topologies.

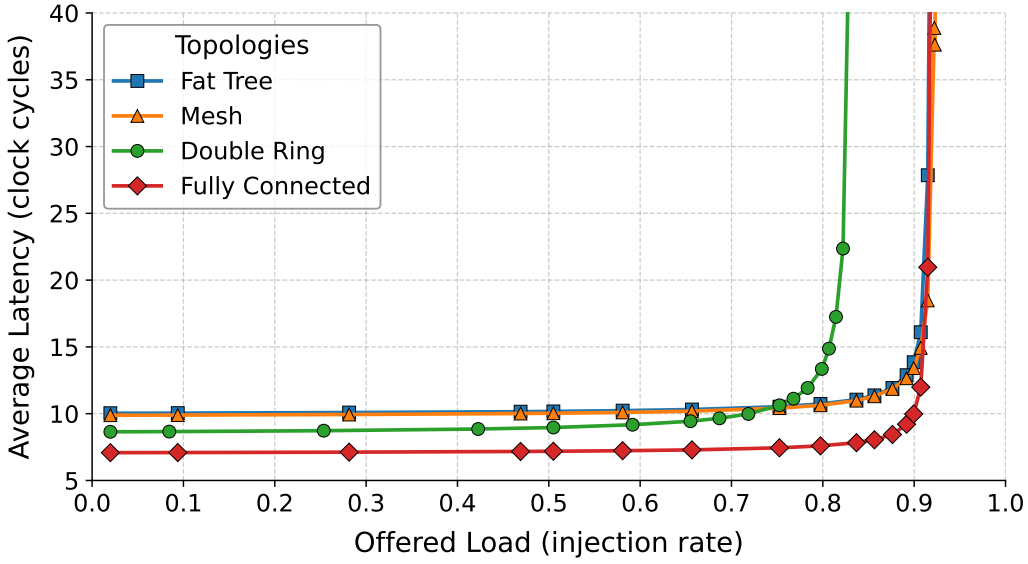


Figure 4.5: Load-Latency Plot for ReConnect under 90% Neighbor Unbalanced traffic pattern across various topologies.

Table 4.2: Effect of pipelining on resource consumption and fMAX of ReConnect's router across various sizes.

Size	ALMs		fMAX (MHz)	
	Comb.	Pipelined	Comb.	Pipelined
2x2	1,393	(+8.2%) 1,507	593	(+26.1%) 748
3x3	2,454	(+9.3%) 2,681	509	(+30.9%) 667
4x4	5,466	(-31.8%) 3,728	386	(+52.1%) 586
5x5	4,879	(+11.3%) 5,428	395	(+27.8%) 505
6x6	13,200	(-45.0%) 7,261	326	(+42.4%) 464

Comb.: Combinational router core does not use any pipelining options
Flit Buffer Depth = 8 | Flit Width = 256 bits | Force MLAB (uses LUTRAM instead of SRAM)

put owing to its one-hop paths between endpoints and high bisection bandwidth. Double Ring saturates and begins to deadlock at relatively lower loads due to its circular topology and minimal connectivity across low-radix routers. Mesh and Fat Tree provide a middle ground with similar saturation throughputs of 0.7, but higher low-load latency than the Fully Connected topology.

Figure 4.5 plots equivalent curves for a Neighbor Unbalanced traffic pattern where each endpoint sends 90% of the traffic to its nearest neighbors. All topologies see significant improvement with Double Ring reaching over 80% of injection rate with lower low-load latency than the more expensive Mesh and Fat Tree networks. With high nearest neighbor traffic, the Mesh and Fat Tree and Fully Connected topologies deliver similar saturation throughputs at just over 90% of injection rate.

4.6.3 Resource Utilization and Operating Frequency

Table 4.2 demonstrates that enabling pipelining yields an outsized benefit to router frequency relative to its minimal, and occasionally negative, resource overhead. Across most router sizes, we observe only a modest 10% increase in resource utilization, highlighting the efficiency of the delayed dequeue mechanism detailed in

Section 4.3.2. Counterintuitively, for 4x4 and 6x6 configurations, the purely combinational router core is more resource-intensive than its pipelined counterpart. This anomaly can be attributed to logic duplication in the FPGA toolchain, which optimizes the critical path in the unpipelined design. Pipelining consistently delivers over a 25% increase in fMAX, with the 4x4 configuration peaking at over 50% improvement and the smallest router configurations achieving operating frequencies close to 750 MHz.

Table 4.3 compares the resource utilization and maximum operating frequency (fMAX) of fully pipelined ReConnect and CONNECT implementations across various router sizes and NoC topologies. For all but the largest router configurations tested, ReConnect consumes approximately 50% fewer resources while achieving up to a 20% increase in fMAX. This performance gap widens significantly for mesh and directional torus topologies, where ReConnect achieves roughly 60% resource savings alongside a 30 to 50% frequency improvement. CONNECT's higher resource footprint stems partially from its FIFO implementation, which cannot be inferred as Agilex 7 MLAB (LUTRAM) blocks. ReConnect further reduces resource consumption by optimizing for dimension-ordered routing in mesh-like topologies using crossbar pruning. When allowing on-chip SRAM usage, we still see a 20% decrease in resource usage (ALMs) compared to CONNECT with both designs using the same number of memory blocks; the operating frequency for both remains the same as their LUTRAM equivalents.

Table 4.3: Resource Utilization and Maximum Frequency (fMAX) comparison between ReConnect and CONNECT across various router and NoC configurations.

Topology	**Size**	**ALMs**		**fMAX (MHz)**	
		CONNECT	**ReConnect**	**CONNECT**	**ReConnect**
Router	2x2	3,151	(-52.2%) 1,507	730	(+2.5%) 748
	3x3	4,953	(-45.9%) 2,681	591	(+12.7%) 667
	4x4	7,108	(-47.6%) 3,728	495	(+18.5%) 586
	5x5	10,615	(-48.9%) 5,428	411	(+22.9%) 505
	6x6	11,741	(-38.2%) 7,261	378	(+22.9%) 464
Butterfly	2-ary 4-fly	82,557	(-65.4%) 28,548	400	(+57.5%) 630
Mesh	4x4	98,067	(-66.5%) 32,901	325	(+57.8%) 512
Mesh (Pipelined Links)	4x4	104,917	(-66.0%) 35,718	410	(+35.6%) 556
Directional Torus	4x4	66,490	(-58.2%) 25,252	361	(+56.5%) 565
Directional Torus (Pipelined Links)	4x4	68,043	(-58.6%) 28,137	503	(+26.8%) 638

Flit Buffer Depth = 8 | Flit Width = 256 bits | Force MLAB (uses LUTRAM instead of SRAM) | All Router Pipeline Options

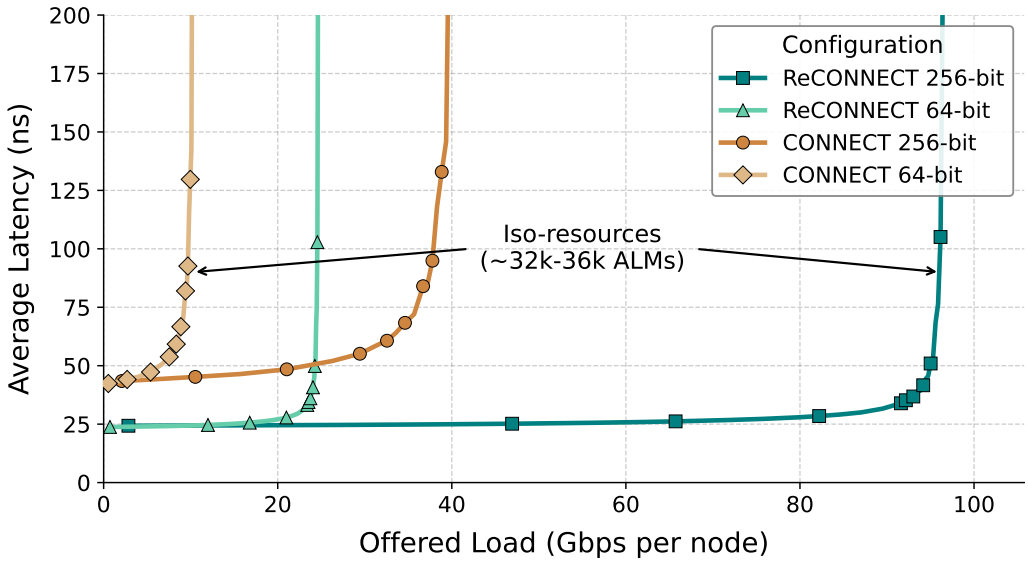


Figure 4.6: Load-Latency curves comparing fully-pipelined mesh configurations operating at their respective maximum frequencies

We also compare the effect of pipelining the links between the routers in mesh-like topologies. Contrary to CONNECT's argument that FPGA NoC routers should be tightly coupled [93], even when the NoCs are not floorplanned, Table 4.3 shows that pipelining the links yields frequency gains, underscoring the importance of revisiting design principles as architectures change. CONNECT sees a 25% jump while the higher baseline frequency of ReConnect increases by around 10%. We expect this gap to widen as designs grow larger, pulling the routers further away from each other.

4.6.4 Latency and Throughput Comparison with CONNECT

Figure 4.6 plots the load-latency curves using the frequency and bitwidths to translate the load fraction and latency into real-world metrics of Gbps and nanoseconds, respectively. ReConnect's 256-bit mesh implementation consumes approximately

the same resources as a 64-bit implementation in CONNECT, while achieving higher saturation load (as fraction of injection rate) and 30% higher frequency. Putting these gains together, iso-resources, ReConnect sustains a 9x higher throughput than CONNECT and close to 50% lower low-load latency.

To study the synthesis-agnostic saturation throughput performance, Figure 4.7 plots the saturation throughputs as a fraction of maximum injection rate. The matrix arbiter used in ReConnect provides stronger fairness guarantees compared to the round-robin scheme used in CONNECT, sustaining higher loads. ReConnect outperforms CONNECT by over 15% for the Mesh and Fully Connected topologies and by over 20% for the Fat Tree topology under uniform random traffic. Double Ring performs similarly since the saturation occurs due to deadlocks and is not affected by the arbitration scheme.

We see larger gains under the unbalanced traffic pattern for Double Ring, Fat Tree and Mesh topologies. Double Ring reaches its true saturation limit due to the smaller traffic fraction traversing long deadlock-prone paths. The Fully Connected topology faces even lower contention with the unbalanced traffic with the increased fairness in arbitration only delivering a marginal improvement.

4.7 Evaluation against HLS

Having established the performance and resource efficiency of ReConnect, in this section, we evaluate ReConnect against an equivalent implementation of the butterfly topology in HLS, based on the communication pattern seen in a common-case optimized database accelerator.

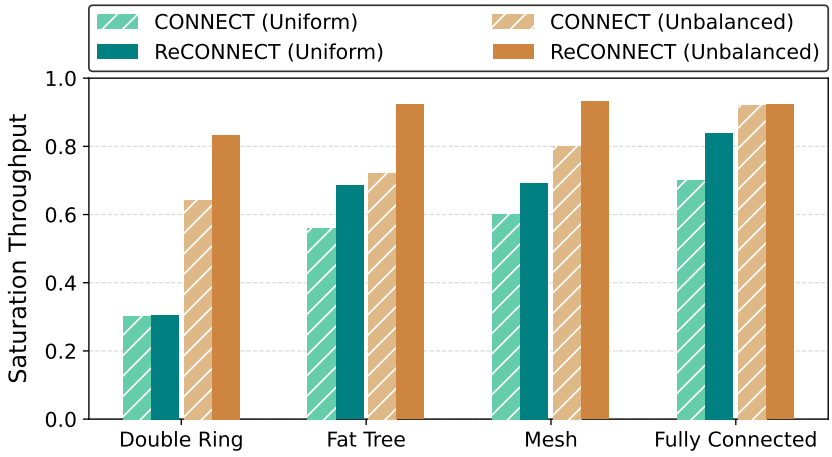


Figure 4.7: Comparison of saturation throughputs as a fraction of maximum injection rate between CONNECT and ReConnect for Uniform Random and 90% Neighbor Unbalanced traffic patterns.

4.7.1 Common-Case Optimized Database Accelerator

Group-by aggregation (or just aggregation) is one of the most widely-used instances of aggregation kernels in data analytics [10, 120, 121]. This operation condenses an input stream of raw `<key, value>` tuples into an output stream whose length equals the number of distinct attributes, by applying an aggregate function (e.g., summation) to combine the values of tuples with the same attribute. For example, aggregation can be used by a business to extract trends in revenue by product. The database recording every transaction may include two columns: one for the Universal Product Code (UPC) and one for the transaction's dollar amount. The aggregation then computes the total revenue (the measure value) per UPC (the grouping attribute).

Optimizing for the common-case in aggregation is a standard technique in software, motivated by the data skew prevalent in many database tables [122]. In these workloads, a small fraction of keys occur with a disproportionately higher

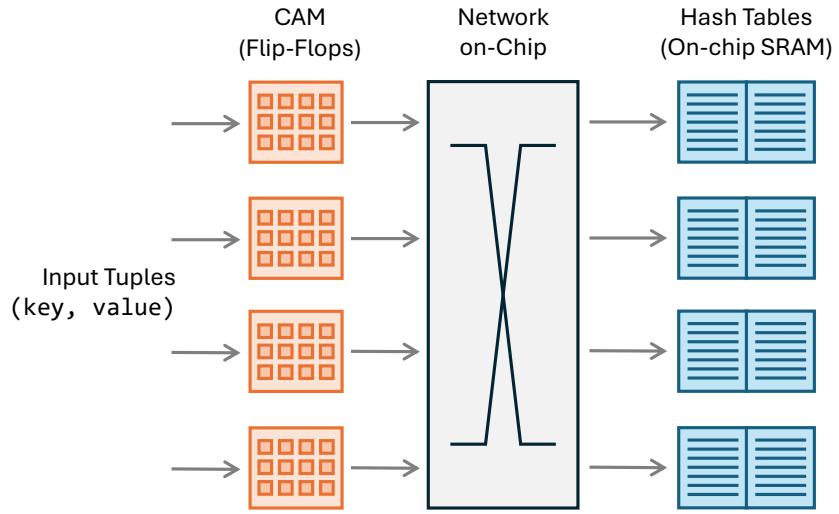


Figure 4.8: Common-case optimized FPGA Group-by Aggregation accelerator uses hierarchical aggregation blocks based on CAMs (Content-Addressable Memories) and Hash Tables to efficiently utilize the heterogeneous memory resources available on the FPGA. A NoC is used to partition the input key domain into disjoint hash tables to increase the total capacity of the hash table array.

frequency, accounting for the vast majority of records. For example, in the sales database, the UPCs of more popular items are expected to appear more frequently than the UPCs of less common products. To exploit this data locality, software implementations typically map these frequently accessed "hot" keys to local, per-core hash tables designed to fit entirely within the L1 cache, while directing rarer keys to larger, secondary backend tables [9, 123, 124].

FPGA-based accelerators can exploit the heterogeneous memory types available in the fabric, ranging from registers and distributed on-chip SRAM to High-Bandwidth Memory (HBMs) and DRAM [125]. Figure 4.8 illustrates the architecture of an FPGA-accelerated streaming group-by aggregation pipeline. The first stage comprises small, register-based Content-Addressable Memory (CAM) aggregation kernels. These kernels absorb frequently occurring keys while allowing rarer

keys to seamlessly overflow into the subsequent stage. These overflow key-value pairs are then captured by a second stage of hash tables, which leverage the significantly larger capacity of the distributed on-chip SRAM. The design achieves spatial parallelism by concurrently processing multiple key-value pairs across parallel instantiations of these CAMs and hash tables. Furthermore, drawing on the Partition and Aggregate strategy from [123], the accelerator uses an all-to-all network to route keys into disjoint hash tables, eliminating key duplication across parallel tables, effectively scaling the total capacity with the number of parallel units.

The common-case for this accelerator is tied to the data skew within the input key-value stream, a characteristic that can vary significantly across different databases. Under high-skew conditions, the small set of hot keys can be efficiently filtered by the CAM stage, drastically reducing the traffic forwarded to the NoC. To efficiently support the broad design space of possible input skews, the NoC must be compile-time configurable, enabling diverse topologies and flexible performance-resource trade-offs. For this evaluation, we use a radix-2 unidirectional butterfly topology for the relative simplicity of its 2×2 routers and the high bisection bandwidth, while providing enough richness in router configurations to perform a comprehensive comparison.

4.7.2 HLS Butterfly NoC Implementation

As illustrated in Figure 4.9, the radix-2 butterfly topology comprises cascading stages of 2×2 routers interconnected in a standard butterfly permutation. Implementing this 2×2 router directly in HLS demands precise scheduling and management of intermediate signals, to realize a fully pipelined architecture capable of sustaining maximum input bandwidth under contention-free conditions [53]. How-

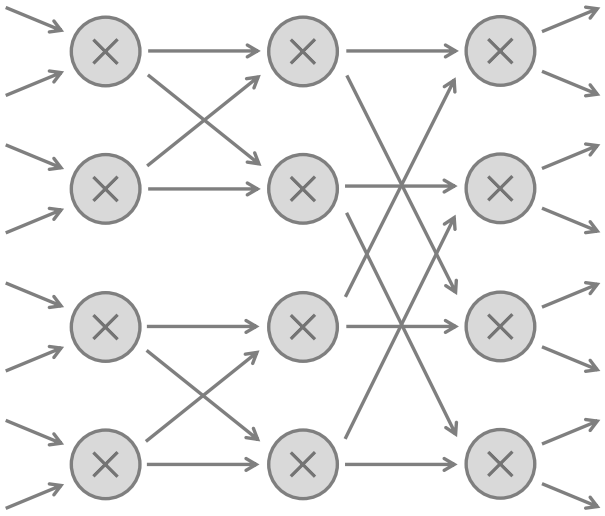
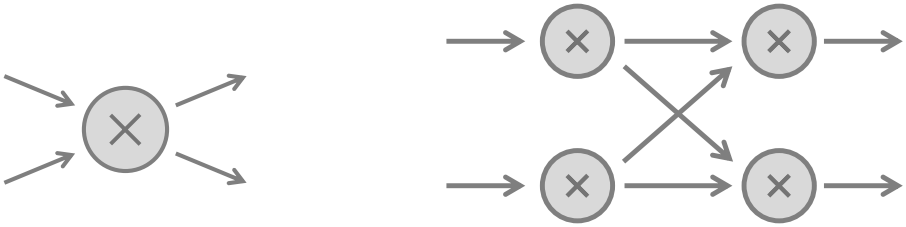


Figure 4.9: An 8 × 8 radix-2 Butterfly Network



(a) Canonical 2 × 2 router

(b) 2 × 2 router implemented using the fork-join design pattern

Figure 4.10: Radix-2 Butterfly 2 × 2 router HLS implementation uses stream-based decomposition to simplify kernels into having only one input or one output.

ever, this comes at a steep cost to developer productivity. Dedicating months of engineering effort simply to circumvent HLS toolchain idiosyncrasies renders the HLS approach fundamentally untenable, resulting in brittle designs that resist future architectural modifications.

Instead, we apply our stream-based decomposition approach to restructure the 2 × 2 router into a modular fork-join dataflow, as illustrated in Figure 4.10b.

```
void router_1to2() {
  [[intel::initiation_interval(1)]] while (1) {
    auto item = TInPipe::read();

    // Get the bit corresponding to the current layer
    unsigned output_idx = (item.idx >> (LAYER_IDX * 1)) & 0x1;
    if (output_idx == 0) {
      TOutPipe0::write(item);
    } else {
      TOutPipe1::write(item);
    }
  }
}
```

Listing 4.1: Stage 1 of the 2 × 2 router implementation in Altera/Intel HLS implements the output port dependent fork

In the first stage, each input is steered (forked) into one of two dedicated streams corresponding to its target output port. The subsequent stage executes a join operation, merging these decoupled streams at each respective output. This arrangement, also known as Virtual Output Queueing (VOQ) [87], maps effortlessly to HLS. As demonstrated in Listings 4.1 and 4.2, our approach requires only concise, highly readable kernels, in contrast to the verbose codebase reported in [53]. Furthermore, this technique scales seamlessly to arbitrary-radix routers, facilitating the development of diverse and complex network topologies similar to ReConnect.

4.7.3 Configurations

To evaluate ReConnect against our Altera/Intel SYCL HLS [46] implementation within a unified environment, we constructed the testbench in HLS and imported ReConnect as a custom RTL library [104]. ReConnect's native streaming interfaces map directly to the ready-valid handshake protocol required by the HLS RTL wrapper, facilitating seamless integration.

Our evaluation targets a 2-ary 4-fly butterfly network connecting 16 end-

```
void router_2to1() {
    TFlit inFlit[2];
    bool valid[2] = {false};
    bool priority = false;

    [[intel::initiation_interval(1)]] while (1) {
        priority = !priority;

        if (!valid[0]) inFlit[0] = TInPipe0::read(valid[0]);
        if (!valid[1]) inFlit[1] = TInPipe1::read(valid[1]);

        // Round-Robin priority implementation
        bool pick0 = valid[0] && (!valid[1] || priority);
        if (valid[0] || valid[1]) {
            TFlit writeFlit = pick0 ? inFlit[0] : inFlit[1];
            TOutPipe::write(writeFlit);
            if (pick0) valid[0] = false;
            else valid[1] = false;
        }
    }
}
```

Listing 4.2: Stage 2 of the 2 × 2 router implementation in Altera/Intel HLS implements the join with Round-Robin arbitration

points via radix-2 routers. For the ReConnect implementation, we evaluate two configurations: (1) a standard Input-Queued design and (2) a Virtual Output Queued (VOQ) configuration. This VOQ setup mirrors the architecture shown in Figure 4.10b, leveraging ReConnect's configurable routers to establish a strict iso-throughput comparison against HLS. For the equivalent HLS implementation, we explore two arbitration strategies: standard Round-Robin and a fixed-priority scheme that simplifies logic by consistently favoring the first input in case of routing conflicts.

4.7.4 Experiment Setup

We target the Terasic DE10-Agilex Accelerator Card [126], which hosts an F-Tile Agilex 7 (AGF014) Speed Grade-2 device. The Agilex 7 card has 487,000 ALMs, 139

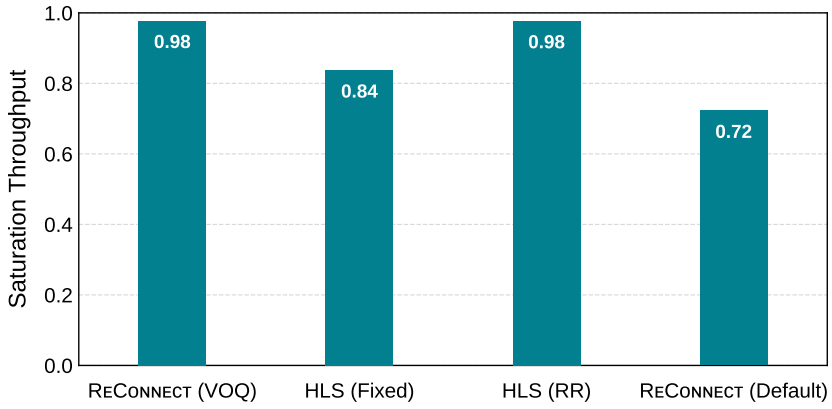


Figure 4.11: Saturation Throughput across ReConnect (Virtual Output Queued (VOQ) and default Input Queued) and HLS-based (Fixed Priority and Round-Robin) Butterfly implementations as a fraction of peak injection rate.

Mb in on-chip SRAM and 4x8 GB off-chip DDR4 memory. Intel oneAPI Compiler version 2023.2, Quartus Pro 21.2 along with the OpenCL BSP provided by Terasic are used for HLS compilation, FPGA synthesis, and deployment, respectively. All the configurations synthesize to 500 MHz, limited by the surrounding logic and HLS platform's maximum frequency. Resource utilization is measured in ALMs, the fabric logic (LUTs and registers) unit in Altera/Intel FPGAs. To evaluate saturation throughput, the maximum steady-state throughput served by a configuration, we inject uniform random traffic representative of the aggregation workload.

4.7.5 Results

Figure 4.11 plots the saturation throughput as a fraction of the maximum injection rate (16 flits/cycle for the 16 endpoints), Figure 4.12 details the normalized resource utilization, and Figure 4.13 presents the same data as a design space. ReConnect's VOQ implementation achieves the exact same saturation throughput as the Round-Robin HLS design, confirming their equivalence, while consuming 35% fewer re-

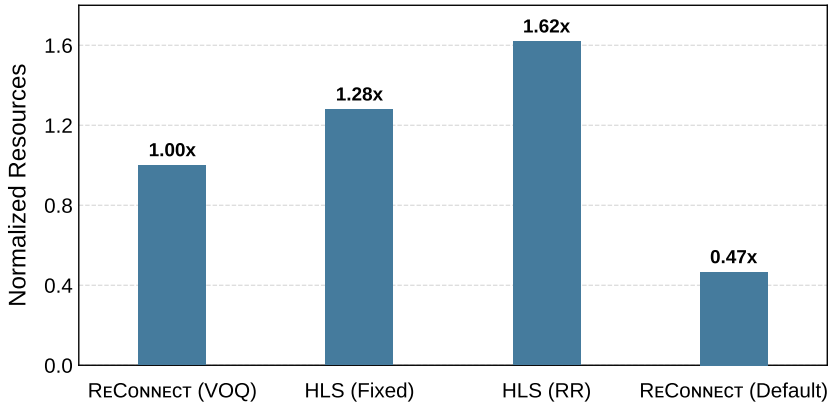


Figure 4.12: Resource utilization across ReConnect (Virtual Output Queued (VOQ) and default Input Queued) and HLS-based (Fixed Priority and Round-Robin) Butterfly implementations normalized against ReConnect's VOQ implementation.

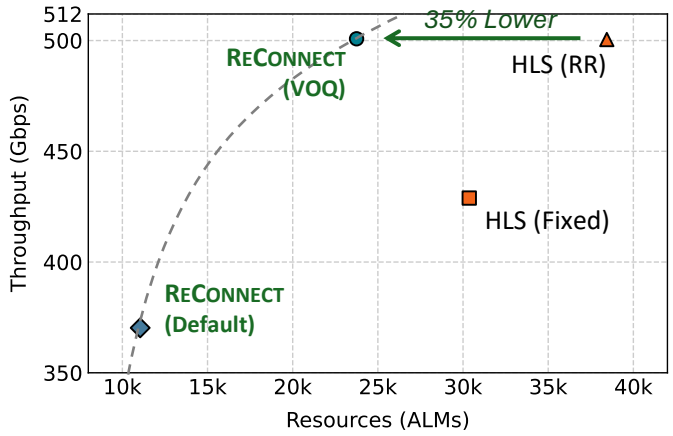


Figure 4.13: Throughput vs Resource Utilization Pareto Plot across ReConnect and HLS-based Butterfly implementations. ReConnect pareto dominates HLS-based implementations providing a more useful tradeoff between performance and resource consumption.

sources. Although the fixed-priority HLS arbitration saves some resources at the expense of throughput, it remains pareto-dominated by ReConnect's VOQ configuration. Furthermore, ReConnect offers an alternative pareto-optimal design point through its default Input-Queued configuration. Requiring only one-third the resources of the HLS implementation and only half those of ReConnect's VOQ implementation, this configuration achieves a 26% lower saturation throughput. This trade-off can be extremely effective in the target aggregation accelerator if, in the common-case, the CAMs filter even 30% of the incoming traffic, reducing the load on the NoC to what can be supported by the Input-Queued version.

4.8 Summary

We presented ReConnect, a novel open-source RTL-native NoC generator for the modern FPGA. ReConnect builds off an updated set of design principles for FPGA-friendly NoC design, targeting intra-fabric streaming communication patterns, deeper pipelining within and between routers, the larger logic capacity in CLBs. ReConnect also enables aggressive topology specific optimizations leaning into FPGAs' programmability to deliver a significant reduction in resource usage. Furthermore, to enable seamless framework-agnostic integration with existing accelerator designs, a single highly parameterized router implementation in System Verilog drives ReConnect's NoC generation capability. Compared to CONNECT, ReConnect achieves an improvement of over 9x in NoC throughput powered by a 60% reduction in resource utilization and a simultaneous 30-60% increase in fMAX, reaching over 550 MHz for mesh-like topologies.

Through our evaluation of ReConnect, we further demonstrated the limitations of HLS in representing complex input-dependent design patterns. For

all-to-all communication, HLS implementations force a compromise between development productivity and hardware efficiency. RTL implementations, such as ReConnect, offer better reusability and maintainability, while remaining highly-configurable and seamlessly integrating into existing HLS frameworks. Although the stream processing paradigm can reduce the HLS implementation effort, the resulting networks are pareto-dominated by ReConnect, with ReConnect exposing a broader performance-resource tradeoff space, essential for deploying common-case optimized accelerators on FPGAs.

Chapter 5

Queueing-based Performance Modeling, Simulation and Tuning

Deployment-specific customized instantiations are critical to maximizing resource efficiency for input-dependent streaming pipelines whose performance varies significantly with input workload [11, 12, 22, 127, 128]. FPGAs are uniquely positioned to accelerate such input-dependent applications, combining inherent programmability (through reconfiguration) with hardware-class performance and efficiency [31, 125]. Design-space exploration (DSE) tools, central to generating these custom instantiations, rely on models that capture the performance-resource tradeoffs of High-Level Synthesis (HLS)-based design generators.

In this chapter, we present RapidQ, a systematic approach to modeling and tuning input-dependent streaming pipelines at deployment time for FPGAs[1].

[1]This work was done in collaboration with Bin Li from Intel Corporation and is set to appear in IISWC 2026 [129]

RapidQ leverages the structure of high-performance streaming pipelines in HLS, to model them as queueing systems [62], where buffers are represented as queues and processing modules as servers, abstracting the functionality within. We employ functional software simulation, once per trace, to quantify the load each input data element generates for every server and the storage space required in the queues. By abstracting away detailed functional logic while retaining the streaming structure, the model remains lightweight, operating at the queue-server boundaries without sacrificing the accuracy needed for performance estimation.

To estimate end-to-end performance, we implement a fast queueing simulator driven by the RapidQ model. By combining HLS synthesis reports with designer annotations, the simulator programmatically converts the functional simulation trace into precise timing characteristics for each data element for any design configuration. This approach decouples functionality from timing, allowing a single functional trace to be reused across iterations to evaluate various combinations of module throughputs and buffer sizes. By eliminating the need for repeated functional simulation, RapidQ significantly reduces the turnaround time for design-space exploration.

Lastly, driven by the queueing performance simulation, we develop an automated tuning flow to identify resource-efficient configurations for input-dependent streaming pipelines that meet specific throughput targets for a given input workload. The exploration space encompasses both buffer sizes and parallelism/unroll parameters for the streaming modules. We formulate a resource cost function that normalizes heterogeneous FPGA resources into a unified percentage-area metric. This enables the tuning loop to efficiently tradeoff module resources against buffering capacity, minimizing the total resource footprint of the pipeline.

We evaluate our modeling framework and tuning flow using two input-

dependent streaming pipelines: (1) the 3D Rendering pipeline from the Rosetta benchmark suite [1] using both provided and synthetically generated traces, and (2) the Multi-String Matching Pipeline from the Pigasus Intrusion Prevention System [6], evaluated against real-world network security traces. Our results demonstrate that RapidQ accurately estimates the performance of the pipelines across diverse design configurations (within 3% error) while being 7x faster than state-of-the-art HLS simulators [2, 3] and over 100x compared to RTL simulation. Furthermore, using our tuning flow, we show that co-optimizing module throughputs and buffer sizes saves up to 42% in resource footprint compared to baselines that tune these parameters in isolation.

5.1 Background and Motivation

5.1.1 Input-Dependent Performance

Consider the computation in dense matrix multiplication. For a fixed matrix size, the datapath's operations and control flow remain entirely deterministic, regardless of the matrix values. A performance model for finding the optimal tiling configuration for a matrix multiplication accelerator can be expressed analytically in closed-form, simplifying design-space exploration.

In contrast, deriving performance and optimizing resource efficiency for input-dependent designs is non-trivial. Consider the 3D Rendering pipeline from the Rosetta benchmark suite [1], as shown in Figure 5.1. The pipeline processes a stream of triangles through input-dependent rasterization (R) and zculling (Z) modules. For typical triangle sizes, module R has lower throughput than Z, ensuring Z is not the bottleneck. However, a burst of small triangles can quickly overwhelm module Z, stalling the pipeline. While increasing buffer capacity can absorb short

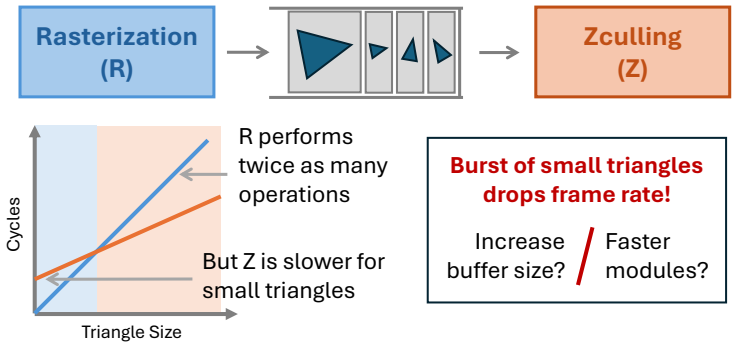
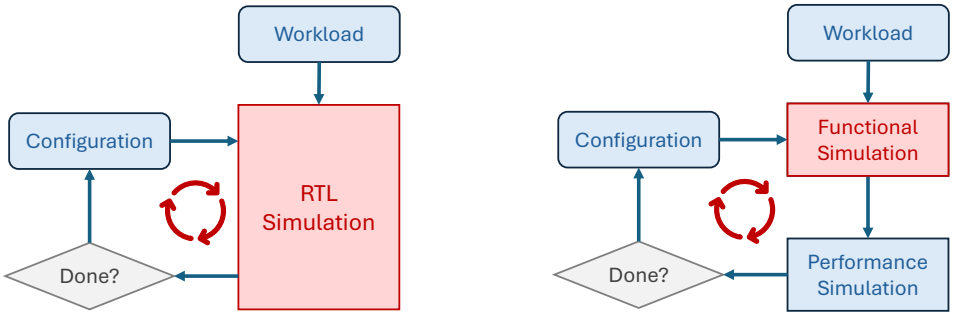


Figure 5.1: Performance of 3D Rendering from the Rosetta benchmark suite [1] varies with input triangle size. The tradeoff between module throughput and buffer size for resource efficiency is not trivial.

bursts, a longer burst might be better served by increasing module parallelism to meet the target frame rates. Hence, performance is dictated by many factors, such as the distribution of incoming triangle sizes, how the distribution changes, and the relative hardware costs of compute modules versus buffering. Composing these kernels in a streaming pipeline further compounds the complexity, creating a tightly coupled, multi-dimensional design space. *The time-varying nature of input-dependence and its coupling across pipeline stages can no longer be expressed in a simple closed-form analytical model, necessitating simulation.*

5.1.2 Related Work

Simulation is essential for estimating the performance of input-dependent applications for design-space exploration (DSE). For FPGA designs written natively in RTL, this necessitates RTL simulation. While RTL simulation provides cycle-accurate performance estimates, it is extremely slow due to the detailed bit- and cycle-level nature of the event-driven execution, hampering DSE as shown in Figure 5.2a. For large traces, synthesizing the design and executing it directly on



(a) DSE is bottlenecked by lengthy RTL Simulations in the exploration loop.

(b) Repeated functional simulations as part of the exploration loop remains the bottleneck for tools like FIFOAdvisor [61].

Figure 5.2: Prior work has shown significant speedups compared to RTL simulation-based approaches by decoupling performance simulation from functional evaluation of workloads for HLS implementations of input-dependent designs. However, repeated functional simulations make the approach unscalable to large real-world workloads.

the FPGA can sometimes be faster, provided that sufficient observability is maintained through hardware instrumentation [130]. To tackle the bottlenecks of RTL simulation, many prior works have developed HLS-based DSE tools targeting input-*independent* designs with static, model-based techniques [55–59]. However, very few frameworks address the unique challenges of performance estimation and tuning for input-dependent dataflows and streaming pipelines.

REFINE [60] is a runtime execution-based approach that uses hardware instrumentation to monitor stalls directly on the FPGA. Using partial reconfiguration-based incremental compilation, REFINE accelerates synthesis for iterative tuning while extracting accurate performance metrics directly from the FPGA. *However, even with faster compiles and a search space limited to module throughput parameters, the tuning process requires hours in the best-case scenario.* Our performance model captures input-dependent behavior without the need for full RTL deployment,

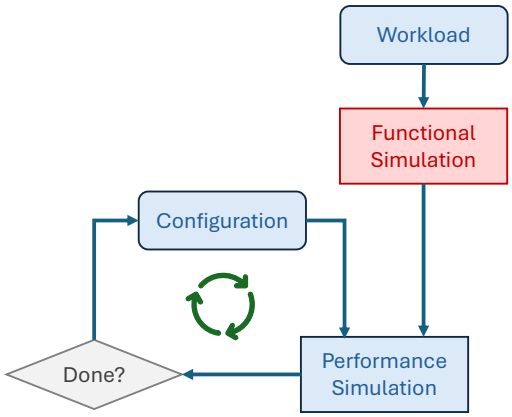


Figure 5.3: Leveraging the structure in input-dependent streaming pipelines represented in HLS, RapidQ aims to isolate functional characterization of the workload from the DSE loop and make performance simulation faster using a higher-level queueing-based abstraction.

significantly reducing iteration time. We directly compare our approach against REFINE using the input-dependent 3D Rendering benchmark from the Rosetta benchmark suite [1].

Our simulation and tuning flow shares similarities with the decoupled approach of LightningSim [2, 131], which separates trace generation from fast performance simulation. LightningSim is powerful, capable of automatically estimating the performance of existing HLS designs with near-cycle accuracy (close to 100%), significantly shortening the feedback loop for DSE in the design phase as shown in Figure 5.2b. FIFOAdvisor [61] uses LightningSim estimates to tune buffer sizes, but has limited simulation scalability. Real-world workloads [6] are often orders of magnitude longer than the small testbench inputs typically used.

In our work, *we assume a high-performance parameterized pipeline has already been designed and focus on tuning it across diverse input workloads rather than achieving cycle-accuracy.* Figure 5.3 illustrates the two primary design goals that

follow from this approach. First, RapidQ makes simplifying assumptions about the structure of the compute kernels in the pipeline, enabling the decoupling of functional characterization of the input workload from the configuration sampled in the DSE loop. Second, to accelerate performance simulation, our queueing abstraction offers a more scalable solution, trading off *minor accuracy losses for significant gains in simulation speed* by restricting modeling to packets at the stream boundaries.

Furthermore, unlike brief testbench traces, we observe that real-world workloads exhibit significant time-varying behavior, such as burstiness across modules in input-dependent streaming designs, as illustrated with the 3D Rendering example in Section 5.1.1. Under these dynamic conditions, tuning buffer sizes in isolation while modules remain under-provisioned can result in excessive memory usage and a resource-inefficient design; tuning only modules can be equally inefficient. Our approach takes a step further by co-tuning both module throughputs and buffer sizes in contrast to REFINE, which tunes only modules and FIFOAdvisor and other works [132, 133], that tune only buffers for performance or deadlock avoidance. While [134] advocates for similar co-tuning, their work focuses on preventing stalls in synchronous dataflow (SDF) models. They assume constant throughput through buffers and depend on static analysis to target data-independent workloads. We argue that co-tuning is essential to preventing resource inefficiency arising due to time-varying loads (or bursts) in *input-dependent* streaming pipelines, by exploring the tradeoff between throughput and buffer sizing.

5.2 RapidQ Performance Model

Inspired by queueing-theoretic analysis of computer systems [38, 39, 62], we develop the *RapidQ* framework to model input-dependent streaming pipelines on FPGAs.

In this section, we introduce the three primary components of the RapidQ performance model and detail how they address hardware-specific challenges: (1) the *Timing Model* captures the static HLS schedule independent of the workload; (2) the *Workload Model* converts the input data trace into an abstract load representation for each server, agnostic to throughput configurations, which together with the workload model provides per-kernel timing for each input packet; and (3) the *System Queueing Model* represents the dataflow graph structure of the pipeline, translating the per-kernel timing into end-to-end performance.

5.2.1 Server Timing Model

Calculating accurate timing through each server requires a thorough characterization of the kernel's underlying microarchitecture. Typically, the complex, input-dependent behavior of streaming kernels dictates that detailed RTL simulation or HLS scheduler analysis be performed to extract timing metrics for every new configuration. However, the design patterns introduced in Chapter 2 impose structure on these input-dependent streaming computations, significantly simplifying characterization.

Specifically, decoupling input-dependence from the core parallelizable compute, making the kernel amenable to statically scheduled HLS, yields a standardized implementation pattern as shown in Listing 5.1. The core inner loop is pipelined, iterating over the input stream to perform input-independent parallel processing on a per-flit basis. The parallelism and resource utilization of the kernel are in turn controlled by the UNROLL parameter of the innermost for loop processing elements within each flit. An optional outer loop handles packet-level operations, executing exactly once per packet regardless of the packet's total flit count. Under this struc-

```
while (true) {
  // Optional per-packet logic
  while (!eop) {
    // Can read from multiple streams
    in_flit_t in_flit = in.read();

    out_flit_t out_flit;

    #pragma unroll
    for (int i = 0; i < UNROLL; i++) {
      // Do some parallel processing
    }

    // Can write to multiple streams
    out.write(out_flit);
  }
}
```

Listing 5.1: Parameterized streaming HLS kernels decoupled for input-dependence follow a standard parallelizable design pattern enabling analytical modeling of per-kernel timing independent of the input workload.

ture, input-dependence manifests primarily as variation in packet length (flits per packet) across the workload.

Based on this template, the inter-packet delay can be modeled as:

$$\alpha + n\beta$$

where n is the number of flits in a packet, β is the inner loop's per-flit inter-iteration delay (also called Initiation Interval or II) and α is the fixed inter-packet overhead. This formulation directly mirrors classical communication cost models in distributed computing because both paradigms process packetized data through pipelined streaming resources. The fixed overhead α is analogous to the message startup latency, the flit count n corresponds to the payload size, and the per-flit delay β represents the reciprocal channel throughput (service time per flit).

Ideally, high-performance streaming kernels are designed to be fully pipelined

with no inter-packet overhead ($\alpha = 0,\ \beta = 1$), achieving maximum throughput. Under these ideal conditions, the cycle delay scales linearly with the number of flits per packet (n). In practice, however, memory dependencies can introduce latency overheads. For example, the zculling module from the 3D Rendering example has a Read-After-Write dependence on an internal memory buffer between consecutive triangles (packets). This loop-carried dependency prevents pipelining of the loop over packets, introducing a fixed cycle penalty (α) per triangle. While this overhead is amortized over large triangles with high flit counts, it becomes the dominant performance bottleneck for small triangles. Hence, explicitly incorporating α into the timing model is essential for accurate performance estimation across diverse workload traces.

5.2.2 Workload Model

The workload model characterizes n from the timing model to calculate the per-packet delay through kernels in the streaming pipeline. A key requirement for the workload model is parameter independence: the ability to translate the model into timing information for all module throughput combinations with a single model representation. This is straightforward for most modules where the number of flits per packet is governed directly by the UNROLL parameter and can be simply calculated by dividing the number of flits by the UNROLL factor. For example, the rasterization module shown in Figure 5.1 processes a triangle's bounding box a fixed number of pixels at a time. The model can dynamically calculate the cycle count per triangle for any given UNROLL factor without requiring re-simulation using the total bounding box size as the baseline number of flits.

Irregular Parallelism and Sparsity. Sparsity, however, prevents this translation

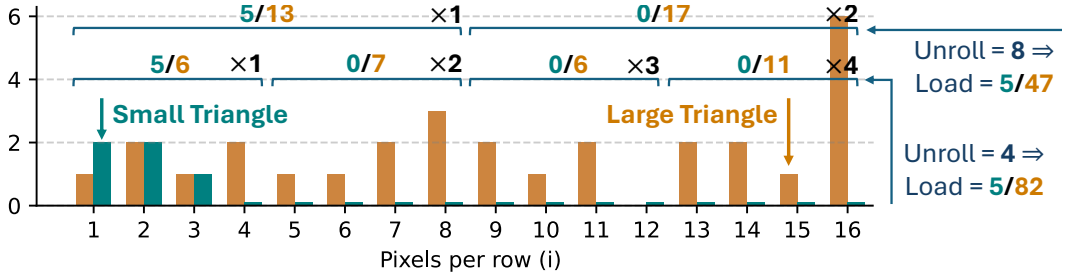


Figure 5.4: Sparsity histogram for 3D rendering. Large triangles benefit from higher unrolling which increases sparsity for smaller triangles.

from a single number into number of flits for all possible configurations. In the zculling module of the 3D Rendering pipeline, the number of pixels per row varies both within and across triangles. However, efficient parallel hardware requires that a fixed aligned width of pixels be accessed every cycle to allow banking. This leads to wasted computation for *invalid* padding pixels within an aligned input element, which we characterize as intra-element sparsity.

This sparsity is an intrinsic overhead of mapping irregular parallelism to regular spatial hardware as described in Chapter 2, and ignoring it leads to significant modeling inaccuracies. For example, processing 16 valid pixels might require multiple cycles even with an unroll factor of 16 if the pixels are fragmented with 50% sparsity, rendering any unrolling beyond 8 ineffective. We capture this sparsity explicitly using a histogram of the Number of Non-Zero elements (NNZ) along the parallelism dimension, tracked up to the maximum UNROLL factor in the search space (an example is shown in Figure 5.4). Using this histogram, the effective load for a particular unroll factor is calculated as:

$$Load = \sum_{i=0}^{Max\ Unroll} h[i] \left\lceil \frac{i}{Unroll} \right\rceil$$

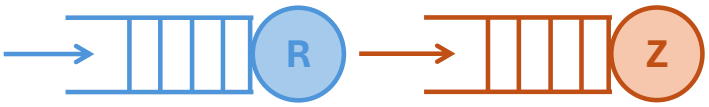


Figure 5.5: A queueing system with two servers and queues in a chain modeling the input-dependent system from Figure 5.1. Queues and servers form the central building blocks of queueing systems.

5.2.3 System Queueing Model

Lastly, the system queueing model captures the end-to-end performance of the streaming pipeline, modeling the queueing behavior arising from inter-packet and inter-kernel interactions. The queueing model is composed of two fundamental primitives: *servers*, which represent hardware processing kernels, and *queues*, which model the interconnecting buffers. Figure 5.5 illustrates this mapping applied to the example in Figure 5.1. This graph of queues and servers captures the complex interaction between various input-dependent dimensions while decomposing the design space into smaller subspaces around each server, making systematic exploration tractable. The queueing model is also highly flexible, allowing the building blocks to represent complex input-dependent design structures such as early-exit patterns and forks in the dataflow graph. Time-varying behaviors, especially workload burstiness, are implicitly modeled in the transient dynamics between queue occupancy and server processing rates, effectively estimating pipeline stalls and backpressure.

5.3 RapidQ Simulator

The queueing model serves as the performance abstraction for capturing the performance-resource tradeoffs of input-dependent pipelines. However, analytically deriving performance of complex queueing systems, especially with trace-defined

```
// Instantiate components
initiator   = new server_source("initiator", "trace.csv");
fifo_1      = new fifo("fifo_1");
raster      = new server("rasterization");
fifo_2      = new fifo("fifo_2");
zcull       = new server("zculling");
target      = new sink("target", "-1");

// Bind sockets to build system
initiator->socket.bind(fifo_1->target);
fifo_1->initiator.bind(raster->target);
raster->initiator.bind(fifo_2->target);
fifo_2->initiator.bind(zcull->target);
zcull->initiator.bind(target->socket);
```

Listing 5.2: Example two server system in the RapidQ Simulator which can model the example from Figure 5.1

distributions, is often mathematically intractable even with significant approximations [135]. We opt for a simpler discrete-event simulation approach driven by the queueing model.

We implemented our RapidQ simulator using SystemC's Transaction Level Modeling (TLM) framework [136, 137]. As a C++ extension, the TLM infrastructure facilitates the modular construction of a library of queueing components to express any input-dependent streaming pipeline based on its system queueing model. To accelerate execution, the RapidQ simulator operates on *simulation packets*: constructs that can aggregate the behavior of multiple data elements. By decoupling simulation packets from the packets in the hardware design, the simulator maintains flexibility in accurately capturing fine-grained input-dependent dynamics. Table 5.1 lists all the modules available in the simulator library. Listing 5.2 shows the queueing system from Figure 5.5 as written using the simulator library.

The simulator supports a trace-driven mode in which the source module can be customized per application to pull data from the workload model. The source

Table 5.1: Modules available in the RapidQ Simulator library. The first set of modules are sufficient for a linear pipeline.

Module	Function
server	server with one input and output socket; processes one packet at a time
fifo	queue; holds packets up to the specified capacity before backpressuring
source	generates packets using trace or distribution
sink	server without an output socket
fork	steers packets from one input to many outputs
merge	multiplexes packets from multiple inputs to a single output
replicate	copies packets from one input to many outputs
fusion_server	server with multiple inputs; produces one output packet after consuming one packet from each input

module is responsible for translating the contents of the trace into per-module timing for each packet based on the module's timing model. These parameters are stored in the packet object that each module probes to calculate its decision and delay variables using the parameters corresponding to itself. For example, for the system in Figure 5.1, the trace would consist of entries such as $(x, [y_0, y_1...])$, where x is the number of pixels in the bounding box for the rasterization module and y captures the sparsity histogram in the input to the zculling module. The trace-driven mode can be mixed with timing-dependent behavior, such as capacity-dependent overflow in forks, which cannot be captured in a functional simulation trace.

The simulator also features a rich set of extensions to collect metrics from its queue and server modules. The modules are equipped to record all events necessary to calculate relevant statistics, such as latency of packets and max occupancy through a queue.

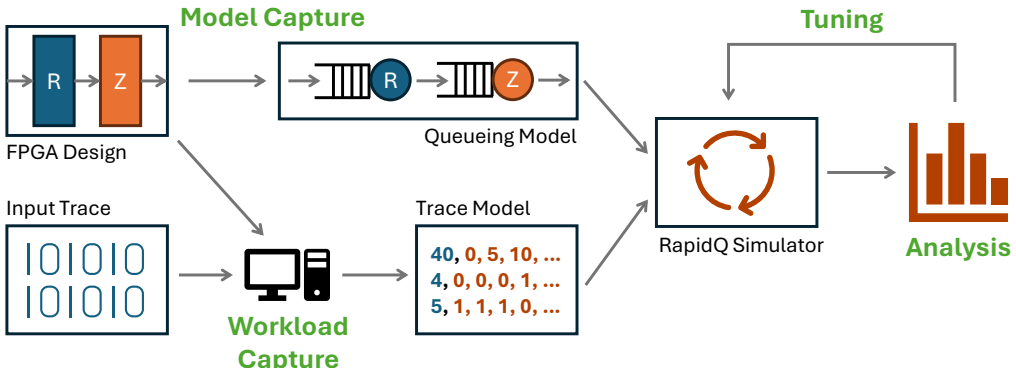


Figure 5.6: RapidQ Workflow

5.4 Model and Workload Capture

The first step of the RapidQ approach involves capturing the input-dependent streaming pipeline and workloads using the performance model described in Section 5.2. This modeling flow is a one-time effort per application design and is amortized over multiple tuning runs, enabling designers to build a reusable deployment-time tuning infrastructure for their high-performance input-dependent streaming FPGA pipeline. In this section, we describe the steps to model an application using RapidQ.

5.4.1 Model Capture

Our system queueing model has a one-to-one correspondence with how streaming pipelines are implemented in HLS, where streams or pipes representing FIFOs connect processing modules. This allows a direct translation of an HLS graph into our queueing model, but it can lead to the inclusion of input-independent modules, such as compaction used for sparsity (Section 2.2.3), which are superfluous to the performance model. We leverage the designer's insights into their design to prune unnecessary or redundant sections from the model and merge overly detailed hierarchies, thereby improving simulation speed. The resulting model is readily im-

plemented in the RapidQ simulator using the building blocks described in Section 5.3.

Model capture must also account for the timing model (α and β parameters from Section 5.2.1) for modules that might not be fully pipelined, for example, due to memory dependency issues. This scheduling information is available in HLS synthesis reports and may also vary between different unroll factors. During the implementation of the RapidQ simulation, the designer must explicitly encode this latency to correctly translate the load presented by the trace into precise timing delays. This characterization step incurs minimal overhead: HLS synthesis completes on the order of seconds to minutes, only needs to be run once per design, and can be parallelized across individual kernels.

5.4.2 Workload Capture

Since RapidQ operates at the boundaries of buffers and modules, instrumenting the design, especially using HLS, requires very minimal design modifications. As discussed in Section 5.2.2, the goal is to quantify the number of flits generated by each packet in all components of the model. This can be achieved by instrumenting buffers within the functional software simulation to extract relevant counts (e.g., pixel counts for rasterization). This instrumentation is not synthesized, making it transparent to the rest of the HLS and FPGA implementation flows.

To model sparsity accurately, we must extract a histogram of valid elements within a data element. To support the full design space of parameters, functional simulation must be parameterized to the maximum possible UNROLL factor for each module. This captures the full resolution of sparsity information, which can then be translated into the effective load for a specific UNROLL configuration in

the simulator.

5.5 Tuning for Resource Efficiency

The goal of the tuning flow is to enable users of input-dependent streaming pipelines to generate a resource-efficient configuration that meets the throughput performance target for their input workload. Unlike model capture, which is a one-time characterization per design, the tuning flow must be repeated for each unique deployment context, such as a new input workload, a different FPGA device or performance target. To this end, we developed a fully automated tuning flow built on the RapidQ performance model. The flow rapidly co-tunes buffer sizes and module throughputs, with resource minimization as the objective.

In the rest of this section, we first formally define an application configuration. We then describe the resource cost function used to evaluate candidate solutions and, lastly, the iteration strategy used to traverse the search space.

5.5.1 Design Configuration

As described in Section 2.4.2, modules in streaming dataflow pipelines typically perform simple parallelizable computations. Reflecting this structure, the RapidQ performance model exposes two primary configuration parameters per server-queue pair: a throughput configuration parameter (unroll factor) and a buffer depth parameter. We exclude parameters that alter the application's functionality. Such changes would invalidate the existing workload model, effectively constituting a new design that must repeat workload capture.

5.5.2 Resource Model

Modern FPGAs are composed of a heterogeneous mix of resources comprising logic modules (Look-up tables and flip-flops), On-Chip SRAM and DSP blocks [102]. Co-tuning buffers and modules requires a unified cost function that combines these resources into a single value.

For this work, we define the resource cost as the sum of normalized utilization fractions/percentages of each resource type. Normalization is performed against the maximum availability of each resource type on the specific target device, allowing the metric to adapt to the varying mixes of different FPGA families. The optimization objective is then to minimize the sum of these fractions, subject to the constraint that no individual resource type exceeds the device limits:

$$Cost = f_{logic} + f_{SRAM} + f_{DSP} < 3$$

$$f_{logic} < 1,\ f_{SRAM} < 1,\ f_{DSP} < 1$$

Streaming Module Resources

Although our tuning methodology is orthogonal to the specific resource model, in this dissertation, we adopt a first-order compositional approach [138] that aggregates individual module estimates, to avoid repeated full-system HLS runs. We use pre-characterized HLS resource estimates for individual modules across their throughput parameterization spaces. Since the primary objective of the tuning loop is to explore the tradeoff between buffering and module throughputs in input-dependent pipelines, estimation accuracy is secondary to capturing correct scaling trends. These estimates are collected offline and do not directly impact tuning time.

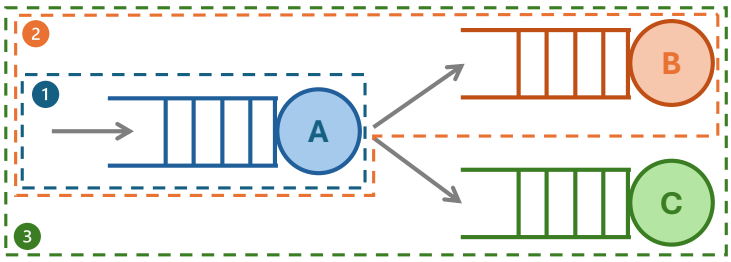


Figure 5.7: Sub-designs tuned when using greedy front-to-back tuning for a dataflow directed acyclic graph with branches

Calculating Buffer Resources

In the target Altera FPGA architecture that we use in this work, FIFOs are implemented using either logic resources for shallow depths ($\leq$32) or dedicated on-chip SRAMs for larger capacities. To avoid the complexity of heterogeneity with buffer resource modeling and the relative abundance of logic resources, we ignore the resource consumption of logic buffers, similar to FIFOAdvisor [61]. The M20K SRAM resource usage is calculated based on the supported aspect ratios (1024×20-bit or 512×40-bit):

$$\#SRAM = \min \begin{pmatrix} \lceil Width/40 \rceil \lceil Depth/512 \rceil, \\ \lceil Width/20 \rceil \lceil Depth/1024 \rceil \end{pmatrix}$$

5.5.3 Greedy Front-to-Back Tuning

Co-tuning buffers and module throughput exposes a broad design space. An unstructured exploration of this space for real-world workloads can be computationally intractable despite the enhanced simulation agility provided by RapidQ. We observe that, in dataflow directed acyclic graphs, the throughput capacity of a particular stage (server-queue pair) is independent of downstream performance. While downstream bottlenecks can create backpressure, they cannot be alleviated by changing the configuration of the upstream stages.

Hence, we structure our design-space exploration (DSE) to operate front-to-back, greedily tuning each stage to be resource minimal before proceeding to the next. All upstream stages are fixed in their tuned configurations when tuning a particular stage, ensuring that the simulation accurately captures the arrival process into the stage, as shown in Figure 5.7.

Per-Stage Tuning

With a more manageable design space for each stage, the search for the optimal resource configuration requires fewer iterations. We opt for a heuristic approach to optimization prioritizing simplicity over efficiency to showcase the effectiveness of our modeling approach and tuning goals (advanced optimization strategies are left to future work). Our flow for each stage follows these steps:

1. *Establish Worst-Case Throughput.* We first determine the upper bound for module throughput that satisfies the throughput target given a minimum-sized buffer. We increase the buffer size until we can meet the throughput for the widest module if minimum buffering fails.

2. *Iterate Module Throughput.* We reduce the module throughput to its next lower configuration. For each value, we test the feasibility with an unbounded buffer, then use binary search to identify the minimum buffer size needed to meet the target throughput.

3. *Cost Minimization.* We repeat Step 2, evaluating the resource cost for each valid configuration. The search stops when the configuration becomes infeasible or when the resource cost strictly increases.

Backpressure Propagation

An alternative simpler strategy is to conservatively over-provision each buffer-module pair to isolate backpressure locally at each stage (as seen in [139] for microservices). Our front-to-back tuning relaxes this strict isolation, allowing a stalled downstream stage to leverage *unoccupied upstream buffers*, effectively exploiting existing resources when upstream stages are not stressed by the same input conditions. If this cascading backpressure ultimately induces an upstream overflow, the tool increases the *local buffering of the responsible downstream stage*. This approach accurately provisions the pipeline to handle complex congestion propagation while avoiding the inefficiency of rigid over-provisioning.

5.6 Application Studies

In this work, we test our abstraction and flow on the 3D Rendering application from the Rosetta benchmark suite [1], and a Multi-String Matching pipeline from a real-world network security application, Pigasus [6]. These benchmarks were selected to present unique modeling challenges that rigorously stress our simulation and tuning flow. In this section, we characterize the application models, their input workloads, and the tuning objective.

5.6.1 3D Rendering Benchmark

The 3D Rendering application processes a stream of 3D triangles per frame and projects the geometry onto a 2D plane, subject to real-time frame rate constraints. The pipeline, previously introduced in Figure 5.1, consists mainly of two input-dependent modules: rasterization and zculling. Input dependence in this workload arises from two sources: sizes and shapes of triangles (which dictate the number of

pixels generated) and the number of triangles per frame.

While structurally simple (Figure 5.5), this benchmark evaluates our flow's ability to model the intricacies of memory-dependence-induced latency overhead in the zculling module and input sparsity (Section 5.2). The design space is constrained to a single unroll parameter that governs the parallelism of both modules. Tuning focuses primarily on the zculling buffer, since the rasterization buffer is negligible due to the data expansion nature of the stage (triangles to pixels). Furthermore, because the module logic is light on resources, the design is highly sensitive to buffer sizing, where excessive buffering can rapidly dominate the total area cost.

Traces. As the original benchmark suite ships with only a single trace, we generated a set of synthetic traces to evaluate robustness. These traces introduce bursts of small triangles of varying lengths into the original trace to emulate realistic high-stress rendering scenarios. All traces are normalized to process the same total number of pixels to ensure fair performance comparison.

5.6.2 Multi-String Matching

We evaluate the Multi-String Matching pipeline derived from the FPGA portion of Pigasus [6] previously described in Chapter 3, a representative real-world workload targeting a 100 Gbps line rate. The pipeline consists of three filtering stages: (1) Fast-Pattern Matcher (FPM), (2) Header Matcher (HM), and (3) Conjunct Pattern Matcher (CPM). The feasibility of this application on FPGAs strongly hinges on its input dependence which is ingrained in the pipeline, as shown in Figure 5.8 with each server-queue pair having its own unroll and buffer sizing parameter.

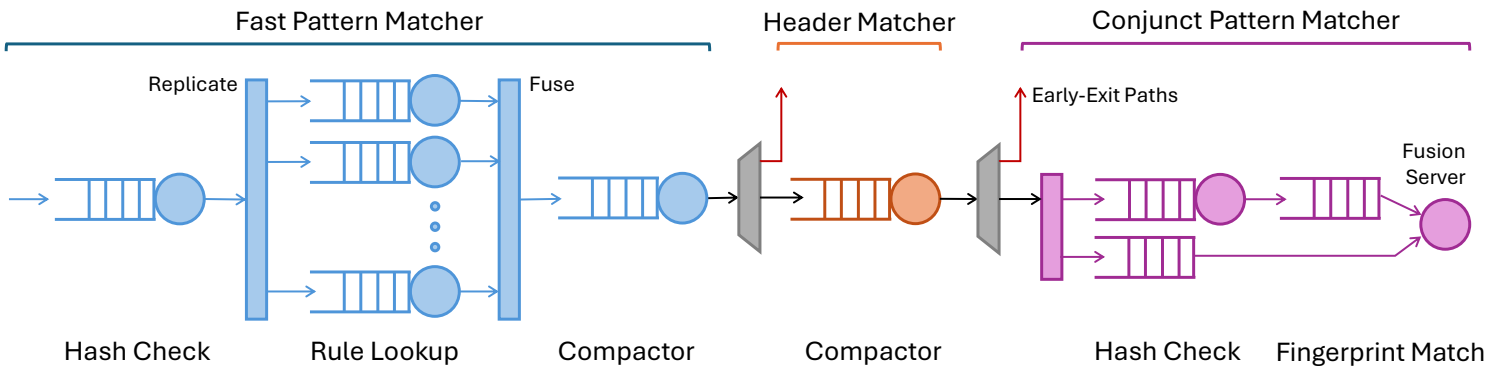


Figure 5.8: RapidQ Performance and Simulation model for the Input-Dependent Streaming Pipeline of Multi-String Matching

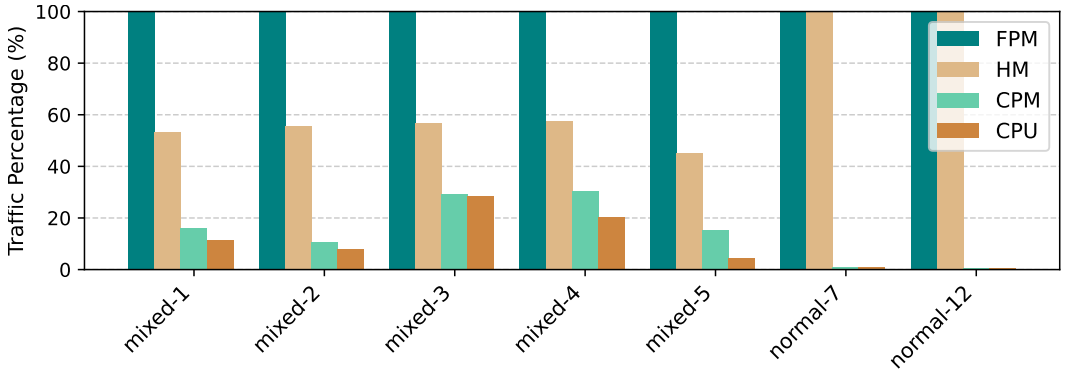


Figure 5.9: Percentage of input traffic entering each stage in the string matching pipeline for different traces.

Although the design is fully pipelined, it exhibits many input-dependent design patterns, such as early-exit and compaction stages which densify sparse elements in the streams. The dominant resource consumers are the two hash check stages (which utilize both logic and memory) and the rule lookup compactors inside the FPM (which are logic intensive). Strategically buffering bursts for these modules can significantly save on the pipeline's resource consumption, compared to over-provisioning the compute.

Traces and Rulesets. We use the publicly available Snort Registered Ruleset (snapshot-29141) [140] as in Pigasus [6]. We also use the same set of Stratosphere traces [141] to allow for direct comparisons: CTUMixed-Capture-1–5, CTU-Normal-7, and CTU-Normal-12. We refer to them as mixed-1–5, normal-7, and normal-12, respectively.

Observing Input Data-Dependence. Using the traces for multi-string matching, we can directly observe the input-dependent behavior that is a motivation for this work. The traces each have a different mix of malicious and benign flows, and this difference in the fraction of malicious flows directly affects packet traffic between

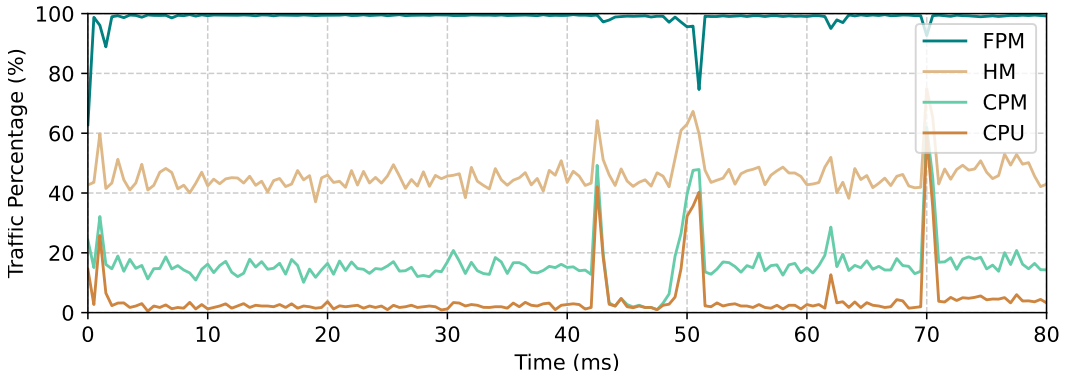


Figure 5.10: Percentage of input traffic entering different stages in the string matching pipeline for mixed-5 plotted against time

stages, as shown in Figure 5.9. Figure 5.10 plots the same traffic percentages, but over the duration of the mixed-5 trace. It shows that even though on average about 15% of the traffic entered the CPM stage, when seen over time, the rate of traffic is bursty. Tuning the modules for this trace with minimal buffers would lead to over-provisioning to the worst-case, and misconfigured module throughputs can require extremely large buffers that could exceed the FPGA SRAM capacity.

5.7 Evaluation

In this section, we characterize and evaluate the RapidQ simulation and tuning framework. We show that:

1. RapidQ's model-based simulation achieves high accuracy while providing a speedup of over **7x** compared to state-of-the-art cycle-accurate simulators.

2. By co-tuning module throughputs and buffer sizes, our flow generates resource-efficient designs for diverse input traces, saving up to **42%** in resources compared to tuning either parameter in isolation.

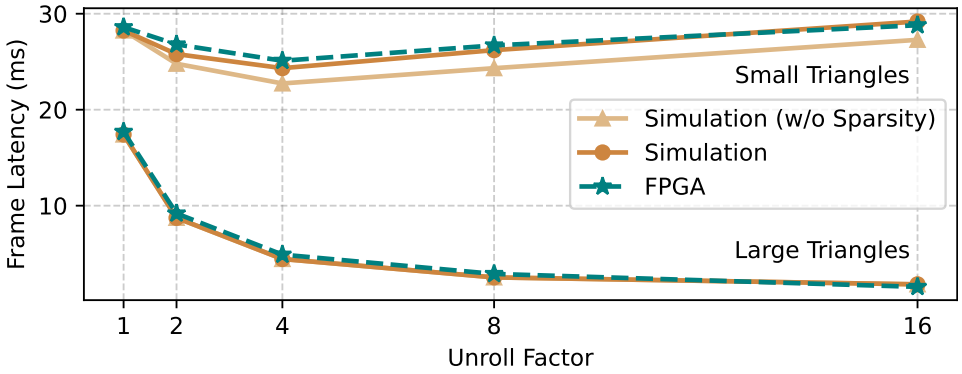


Figure 5.11: Simulation accuracy for the 3D Rendering pipeline for traces with different sized triangles when compared against FPGA execution with varying configuration parameters

5.7.1 Experiment Setup

We target the Terasic DE10-Agilex Accelerator Card [126], which hosts an F-Tile Agilex 7 (AGF014) Speed Grade-2 device. The Agilex 7 card has 487,000 ALMs, 139 Mb in on-chip SRAM and 4x8 GB off-chip DDR4 memory. Intel oneAPI Compiler version 2023.2, Quartus Pro 21.2 along with the OpenCL BSP provided by Terasic are used for HLS compilation, FPGA synthesis, and deployment, respectively. The simulation is run on a workstation equipped with an Intel i9-13900K processor and 128 GB DDR4-3600 memory.

5.7.2 Simulation Accuracy

While Multi-String Matching consists of fully pipelined modules, the 3D Rendering pipeline presents a more rigorous modeling challenge due to its complex HLS schedule. We validate our simulated performance using both the standard benchmark trace and a synthetic *small triangles* trace (Figure 5.11). Our simulation tracks FPGA execution very closely, with a worst-case underestimation of less than 3% for the small triangles trace. The effect of the memory-dependence latency penalty

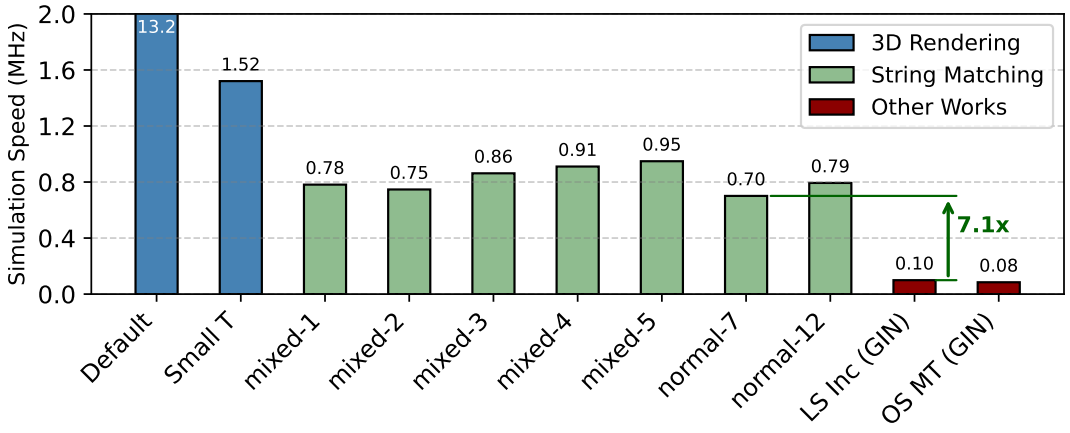


Figure 5.12: Simulation frequency across applications and traces compared to state-of-the-art HLS simulators (LS Inc: LightningSim [2] Increment Mode, OS MT: OmniSim [3] Multi-Threaded Execution)

can be seen for the small triangles, which worsens with the unroll factor due to deeper inner-loop pipelining. We also illustrate the effect of omitting sparsity modeling (Section 5.2.2), which leads to significant accuracy degradation and a severe underestimation of the runtime.

5.7.3 Simulation Speed

Figure 5.12 quantifies simulator performance (measured in simulated cycles per second) across applications and traces. Despite being single-threaded, RapidQ achieves simulation speeds within two orders of magnitude of native FPGA execution. 3D Rendering, being a small design, achieves higher simulation speeds, although the overhead of processing many small packets in the *small triangles* trace reduces efficiency. RapidQ maintains consistent simulation performance for Multi-String Matching, despite being a large design, across all traces regardless of input complexity.

We compare RapidQ against two state-of-the-art simulators: LightningSim

(Incremental) [2] and OmniSim (Multi-Threaded) [3] for their fastest (best-case) real-world application simulation (FlowGNN GIN [72]). To ensure a rigorous comparison, we isolate the execution time of the core simulation by excluding the one-time functional tracing cost common to all three methodologies. RapidQ's worst-case execution achieves over a 7x speedup compared to the best-case performance of these baselines, effectively exchanging a minimal loss in accuracy for significantly faster tuning iterations. This comparison confirms that RapidQ's performance gains are directly attributable to its lightweight queueing abstraction, which substantially lowers the computational burden of simulation.

5.7.4 Tuning

We apply our tuning flow to identify the resource-optimal configuration that meets the throughput targets for each trace. All results are verified by synthesizing and executing the configurations on the FPGA. We benchmark our co-tuning approach against two baseline strategies found in prior work:

- *Fixed Module* tunes only buffer sizes while keeping module throughputs constant, similar to FIFOAdvisor [61].
- *Minimum Buffer* tunes only module throughputs while setting buffers to their minimum, similar to REFINE [60].

3D Rendering

Figure 5.13 shows the resource consumption for the tuned configurations of the 3D Rendering pipeline. For traces with long bursts, the *Fixed Module* approach either violates the throughput target or results in excessive, resource-inefficient buffering. The *Minimum Buffer* approach is overly conservative even for short bursts, selecting

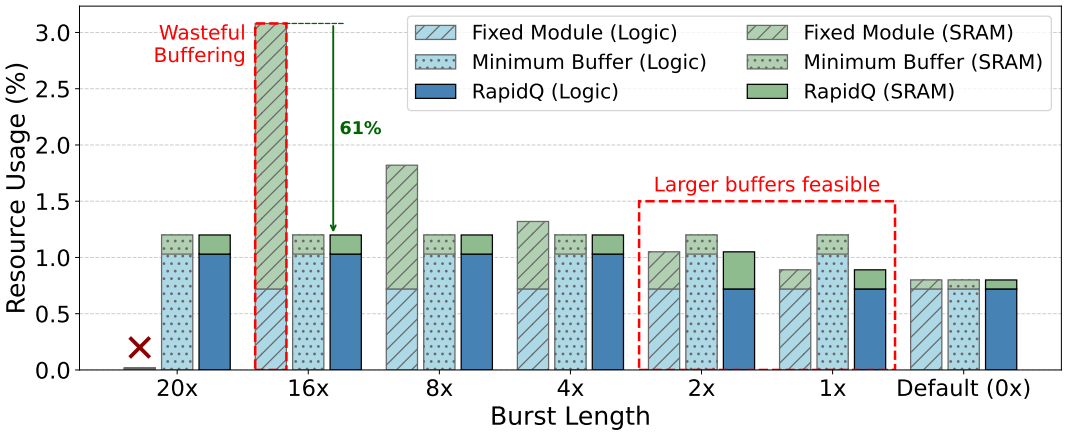


Figure 5.13: Tuning results for 3D Rendering across varying burst lengths of small triangles comparing RapidQ with baseline approaches

high-throughput module configurations rather than utilizing cheaper buffer capacity. RapidQ achieves optimal balance, scaling both module throughputs and buffer sizes to find the true resource-optimal configuration for every trace. Moreover, RapidQ converges within seconds, in contrast to the hours required by runtime execution approaches like REFINE [60].

Multi-String Matching

We see similar trends for the string matching pipeline in Figure 5.14. For the *Fixed Module* baseline, we utilized module throughputs derived from trace mixed-5. Traces mixed-1, 3 and 4 direct a larger fraction of traffic into the CPM stage of the pipeline, requiring more buffering than SRAM available on the FPGA, rendering the design invalid when using the *Fixed Module* approach. For lighter traces (normal-7 and 12), this design is severely over-provisioned.

The *Minimum Buffer* approach defaults to provisioning modules for the worst case in all the traces since each of them has at least a short burst of malicious traffic, resulting in consistently high resource usage. Our RapidQ approach, by co-tuning

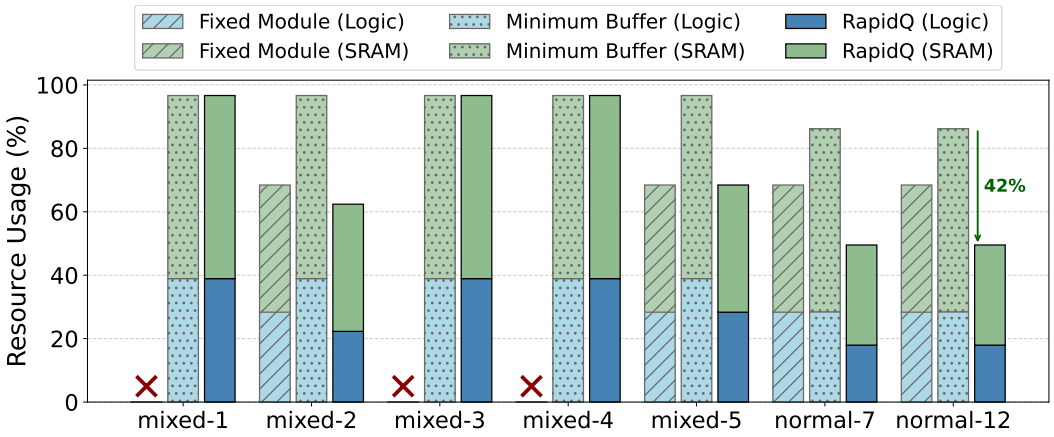


Figure 5.14: Tuning results for the Multi-String Matching pipeline across the Stratosphere traces comparing RapidQ with baseline approaches

module throughputs and buffer sizes, yields up to 42% resource savings. For normal-7 and 12, a small increase in buffering absorbs their shorter bursts of malicious traffic, allowing significantly reduced module throughputs and substantial area savings.

5.8 Discussion

5.8.1 Factoring Model Development Time

RapidQ incurs a one-time model creation overhead of 2 to 3 hours, which is amortized over the design's lifetime, covering subsequent updates that do not necessitate full remodeling. The consequence of this investment is that RapidQ achieves a significant reduction in per-trace optimization runtime to *under 45 minutes, compared to over 5 hours for performance models like LightningSim, or multiple days for RTL simulation.* In this section, we present a breakdown of these time requirements.

Tuning Time

The time required for each automated tuning run is calculated as:

$$N \times \left(L \times \frac{f_{FPGA}}{f_{sim}} \right)$$

where N is the number of tuning iterations, L is the trace length (in FPGA runtime) and f_{FPGA} and f_{sim} are the FPGA and simulation frequencies, respectively. On average, tuning runs require ∼50 simulations (8 iterations × 6 stages). Assuming $L = 100$ ms (derived from `normal-7`) and $f_{FPGA} = 400$ MHz, we can calculate the runtime using f_{sim} values from Figure 5.12. LightningSim requires 5.7 hours, whereas RapidQ completes the full pipeline simulation in under 45 minutes. Notably, applying our front-to-back strategy reduces RapidQ's time by nearly an additional 50%.

Model Development Time

The initial model development is split into three steps:

1. *Streaming Graph Capture:* Rewriting the streaming module graph using RapidQ's SystemC simulator primitives (Figure 5.2). For even an application as large as string matching, this takes only minutes.

2. *Timing Capture:* Extracting initiation intervals from each module's HLS loop analysis report, requiring roughly 10 to 20 minutes.

3. *Trace Capture:* Instrumenting HLS streams to log per-packet data at design boundaries and translating the output for the simulator, taking approximately 2 hours.

Altogether, the total model development time takes well under 3 hours, showing that *RapidQ is overall faster than LightningSim even when tuning for a single trace, with the speedup compounding as the model development time is amortized across multiple traces.*

5.8.2 Expert Designer Fallacy

It might be tempting to conclude that the expertise required to construct the Server Timing Model (Section 5.2.1) renders RapidQ's analysis and tuning capabilities redundant. While developing high-quality, parameterized modules and composing them into a streaming pipeline certainly demands a deep understanding of the computation, RapidQ is not intended to assist with this initial development or debugging phase. Instead, its value becomes apparent when the challenge shifts significantly at deployment. Even an expert designer cannot statically determine the optimal parameter configuration without characterizing the eventual deployment conditions. The complex, input-dependent interactions within a streaming graph under various input workloads are not easy to reason about unaided, despite a thorough understanding of each individual module. *RapidQ addresses this gap by allowing designers to construct a performance model once and ship the tuning framework alongside their parameterized design, thereby enabling rapid, case-by-case tuning for any current or future workload.*

5.8.3 Tuning Strategies

RapidQ's performance abstraction is entirely decoupled from the specific optimization technique applied during tuning. In this work, we presented a front-to-back tuning strategy as a concrete demonstration of how RapidQ achieves resource savings

for cycle-free pipelines. However, RapidQ's queueing-based approach is sufficiently general to model more complex graph structures (like cycles) and design constraints while preserving its high-level abstraction. When paired with our rapid simulation framework, RapidQ opens up avenues for future work to develop more comprehensive tuning strategies across a broader class of streaming pipelines, unencumbered by lengthy simulation turnaround times.

5.8.4 Modeling Automation

Although we perform modeling manually in this work, integrating the steps as an add-on to an existing HLS toolflow is limited to engineering effort. Since RapidQ mimics the streaming HLS model of pipeline construction, task graphs can be parsed to generate simulation topology, and stream wrappers can automate workload extraction. The key challenge lies in identifying the appropriate HLS scheduling reports to guide timing characterization. Using compiler IR [2] can add unnecessary detail to our lightweight model. A practical alternative could be to measure module latencies via a targeted RTL simulation on small sample inputs.

5.9 Summary

We introduced RapidQ, a comprehensive framework for performance modeling, simulation, and rapid tuning of input-dependent streaming pipelines on FPGAs. By modeling streaming dataflow computations as queueing networks, RapidQ enables FPGA accelerator designers to generate a lightweight representation that accelerates performance simulation by over 7x compared to state-of-the-art cycle-accurate HLS simulators and over 100x compared to RTL simulation. Our methodology decouples workload capture from timing analysis, enabling a single functional simulation trace

to drive the evaluation of a vast design space without re-execution. Empowered by this agility, our holistic automated tuning flow co-optimizes module throughputs and buffer sizes, achieving resource savings of up to 42%.

Chapter 6

Conclusion

The primary objective of this dissertation was to broaden the scope of hardware acceleration to encompass input-dependent streaming applications utilizing Field-Programmable Gate Arrays (FPGAs). In the current era of architectural specialization, custom hardware is essential for achieving cost-effective and scalable performance. However, traditional hardware platforms, such as Domain-Specific Accelerators (DSAs), lack the flexibility necessary to support the dynamic performance demands of common-case optimized applications. While FPGAs offer a superior tradeoff between programmability and hardware-level efficiency, their adoption has been constrained by rigid toolchains oriented toward the static design flows typical of DSAs. To overcome these limitations, this dissertation proposed a systematic methodology for representing input-dependent streaming applications as configurable High-Level Synthesis (HLS) design templates, coupled with a novel performance abstraction that facilitates rapid design-space exploration.

The first part of this dissertation enables the development of configurable design templates in HLS for representing input-dependent streaming applications.

HLS language features such as preprocessor directives and automatic hardware generation provide the parameterizability and designer productivity this approach requires, while a systematic, stream-based decomposition of input-dependence preserves performance despite the limitations of static scheduling. Chapter 2 presents key input-dependent design patterns that arise from common-case optimizations, along with strategies for using the stream processing paradigm to generate efficient hardware for these dataflow patterns. These design patterns allow HLS implementations of streaming pipelines to decouple core parallelizable computations from input-dependence, yielding kernels that are amenable to spatial hardware acceleration and that achieve high performance and resource efficiency without sacrificing configurability.

To demonstrate the practical effectiveness of this stream processing-based structuring for input-dependent applications, Chapter 3 presented a case study redesigning an FPGA-accelerated string matching pipeline, originally implemented in RTL for a network intrusion prevention system. This effort developed RapidScan, a high-performance, HLS-based string matching library that exposes extensive functional and performance parameterization. Using stream-based decomposition, this chapter demonstrated that HLS can effectively capture the varied input-dependent dataflow patterns in string matching, achieving RTL-comparable resource efficiency alongside superior configurability and seamless integration with surrounding kernels. Compared to the Pigasus baseline, a RapidScan-based Intrusion-Prevention System scales to higher network line rates with less than a 10% increase in logic utilization. Furthermore, RapidScan enables RapidDetect, a 200 Gbps FPGA-accelerated log monitoring solution that is 4x more cost-effective than existing software approaches, highlighting the broader viability and value of hardware accelera-

tion for dynamic, input-dependent workloads.

To further understand the limits of HLS in representing complex input-dependent dataflow, Chapter 4 explored the shuffle design pattern. Unlike other patterns, the configurability demands of the shuffle operation are standardized to well-established NoC topologies and router parameters. Lacking the need for ad-hoc flexibility, this pattern is uniquely suited for traditional RTL implementations that prioritize raw performance and hardware efficiency. To establish a state-of-the-art RTL baseline for this communication pattern, this research developed ReConnect, an RTL-native soft NoC generator. By harnessing modern FPGA toolchain compiler optimizations, ReConnect overcomes the configurability limitations of standard Hardware Description Languages (HDLs). This approach imparts extensive parameterizability while delivering over a 9x improvement in iso-resource throughput compared to existing soft NoC generators. When evaluated against an HLS-based butterfly topology serving the communication demands of a common-case optimized database aggregation accelerator, ReConnect consumes 35% fewer resources and exposes a broader, superior design space along the pareto front. With its high performance, modular generation capabilities, and plug-and-play streaming interfaces, ReConnect is well-positioned to facilitate the next generation of research in FPGA acceleration and virtualization that utilize such communication patterns.

Lastly, Chapter 5 addressed the lack of fast performance simulation tools for the design-space exploration (DSE) of FPGA-based, input-dependent streaming pipelines. Leveraging the structure imparted by the previously discussed design patterns and the stream-based decoupling of compute kernels from input-dependence, this chapter introduced RapidQ, a lightweight, queueing-based performance abstraction. RapidQ is predicated on the insight that the core compute kernels within these

pipelines are typically simple, highly parallelizable implementations well-suited for hardware acceleration. Consequently, the framework distills the detailed functionality of each kernel into its corresponding unroll factor, while representing the interconnecting streams as queues to accurately model input-dependent behavior across workload packets. By utilizing functional simulation to capture the workload behavior of these HLS implementations, RapidQ drives a fast queueing simulator that operates 7x faster than state-of-the-art approaches. Empowered by this rapid simulation capability, the chapter also detailed an automated DSE flow that co-tunes buffer sizes and module throughputs, achieving over 42% savings in resource utilization.

By coupling a systematic approach to developing configurable design templates with rapid performance estimation, this dissertation bridges the gap between the high performance of hardware accelerators and the agility required by input-dependent workloads. Rather than offering isolated optimizations, the integration of stream-based HLS design patterns, robust RTL infrastructure, and agile design-space exploration provides a comprehensive ecosystem for hardware developers. By lowering the barrier to deploying common-case optimized streaming applications on FPGAs, this dissertation will democratize access to hardware acceleration, paving the way for more scalable and cost-effective solutions for critical, high-throughput domains such as network security and database analytics.

6.1 Future Directions

6.1.1 Design Patterns + Dynamically Scheduled HLS

The design patterns described in Chapter 2 are essential in overcoming the limitations of static scheduling in commercial HLS toolflows. While this approach

effectively targets applications where input-dependence is limited to common-case optimizations, such as string matching and database aggregation, other applications such as sparse graph processing exhibit fine-grained input-dependence. For these applications, decomposing the application into our streaming design patterns would incur significant infrastructure overhead due to the generation of a large number of kernels with relatively small compute cores.

Prior research has explored alternative methods for expressing fine-grained dynamic dataflows. Dynamically scheduled HLS frameworks, such as Dynamatic [142], have shown potential for higher performance over static scheduling for these specific kernels. However, they widen the semantic gap between high-level C/C++ software and the resulting hardware, obscuring the cause-and-effect relationship between programmer interventions and hardware efficiency. This can lead to a loss in productivity gains and parameterization capabilities that are central to utilizing HLS for configurable design templates. Other approaches utilize dataflow primitives to offer domain-specific solutions, conceptually similar to our design patterns, for areas like sparse data processing [143], but lack the generality required to support arbitrary input-dependent applications.

Because statically scheduled hardware is inherently well-suited for spatial architectures, achieving better resource efficiency by minimizing control overhead [142], mapping input-dependent applications to statically parallelizable compute structures remains a preferable baseline in many cases. Future research could explore hybridizing these methodologies: exploring how the systematic, coarse-grained stream processing approach presented in this dissertation might be combined with dynamically scheduled HLS tools. Selectively applying dynamic scheduling to specific input-dependent infrastructure kernels (such as compactors or shuffle NoCs)

or to computations exhibiting fine-grained dynamism could expand the capacity of HLS to represent input-dependent applications as configurable design templates. Automatic partitioning techniques can further ease designer productivity, intelligently isolating input-dependent regions, and choosing the most appropriate HLS backend to produce efficient hardware for the complete application.

6.1.2 Accelerating Functional Simulation

As illustrated in Figure 5.3, a primary objective of RapidQ (Chapter 5) was to decouple functional simulation from the design-space exploration (DSE) loop to avoid the prohibitive time costs associated with large-scale, real-world workloads. Nevertheless, because functional simulation must still be performed once per workload, optimizing its runtime remains essential when evaluating multiple workloads.

In the context of monolithic kernels, functional simulation is essentially a software execution of the HLS code, which generally achieves software-level performance. To execute a streaming pipeline, however, modern HLS tools [45, 50] rely on concurrent execution of the kernel graph, where each process executes the computation of a single kernel. Data is transferred between these stream-connected kernels via inter-process communication (IPC) to resolve the end-to-end pipeline. While this method performs adequately for large, compute-bound kernels, it scales poorly: as the kernel count grows, the IPC overhead and the massive number of concurrent processes severely bottleneck the simulation runtime. This issue is further exacerbated by stream-based decomposition, a necessary technique for modeling input-dependent streaming pipelines (Chapter 2), which inherently generates numerous lightweight kernels and exacerbates the communication overhead.

Given that functional simulation is not intended to yield accurate hardware

performance metrics, an alternative workaround is for designers to implement simplified, monolithic kernels exclusively for simulation purposes. While this effectively reduces the kernel count and mitigates the communication bottleneck, it introduces substantial overhead for the programmer. Furthermore, maintaining two distinct versions of the codebase detracts from the fidelity of functional verification, as the simulated design diverges from the actual streaming pipeline intended for synthesis.

Future research could explore new functional simulation infrastructures targeting streaming pipelines that consist of many small kernels, where communication overheads dominate execution time. Transitioning to a single-threaded, event-driven simulation model could significantly accelerate execution by mitigating IPC and context-switching penalties. However, to maintain high performance, this would necessitate strategic partitioning so that compute-intensive kernels continue to execute in dedicated processes. Beyond functional simulation, this event-driven architecture could optimize RapidQ's SystemC-based simulator, which faces similar communication bottlenecks due to its lightweight queueing server models. A unified infrastructure handling both functional and performance simulation for RapidQ could enable tighter automation with HLS toolchains, significantly improving accessibility for designers.

6.1.3 Runtime Modeling and Tuning

While this dissertation primarily focused on compile-time tuning and design-space exploration for representative workloads, real-world applications frequently encounter dynamic conditions that a single, statically tuned implementation cannot efficiently accommodate. Diurnal variations in datacenter traffic [144], for instance, can radically alter performance requirements, causing static designs to become

unnecessarily over-provisioned during low-load periods and bottlenecked during peak utilization.

To address this, future work could explore runtime tuning mechanisms. At a minimum, such systems would need to dynamically identify the active workload to select from a library of pre-compiled application variants. Generalizing this approach to any unseen workload would necessitate the development of lightweight, real-time profiling tools for FPGAs that can capture input-dependent behavior without adding significant resource overhead or degrading the operating frequency of the design. Moreover, dynamic design-space exploration would require a surrogate performance model, since runtime testing directly on FPGAs is severely limited by the impact of long reconfiguration times on application execution.

RapidQ could be extended from its current trace-driven approach to distribution-based modeling and simulation, which aligns naturally with its underlying queueing framework. By leveraging RapidQ's existing queueing analysis, designers could strategically guide FPGA instrumentation to capture essential distribution metrics at stream boundaries, ensuring accurate performance profiling with minimal resource overhead. These statistical distributions could subsequently drive the RapidQ simulator for runtime design-space exploration, facilitating seamless, one-shot updates to the active FPGA configuration. Furthermore, transitioning to a distribution-based model would allow RapidQ to provide solutions with probabilistic performance guarantees, thereby preventing the hardware from overfitting to any single workload trace.

To achieve kernel-level tuning without the prohibitive overhead of reconfiguring the entire FPGA, further research into FPGA virtualization [145–148] would be critical. By partitioning FPGAs into smaller virtual FPGA tiles (vFPGAs)

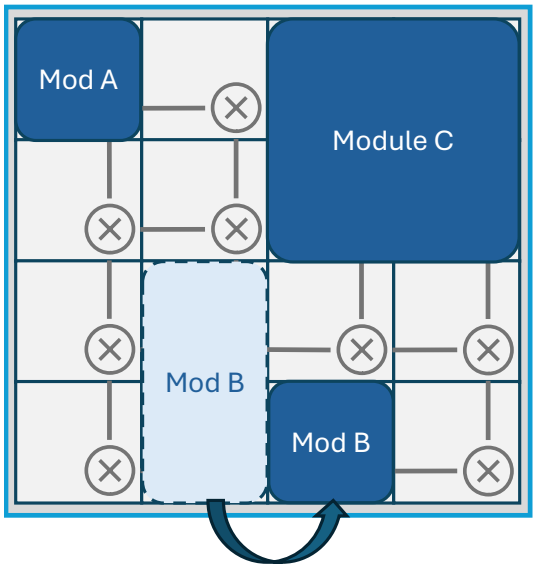


Figure 6.1: A virtualized FPGA fabric can enable software-like runtime tuning.

connected via NoC-style, latency-insensitive interfaces, systems could dynamically adapt specific kernels within streaming pipelines to better suit workload demands, as shown in Figure 6.1. Furthermore, integrating runtime instrumentation with runtime resizing of vFPGA tiles could bring RapidQ's tuning capabilities much closer to the flexibility of software auto-tuning.

6.1.4 Heterogeneous System-Level Modeling

Over the past decade, the architectures of reconfigurable, general-purpose, and custom-compute accelerators have steadily converged. As illustrated in Figure 6.2, processors are increasingly incorporating specialized hardened units, while communication-focused ASICs (such as Intel IPUs and Nvidia DPUs) are integrating general-purpose compute to broaden their applicability and future-proof the SoC. Currently, heterogeneous systems tend to statically map specific hard-

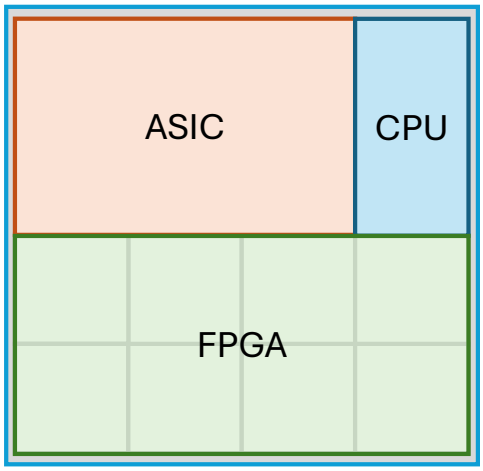


Figure 6.2: Modern accelerators are composed of heterogeneous compute resources at different points in the programmability vs efficiency tradeoff spectrum.

ware components to distinct applications or discrete functional phases. However, the tradeoff spectrum between highly efficient hardened accelerators and flexible software-programmable compute remains largely unexplored for serving the same functionality under varying workload demands.

RapidQ's queueing-based performance model has primarily focused on streaming pipelines on the FPGA. However, systems like Pigasus and RapidDetect already rely on software to perform rare-case processing. This software fallback could be further leveraged to handle dynamic bursts in the workload that depart from the common case but are shorter than the reconfiguration timescale of FPGAs. While CPUs lack the raw efficiency of hardware accelerators, their rapid program-switching and multi-tenancy capabilities could reduce overall costs when absorbing these high-frequency fluctuations, thereby preventing the need to over-provision the FPGA. At the other end of the spectrum, ASICs might be a better choice to serve the baseline load threshold that the workload never drops below.

RapidQ's abstraction could be extended to model this interaction between application workload demands and the programmability and processing power of heterogeneous components, potentially achieving a system more efficient than one relying on a single kind of compute resource. While queueing abstractions are inherently well-suited for performance analysis of computer systems [62], advancing this design-space exploration would require hybrid cost models that merge the static resource provisioning costs of hardware with the time-based utilization metrics of programmable software platforms. Developing robust workflows to dynamically map application components across this spectrum could inform the design of next-generation heterogeneous devices, driving efficiency gains in this era of specialized hardware accelerators.

Bibliography

[1] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, (New York, NY, USA), p. 269–278, Association for Computing Machinery, Feb. 2018. (document), 5, 5.1.1, 5.1, 5.1.2, 5.6

[2] R. Sarkar and C. Hao, "Lightningsim: Fast and accurate trace-based simulation for high-level synthesis," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 1–11, May 2023. (document), 1.3.2, 5, 5.1.2, 5.12, 5.7.3, 5.8.4

[3] R. Sarkar and C. Hao, "Omnisim: Simulating hardware with c speed and rtl accuracy for high-level synthesis designs," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, (New York, NY, USA), p. 1334–1346, Association for Computing Machinery, Oct. 2025. (document), 1.3.2, 5, 5.12, 5.7.3

[4] "Snort." https://www.snort.org. 1.1, 3.2.1

[5] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern {CPUs}," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 631–648, 2019. 1.2.1

[6] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1083–1100, USENIX Association, Nov. 2020. 1.3.2, 3.1, 3.2.1, 3.3, 5, 5.1.2, 5.6, 5.6.2, 5.6.2

[7] J. Chen, X. Zhang, T. Wang, Y. Zhang, T. Chen, J. Chen, M. Xie, and Q. Liu, "Fidas: fortifying the cloud via comprehensive fpga-based offloading for intrusion detection: industrial product," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 1029–1041, 2022. 1.1, 1.2.1

[8] "Elasticsearch." https://www.elastic.co. 1.1, 1.2.1, 3.1, 3.2.2, 3.4.1, 3.4.4

[9] D. Xue and R. Marcus, "Global hash tables strike back! an analysis of parallel group by aggregation," *Proceedings of the VLDB Endowment*, vol. 19, p. 523–535, Nov. 2025. 1.1, 4.7.1

[10] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker, "Quantifying tpc-h choke points and their optimizations," *Proc. VLDB Endow.*, vol. 13, p. 1206–1220, Apr. 2020. 1.1, 4.7.1

[11] B. Biggs, C.-S. Bouganis, and G. Constantinides, "Atheena: A toolflow for hardware early-exit network automation," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 121–132, 2023. 1.1, 2.2.1, 5

[12] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, p. 2464–2469, Dec. 2016. 1.1, 2.2.1, 5

[13] I. L. Markov, "Limits on fundamental limits to computation," vol. 512, no. 7513, pp. 147–154. 1.2

[14] J. Shalf, "The future of computing beyond moore's law," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190061, 2020. 1.2

[15] "Our eighth generation TPUs: Two chips for the agentic era." 1.2

[16] "AWS Trainium." 1.2

[17] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. Smullen IV, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. S. Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D.-k. He, C. R. Ho, R. W. Huffman Jr., E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor, N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V.-M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. K. Wu, "Warehouse-scale video acceleration: Co-design and deployment in the wild," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 600–615, Association for Computing Machinery. 1.2

[18] M. van der Hagen and B. Lucia, "Client-optimized algorithms and acceleration for encrypted compute offloading," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pp. 683–696, Association for Computing Machinery. 1.2

[19] A. Golder, R. Kumar, S. Taneja, K. Race, P. Aseron, J. Greensky, W. Wang, H. Gong, L. Kethareswaran, V. Suresh, A. Vartak, A. Challagundla, J. Casas, P. Lalwaney, D. Kim, C. N. Gutierrez, E. Z. Ramos, W. Cho, J. M. R. Chaves, M. Steiner, D. Lake, N. Yennampelli, K. Nivarthi, K. Bijinapally, B. P. Talamala, S. Valluri, V. Srirambhatla, C. Wilkerson, R. Cammarota, and S. Mathew, "HERACLES: 8192-Way SIMD Programmable Scalable Fully-Homomorphic Encryption SoC for Privacy-Preserving Cloud Computing in Intel 3 CMOS," in *2026 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 69, pp. 424–426. 1.2

[20] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 37–47, ACM. 1.2

[21] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens, "Programmable stream processors," vol. 36, no. 8, pp. 54–62.

[22] K. Mätas, K. Manev, J. Powell, and D. Koch, "Automated generation and orchestration of stream processing pipelines on fpgas," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, p. 1–10, Dec. 2022. 2.1.2, 5

[23] 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)*, pp. 1–13.

[24] I. Burstein, "Nvidia Data Center Processing Unit (DPU) Architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–20.

[25] "What Is a DPU?." 1.2

[26] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," p. 1083–1100, 2020. 1.2.1, 2.2.1, 4.1

[27] "Runreveal's sigmalite." `https://github.com/runreveal/sigmalite`. [Accessed 13-06-2025]. 1.2.1, 3.4.1

[28] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 138–147. 1.2.1

[29] D. Parravicini, D. Conficconi, E. D. Sozzo, C. Pilato, and M. D. Santambrogio, "CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching," vol. 20, pp. 51:1–51:24. 1.2.1

[30] A. Boutros and V. Betz, "Fpga architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, p. 4–29, 2021. 1.2.2, 4.2.2

[31] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, (New York, NY, USA), p. 21–30, Association for Computing Machinery, Feb. 2006. 1.2.2, 5

[32] A. Boutros, S. Yazdanshenas, and V. Betz, "You Cannot Improve What You Do not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference," vol. 11, no. 3, pp. 20:1–20:23. 1.2.2

[33] M. Lavasani, *Generating irregular data-stream accelerators: methodology and applications*. PhD thesis, 2015. 1.2.2

[34] M. Hall and V. 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, (New York, NY, USA), p. 320, Association for Computing Machinery, Feb. 2020.

[35] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, pp. 1–8, Association for Computing Machinery. 1.2.2

[36] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, Dec. 2022. 1.3

[37] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Confer-*

ence on Computer Systems, EuroSys '20, (New York, NY, USA), Association for Computing Machinery, 2020. 1.3.2

[38] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018. 5.2

[39] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, Nov. 2012. 1.3, 1.3.2, 5.2

[40] E. D. Sozzo, D. Conficconi, A. Zeni, M. Salaris, D. Sciuto, and M. D. Santambrogio, "Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs," vol. 55, no. 5, pp. 106:1–106:48. 1.3

[41] "Bluespec compiler (bsc)." https://github.com/B-Lang-org/bsc, 2026. 4.2.3

[42] "Chisel." https://www.chisel-lang.org/. Accessed: 2026-06-22. 1.3, 2, 4.2.3

[43] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020. 1.3

[44] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, p. 18–25, 2009. 1.3.1

[45] "AMD Vitis® HLS." 1.3.1, 2.1.2, 3.3.2, 6.1.2

[46] "Altera high level synthesis compiler." https://www.altera.com/products/development-tools/quartus-prime/hls-compiler. Accessed: 2026-06-22. 1.3.1, 2.1.3, 4.2.3, 4.7.3

[47] S. Basalama and J. Cong, "Stream-hls: Towards automatic dataflow acceleration," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '25, (New York, NY, USA), p. 103–114, Association for Computing Machinery, Feb. 2025. 1.3.1

[48] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: HLS-based Graph Processing Framework on FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, pp. 69–80, Association for Computing Machinery. 1.3.1

[49] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, pp. 51:1–51:42, Aug. 2022. 1.3.1

[50] "SYCL: C++ Programming for Heterogeneous Parallel Computing." https://www.khronos.org/sycl/. 1.3.1, 6.1.2

[51] "Abstract Parallel Programming Model for HLS • Vitis High-Level Synthesis User Guide (UG1399) • AMD Technical Information Portal." 1.3.1

[52] "Streaming data paradigm • vitis high-level synthesis user guide (ug1399) • reader • amd technical information portal." 1.3.1

[53] Z. Zhao and J. C. Hoe, "Using vivado-hls for structural design: a noc case study," 2020. 1.3.1, 4, 4.2.3, 4.7.2

[54] B. C. Schafer and Z. Wang, "High-Level Synthesis Design Space Exploration: Past, Present, and Future," vol. 39, no. 10, pp. 2628–2639. 1.3.2

[55] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "Autodse: Enabling software programmers to design efficient fpga accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, Feb. 2022. 1.3.2, 5.1.2

[56] H. Jun, H. Ye, H. Jeong, and D. Chen, "Autoscaledse: A scalable design space exploration engine for high-level synthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, June 2023.

[57] M. Yu, S. Huang, and D. Chen, "Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for fpga high-level synthesis," in *Intelligent Data Engineering and Automated Learning – IDEAL 2021: 22nd International Conference, IDEAL 2021, Manchester, UK, November 25–27, 2021, Proceedings*, (Berlin, Heidelberg), p. 524–536, Springer-Verlag, Nov. 2021.

[58] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Comba: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, p. 430–437, Nov. 2017.

[59] Y.-k. Choi, P. Zhang, P. Li, and J. Cong, "Hlscope+: Fast and accurate performance estimation for fpga hls," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 691–698, 2017. 1.3.2, 5.1.2

[60] D. Park and A. DeHon, "Refine: Runtime execution feedback for incremental evolution on fpga designs," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, (New

York, NY, USA), p. 108–118, Association for Computing Machinery, 2024. 1.3.2, 5.1.2, 5.7.4, 5.7.4

[61] S. Abi-Karam, R. Sarkar, S. Basalama, J. Cong, and C. Hao, "Fifoadvisor: A dse framework for automated fifo sizing of high-level synthesis designs," in *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 489–495, 2026. 1.3.2, 5.2b, 5.1.2, 5.5.2, 5.7.4

[62] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013. 1.3.2, 5, 5.2, 6.1.4

[63] "HLS Stream Library • Vitis High-Level Synthesis User Guide (UG1399) • AMD Technical Information Portal." 2.1.2, 2.3

[64] "The pipe Class and its Use." 2.1.2, 2.3

[65] V. Milutinović, J. Salom, N. Trifunovic, and R. Giorgi, "The DataFlow Paradigm," in *Guide to DataFlow Supercomputing: Basic Concepts, Case Studies, and a Detailed Example* (V. Milutinović, J. Salom, N. Trifunovic, and R. Giorgi, eds.), pp. 1–39, Springer International Publishing. 2.1.2

[66] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, p. 416–429, June 2017. 2.1.2, 2.4.2

[67] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," vol. 23, no. 1, pp. 20–24. 2.1.2

[68] AMD, "Vitis HLS." 2.1.3, 4.2.3

[69] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 211–218. 2.1.3

[70] "Scheduling." 2.1.4

[71] "Flushing Pipelines and Pipeline Types • Vitis High-Level Synthesis User Guide (UG1399) • AMD Technical Information Portal." 2.1.4

[72] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, p. 1099–1112, Feb. 2023. ISSN: 2378-203X. 2.2.2, 4.1, 5.7.3

[73] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, p. 69–74, Mar. 2008. 2.2.2

[74] S. Henning, A. Vogel, M. Leichtfried, O. Ertl, and R. Rabiser, "ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pp. 2–13, ACM. 2.2.4

[75] S. Neuendorffer and K. Vissers, "Streaming systems in fpgas," in *Embedded Computer Systems: Architectures, Modeling, and Simulation* (M. Bereković, N. Dimopoulos, and S. Wong, eds.), (Berlin, Heidelberg), p. 147–156, Springer, 2008. 2.4.2

[76] AMD, "Streaming data paradigm, vitis high-level synthesis user guide." 2.4.2

[77] "Sigma: SIEM Detection Format." https://sigmahq.io. 3, 3, 3.2.2

[78] S. Obla, T. Tracy, M. Beck, J. C. Hoe, K. Skadron, and W. U. Hassan, "Reconfigurable computing challenge: Rapidscan high-throughput parameterized hls-based streaming string matching library for fpgas," in *2026 IEEE 34th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 311–314, 2026. 1, 4.1

[79] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern cpus," p. 631–648, 2019. 3.1, 3.4.1

[80] "Splunk." https://www.splunk.com. 3.2.2

[81] "Graylog." https://graylog.org/. 3.2.2

[82] Z. Zhao, "Pigasus: Efficient Handling of Input-Dependent Streaming on FPGAs," 6 2022. 3.3.1

[83] "Amd alveo™ v80 compute accelerator." 3.3.2, 3.4.1

[84] "DARPA Engagement 5." https://github.com/darpa-i2o/Transparent-Computing/blob/master/README.md. 3.4.2

[85] "SigmaC: Sigma Conversion Tool." https://github.com/SigmaHQ/legacy-sigmatools. 3.4.2

[86] "Elasticsearch Query DSL." https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html. 3.4.2

[87] Z. Qian, P. Bogdan, C.-Y. Tsui, and R. Marculescu, "Performance evaluation of noc-based multicore systems: From traffic analysis to noc latency modeling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, May 2016. 4, 4.7.2

[88] D. Park, Z. Yao, Y. Xiao, and A. DeHon, "Asymmetry in butterfly fat tree fpga noc," in *2023 International Conference on Field Programmable Technology (ICFPT)*, p. 227–231, Dec. 2023. ISSN: 2837-0449. 4

[89] G. Malik, I. E. Lang, R. Pellizzoni, and N. Kapre, "Hopliteml: Evolving application customized fpga nocs with adaptable routers and regulators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, pp. 40:1–40:33, Aug. 2022.

[90] M. K. Papamichael, P. Milder, and J. C. Hoe, "Nautilus: fast automated ip design space search using guided genetic algorithms," in *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, (New York, NY, USA), p. 1–6, Association for Computing Machinery, 2015.

[91] P. Iff, M. Besta, M. Cavalcante, T. Fischer, L. Benini, and T. Hoefler, "Sparse hamming graph: A customizable network-on-chip topology," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, p. 1–6, 2023. 4

[92] R. R. Sunketa, M. A. Farooq, G. Gore, A. Boston, P.-E. Gaillardon, and A. Arora, "Openfpga-noc: Automated fabric and bitstream generation for noc-based fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 19, Mar. 2026. 1

[93] M. K. Papamichael and J. 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, FPGA '12, (New York, NY, USA), p. 37–46, Association for Computing Machinery, Feb. 2012. 4.1, 4.1, 4.1.2, 4.2.2, 4.3.1, 4.3.1, 4.6, 4.6.2, 4.6.3

[94] N. Kapre and J. Gray, "Hoplite: A deflection-routed directional torus noc for fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, Mar. 2017. 4.1, 4.1, 4.3.2

[95] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: Fpga nocs with provably stall-free fifos," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), p. 222–231, Association for Computing Machinery, Feb. 2019.

[96] S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitert: An efficient fpga noc for real-time applications," in *2017 International Conference on Field Programmable Technology (ICFPT)*, p. 64–71, Dec. 2017. 4.1

[97] N. Kapre and T. Krishna, "Fasttrack: Leveraging heterogeneous fpga wires to design low-cost high-performance soft nocs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, p. 739–751, 2018. ISSN: 2575-713X. 4.1

[98] J. Zhao, A. Agrawal, B. Nikolic, and K. Asanović, "Constellation: An open-source soc-capable noc generator," in *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, pp. 1–7, IEEE, 2022. 4.1, 4.1, 4.2.2

[99] T. Fischer, M. Rogenmoser, T. Benz, F. K. Gürkaynak, and L. Benini, "Floonoc: A 645-gb/s/link 0.15-pj/b/hop open-source noc with wide physical

links and end-to-end axi4 parallel multistream support," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 33, no. 4, pp. 1094–1107, 2025. 4.1, 4.1, 4.2.2

[100] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (score)," in *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, (Berlin, Heidelberg), p. 605–614, Springer-Verlag, Aug. 2000. 4.1

[101] K. Zyla, M. Liess, T. Wild, and A. Herkersdorf, "Hipernoc: A high-performance network-on-chip for flexible and scalable fpga-based smartnics," in *2025 Design, Automation Test in Europe Conference (DATE)*, p. 1–7, Mar. 2025. ISSN: 1558-1101. 4.1.1

[102] J. Chromczak, M. Wheeler, C. Chiasson, D. How, M. Langhammer, T. Vanderhoek, G. Zgheib, and I. Ganusov, "Architectural enhancements in intel® agilex™ fpgas," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, (New York, NY, USA), p. 140–149, Association for Computing Machinery, Feb. 2020. 4.1.1, 4.2.2, 5.5.2

[103] "Amba axi-stream protocol specification." https://developer.arm.com/documentation/ihi0051/b/. Accessed: 2026-06-22. 4.1.1

[104] "Use of rtl libraries for fpga." https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/developer-guide/2024-0/use-of-rtl-libraries-for-fpga.html. Accessed: 2026-06-22. 4.1.1, 4.7.3

[105] "Adding rtl blackbox functions • vitis high-level synthesis user guide (ug1399) • amd technical information portal." https://docs.amd.com/r/en-US/ug1399-vitis-hls/Adding-RTL-Blackbox-Functions. Accessed: 2026-06-22. 4.1.1

[106] W. J. Dally and B. Towles, "Route packets, not wires: on-chip inteconnection networks," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, (New York, NY, USA), p. 684–689, Association for Computing Machinery, 2001. 4.2.1

[107] J. Shalf, "The future of computing beyond moore's law," in *Feynman Lectures on Computation*, pp. 311–332, CRC Press, 2023. 4.2.2

[108] "Ram hdl coding techniques • vivado design suite user guide: Synthesis (ug901) • amd technical information portal." https://docs.amd.com/r/en-US/ug901-vivado-synthesis/RAM-HDL-Coding-Techniques. Accessed: 2026-06-22. 4.2.2

[109] "Inferring memory functions from hls code • quartus® prime pro edition user guide • altera documentation and resources center." https://docs.altera.com/r/docs/683082/25.1/quartus-prime-pro-edition-user-guide/inferring-memory-functions-from-hdl-code. Accessed: 2026-06-22. 4.2.2

[110] M. Abbas and V. Betz, "Latency insensitive design styles for fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, p. 360–3607, Aug. 2018. ISSN: 1946-1488. 4.2.2

[111] "Dual clock fifo example in verilog hdl • quartus® prime pro edition user guide • altera documentation and resources center." https://docs.altera.

com/r/docs/683082/25.1/quartus-prime-pro-edition-user-guide/dual-clock-fifo-example-in-verilog-hdl. Accessed: 2026-06-22. 4.2.2

[112] "Constant propagation (default) • vivado design suite user guide: Implementation (ug904) • amd technical information portal." https://docs.amd.com/r/en-US/ug904-vivado-implementation/Constant-Propagation-Default. Accessed: 2026-06-22. 4.2.3, 4.3.2

[113] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004. 4.3.1, 4.3.1, 4.3.1

[114] G. S. Malik and N. Kapre, "Enhancing butterfly fat tree nocs for fpgas with lightweight flow control," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 154–162, Apr. 2019. ISSN: 2576-2621. 4.3.2

[115] "Retiming in vivado synthesis." https://adaptivesupport.amd.com/s/article/934201?language=en_US. Accessed: 2026-06-22. 4.3.2

[116] "Hyper-retiming (facilitate register movement) • hyperflex architecture high-performance design handbook • altera documentation and resources center." https://docs.altera.com/r/docs/683353/25.1.1/hyperflex-architecture-high-performance-design-handbook/hyper-retiming-facilitate-register-movement. Accessed: 2026-06-22. 4.3.2

[117] "Claude code by anthropic | ai coding agent, terminal, ide." https://claude.com/product/claude-code. Accessed: 2026-06-22. 4.4

[118] "Google antigravity." https://antigravity.google, abstractNote=Google Antigravity - Build the new way. Accessed: 2026-06-22. 4.4

[119] "Verilator user's guide — verilator devel 5.049 documentation." https://verilator.org/guide/latest/index.html. Accessed: 2026-06-22. 4.6.1

[120] A. Gonzalez, A. Kolli, S. Khan, S. Liu, V. Dadu, S. Karandikar, J. Chang, K. Asanovic, and P. Ranganathan, "Profiling Hyperscale Big Data Processing," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–16, ACM. 4.7.1

[121] R. O. Nambiar and M. Poess, "The making of TPC-DS," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pp. 1049–1058, VLDB Endowment. 4.7.1

[122] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang, "An Empirical Evaluation of Columnar Storage Formats," vol. 17, no. 2, pp. 148–161. 4.7.1

[123] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable aggregation on multicore processors," in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pp. 1–9, Association for Computing Machinery. 4.7.1, 4.7.1

[124] O. Polychroniou and K. A. Ross, "High throughput heavy hitter aggregation for modern SIMD processors," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pp. 1–6, Association for Computing Machinery. 4.7.1

[125] A. Boutros and V. Betz, "Fpga architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, p. 4–29, 2021. 4.7.1, 5

[126] "DE10-Agilex Development and Education Kit." https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=142&No=1252. 4.7.4, 5.7.1

[127] L. Woods, Z. István, and G. Alonso, "Ibex: an intelligent storage engine with support for advanced sql offloading," *Proceedings of the VLDB Endowment*, vol. 7, p. 963–974, July 2014. 5

[128] Z. Zhao, J. Melber, S. Sahay, S. Obla, E. Nurvitadhi, and J. C. Hoe, "Exploiting the common case when accelerating input-dependent stream processing by fpga," *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1343–1355, 2023. 5

[129] S. Obla, J. C. Hoe, and B. Li, "Rapidq: Queueing-based performance modeling framework for rapid simulation and automated tuning of input-dependent streaming fpga pipelines," in *2026 IEEE International Symposium on Workload Characterization (IISWC)*, (Boulder, CO, USA), IEEE, September 2026. To appear. 1

[130] J. Kim and C. Hao, "Realprobe: An automated and lightweight performance profiler for in-fpga execution of high-level synthesis designs," in *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 10–18, May 2025. ISSN: 2576-2621. 5.1.2

[131] R. Sarkar, R. Paul, and C. C. Hao, "Lightningsimv2: Faster and scalable simulation for high-level synthesis via graph compilation and optimization,"

in *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 104–114, 2024. 5.1.2

[132] A. Honorat, M. Dardaillon, H. Miomandre, and J.-F. Nezan, "Automated buffer sizing of dataflow applications in a high-level synthesis workflow," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, pp. 8:1–8:26, Jan. 2024. 5.1.2

[133] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, (New York, NY, USA), p. 186–196, Association for Computing Machinery, Feb. 2020. 5.1.2

[134] J. Cong, M. Huang, and P. Zhang, "Combining computation and communication optimizations in system synthesis for streaming applications," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, FPGA '14, (New York, NY, USA), p. 213–222, Association for Computing Machinery, Feb. 2014. 5.1.2

[135] D. L. Jagerman, B. Balcıoglu, T. Altıok, and B. Melamed, "Mean waiting time approximations in the g/g/1 queue," *Queueing Systems*, vol. 46, no. 3, pp. 481–506, 2004. 5.3

[136] "SystemC Transaction Level Modeling (TLM)." https://systemc.org/overview/systemc-tlm/. 5.3

[137] F. Ghenassia, *Transaction level modeling with SystemC*. Springer, 2005. 5.3

[138] C. Lo and P. Chow, "Hierarchical modelling of generators in design-space exploration," in *2020 IEEE 28th Annual International Symposium on Field-*

Programmable Custom Computing Machines (FCCM), p. 186–194, May 2020. 5.5.2

[139] Y. Zhang, Z. Zhou, S. Elnikety, and C. Delimitrou, "Ursa: Lightweight resource management for cloud-native microservices," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, p. 954–969, Mar. 2024. ISSN: 2378-203X. 5.5.3

[140] "Snort ruleset." https://www.snort.org/downloads/#ruledownloads. 5.6.2

[141] "Stratosphere. stratosphere laboratory datasets." https://www.stratosphereips.org/datasetsoverview, 2015. 5.6.2

[142] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 127–136, ACM. 6.1.1

[143] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The Sparse Abstract Machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pp. 710–726, Association for Computing Machinery. 6.1.1

[144] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pp. 267–280, ACM. 6.1.3

[145] Y. Zha and J. Li, "Virtualizing fpgas in the cloud," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Program-*

ming Languages and Operating Systems, ASPLOS '20, (New York, NY, USA), p. 845–858, Association for Computing Machinery, Mar. 2020. 6.1.3

[146] L. Guo, P. Maidee, Y. Zhou, C. Lavin, E. Hung, W. Li, J. Lau, W. Qiao, Y. Chi, L. Song, Y. Xiao, A. Kaviani, Z. Zhang, and J. Cong, "Rapidstream 2.0: Automated parallel implementation of latency–insensitive fpga designs through partial reconfiguration," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 4, pp. 59:1–59:30, 2023.

[147] D. Park, Y. Xiao, and A. DeHon, "Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–10.

[148] J. Chen, A. Panda, H. Unnibhavi, A. Koshiba, and P. Bhatotia, "Shell: A Microkernel-based FPGA Shell Architecture," pp. 2103–2126. 6.1.3