Automated Interconnect Synthesis and Optimization for FPGAs

by

Alexandr A. Rodionov

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Automated Interconnect Synthesis and Optimization for FPGAs

Alexandr A. Rodionov
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2019

One of the key challenges for the FPGA industry going forward is to make the task of designing hardware easier. A significant portion of that design task is the creation of the interconnect pathways between functional modules. Interconnect synthesis tools and Network-on-Chip architectures have been developed to solve this problem. They allow the designer to specify desired connectivity at an abstract logical level, and automate the physical details of the interconnect implementation. In this work, we introduce a new approach to automatically synthesizing interconnect, based on the composition of many simple hardware primitives that each perform an elementary routing, translation, storage, or delay function. This approach is embodied in a new design tool called GENIE, which produces RTL implementations of complete systems containing a designer's functional modules connected with automatically-generated interconnect, whose generation is guided by an initial logical system specification. GENIE leverages its piecemeal approach to interconnect generation to offer more automation and optimization capabilities to the designer than are provided by existing tools. One such capability is the generation of interconnect for "fine-grained" applications in which the overall small size of the system, need for simple communication protocols, and tight latency requirements would incur too high of a performance or area penalty to make productive use of automation with existing methods. We also introduce a new type of user design specification called *synchronization constraints* that allow GENIE to build low-cost, backpressure-free interconnect for applications with fixed-latency modules and achieve correct cycle-level timing using a minimum amount of automatically-inserted delay elements. GENIE's other optimization capabilities include automatic clock domain crossing insertion, interconnect pipelining, and a limited form of network topology generation guided by application performance requirements. Using both FPGA synthesis and RTL simulation of the generated systems, we evaluate GENIE and its features using two applications that serve as extensive and detailed case studies. We also compare GENIE against other system generation tools, NoC architectures, and hand-written RTL interconnect implementations, and are able to show equivalent or higher quality implementations with lower design effort.

Acknowledgements

I would like to thank my supervisor Jonathan Rose for guiding me, supporting me, and keeping me on track during the many years it has taken to produce this work. I am also fortunate to have been surrounded by colleagues working on similar or related problems within the FPGA space whose expertise I could draw upon. Thanks go out to Henry Wong for his insights into FPGA architecture and assistance with optimizing the design of my interconnect primitives, as well as Xander Chin, Mohamed Abdelfattah, Vince Mirian, Andrew Canis, Mustafa Abbas, Kevin Murray, and James Choi, whose work intersected with mine in some way and led to the sharing of ideas and compelling use cases.

This research was also supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) as well as CMC Microsystems which provided licenses for various CAD tools.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) offer an alternative to ASIC manufacturing, providing reduced cost, faster turnaround times, and broader access to digital hardware design to a wider audience. However, designing hardware that targets FPGAs still remains relatively difficult and time-consuming compared to writing software. With software, application behavior is specified as a series of instructions that must be executed. In hardware design, behavior is a consequence of physical structure and its connectivity, and the translation from behavior to structure is a nontrivial task with an immense degree of freedom. Many physical implementations can exist for the same logical functionality. These implementation choices have physical consequences – differences in area and resource usage, and differences in achievable performance due to signal propagation delays. These are some of the extra considerations, on top of planning an application's desired behavior, that make hardware design challenging.

The primary motivation for this work is to make hardware design for FPGAs easier. There are many different ways to approach this problem, but a common thread among them is the raising of the level of design abstraction. At the time of this writing, the primary input to FPGA compilation toolchains is an RTL (Register Transfer Level) description of the design written in a Hardware Description Language (HDL) such as Verilog or VHDL. Although higher-level design tools exist, they must first emit RTL, as this is the lowest-common-denominator representation accepted by the synthesis and technology mapping front-ends of FPGA toolchains. RTL specifies a design in terms of registers and combinational logic, with data moving from register to register every clock cycle, possibly passing through combinational logic that computes and transforms the data. This provides a fairly low level of abstraction, but one that is able to map to the underlying FPGA resources with no ambiguity in terms of clock cycle-level timing. The explicitness of cycle-level timing makes mapping application behavior to RTL challenging.

Raising the level of design abstraction above RTL requires creating a tool that generates RTL to be consumed by the FPGA implementation toolchain. Such a tool could accept some higher-level abstraction or design methodology, and thus streamline the user's design entry experience. Many broad categories of such tools already exist. One popular category of hardware design abstractions is High-Level Synthesis (HLS). The goal of HLS tools is to allow the designer to describe their application using languages like C/C++ or OpenCL, focusing entirely on specifying behaviour, much like a software programmer would. The hardware implementation's structure, connectivity, and cycle-level timing are no longer the designer's concern, and are generated automatically by the tool. While HLS is successful at lowering the effort required to create a correct and functional design, it is fundamentally a behavioral

rather than structural design paradigm, meaning that it may be challenging to guide the underlying hardware implementation to optimize the result for resource utilization or performance.

System integration tools are another category of hardware design abstractions, and are the focus of this thesis. They view an application as a series of functional modules that communicate with each other. This structural description mirrors the physical nature of hardware, and that of RTL, with the key difference being that the links between modules represent an abstract desire for communication instead of concrete physical implementations. The physical implementation of these links, the *interconnect*, is automatically generated by the tool, along with an instantiation of the functional modules. This approach could also be used together with other design abstractions. In theory, the functional modules could be created using any method, such as HLS or RTL, as long as they have the appropriate signal interface with the interconnect. Given that functional modules are opaque black boxes that are provided ready-made to a system integration tool, their instantiation in the final system is a convenient, if trivial automation, and we can consider the main work performed by system integration tools to be the synthesis of interconnect.

Interconnect itself is hardware, whose purpose within the larger application is to implement communication between functional modules. The possible design space of such circuits is large, and they can be challenging to implement correctly. At the extreme, a single wire is the simplest form of interconnect, but it can only enable point-to-point unidirectional communications between one source and one sink with zero cycles of delay. In general, the communication needs of an application are more complex, and require the interconnect to have some logic and functionality of its own. For example, if there is a requirement for a source to communicate with one of many sinks, the complexity of the interconnect is increased. In this case, there must exist some logic in the interconnect which takes the source's specification of which sink it wishes to send data to (via some form of address signal) and uses it to direct traffic to the appropriate destination. An even more complex scenario is if multiple sources wish to communicate with one sink. An arbiter circuit may be required to permit only one source access to the sink, and to stall the others. In order to have well-performing interconnect, physical design must be taken into consideration, such as the pipelining of links. In some cases, registers may need to be inserted into the interconnect to balance the latencies of converging paths, and enable correct circuit operation. For either performance or correctness, there may be many legal locations to insert registers, each with differing total area costs. Non-trivial interconnect is also necessary when interfacing one clock domain with another, as is the problem of selecting the locations of clock domain boundaries. This large design space, coupled with the possibility of evolving communication requirements, motivates the need for automated generation of interconnect.

Our central hypothesis is that FPGA hardware design effort can be reduced by advancing the capabilities of automated interconnect synthesis tools. Reduction of design effort can be achieved in multiple ways, for example:

- Enabling the use of automated interconnect synthesis tools for types of designs which could not previously be targeted in a productive manner.
- Achieving better performance metrics (clock frequency, area usage) than with existing tools.
- Providing the user an interface protocol that reduces the amount of effort required to adapt their functional modules to work with automated interconnect synthesis.

- Adding optimization capabilities to an automated interconnect synthesis flow which enable a rapid exploration of the design space via the changing of a few high-level specifications.

We explore our hypothesis, and these avenues to achieving a reduction in design effort, by developing a new tool called GENIE – the GENeric Interconnect Engine. One of its capabilities is expanding the set of use cases in which automated interconnect synthesis yields productivity gains. Existing tools implicitly require that functional modules have a minimum size/complexity, or *granularity*, to make automatic generation worthwhile. We develop a high-level system representation and automated synthesis framework that allows the expression of *fine-grained* systems, which are smaller than the minimum size/complexity required by existing tools. For these types of systems, the resulting resource utilization and performance of our generated hardware is superior, and approaches that of hand-coded RTL.

In addition to design granularity, another assumption made by existing design concerns the cycle-level timing of hardware. Due to current tools' focus on interoperability with Intellectual Property (IP) cores, the signal interfaces between functional modules and interconnect tend to be standardized. To allow flexibility in timing, these standard interfaces feature latency-insensitivity, allowing for a variable and possibly unknown number of clock cycles to complete a transmission. The downside of latency-insensitivity is that it incurs interconnect overhead, narrowing the list of functional module design styles that yield productive usage of system integration tools. As part of our work, we create a means for the designer to include fixed-latency modules in their systems without the use of latency-insensitive protocols, allowing for the generation of lower-cost interconnect. This is achieved through the creation of a new class of timing constraints that specify cycle-level data arrival relationships on high-level communication links.

The two themes and general goals present throughout our research are automation and optimization. Automation of interconnect synthesis concerns the creation of the interconnect itself – by following a high-level specification from the designer, the tool frees them from the task of specifying the hardware manually at the RTL level. In addition to generating hardware that is correct, it is important that resource utilization and performance are sacrificed as little as possible in exchange for these productivity gains.

Optimization complements automation, and we will explore its role in interconnect synthesis. In the vast design space of interconnect hardware, certain design points yield better or worse performance and area utilization for a given application's communication requirements. Two of the notable examples we explore are the optimization of interconnect topology, and the automatic insertion of registers. We present novel algorithms to perform these tasks as well as an appropriate set of user specification inputs that define the optimization goals.

The contributions of this work are embodied by our GENIE tool, and include:

- A complete open-source end-to-end system generation flow that begins with high-level specification and produces synthesizable SystemVerilog output.
- The ability to automatically synthesize interconnect for *fine-grained* systems in a productive manner.
- The ability to create backpressure-free interconnect for systems containing fixed-latency functional modules by automating cycle-accurate synchronization, and introducing a new type of user specification, *synchronization constraints*, that control this flow.

- An automatic interconnect pipelining flow based on a new algorithm.
- An automatic topology generation flow that attempts to reduce interconnect area while satisfying application performance requirements.

The remainder of this thesis is organized as follows: Chapter 2 provides related background on existing interconnect synthesis tools and techniques. Chapter 3 introduces the GENIE system integration tool, describing its inputs, outputs, and synthesis flow at a high level. This is complemented by Chapter 4, which describes the hardware micro-architecture of the interconnect that GENIE synthesizes. Chapter 5 introduces the two applications that will be used as test cases and benchmarks for evaluating the work presented in Chapters 6-9, which contain detailed descriptions of the features of GENIE that realize our main contributions. Finally, Chapter 10 concludes.

Chapter 2

Background

This chapter reviews the context of our work and previous work related to our research. One of the main contributions of this work is the creation of a new system integration tool, so we begin with an overview of existing tools that perform a similar function. Next, we review related work in the study and design of on-chip networks, specifically those that target FPGAs. The remaining background covers specialized aspects of interconnect generation techniques, such as topology selection and automatic pipelining.

2.1 System Integration Tools

The inputs and outputs of a generic system integration tool are shown in Figure 2.1. System integration tools generate an RTL description of a complete interconnected system from a logical system specification. This specification defines instances of user-provided functional modules and logical links between them. Logical links originate and terminate at *interfaces*, which are higher-level groupings of HDL ports of the functional modules. The signals within an interface have roles in communication, such as flow control or carrying data, and the semantics and timing of these signal roles conform to a protocol that is well-defined and understood by the tool. From the system specification, the tool outputs an HDL module that contains instantiations of the user's functional modules connected by auto-generated physical interconnect that realizes the desired logical connectivity. For example, in Figure 2.1, the system specifi-

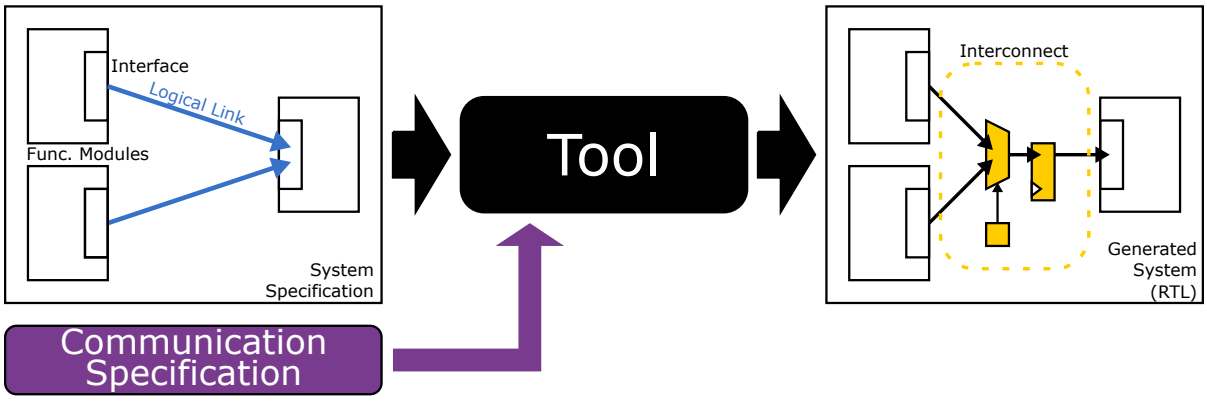

Figure 2.1: Inputs and outputs of a system integration tool

cation contains three functional modules, each with one interface. Two logical links are defined between these interfaces: one each from the left modules, sending to the right module. This logical specification is realized by the tool as a 2-to-1 multiplexer controlled by an arbiter, followed by a register.

In addition to the structural system specification, metadata about the nature of the communication between functional modules (the "Communication Specification" in Figure 2.1) may also be provided as input. Examples include the bandwidth or latency requirements of specific links, a customized weighting of arbitration shares, or information relating to the timing of transmissions over logical links. This information may be used by the tool to guide the generation of interconnect.

The above description of a generic system integration tool has many existing concrete realizations, including those that target FPGAs. One of the motivations for our research is the observation that these existing products have been specialized to fit a narrow (but very common) set of use cases within the Electronic Design Automation market, limiting their potential for automation and optimization of interconnect. We explore these existing tools, and their limitations, in the next sections.

2.1.1 FPGA Vendor Tools

The major FPGA vendors provide design software suites for their devices, and system integration tools are included among these. Intel (formerly Altera) has Qsys [7], Xilinx has the Vivado IP Integrator [69], and Lattice has LatticeMico [47]. They provide a GUI-based means of design entry, as well as scripting-based (TCL) to instantiate modules and define links.

Modularity and reusability are heavily emphasized, with the goal being to allow a user to create a working system without writing any HDL. Their main use case is the creation of Systems-on-Chip (SoC) containing one or more soft microprocessors, memory blocks, and peripherals. A large library of these and other Intellectual Property (IP) blocks is provided, although it is possible to include custom HDL modules as well.

As a result of this SoC-centric focus, existing FPGA system integration tools are built mainly around memory-mapped communication. This is very natural and convenient for processors, but, as will be discussed later, is limiting in the general case. The main interface and communication protocols expected by FPGA system integration tools are standard protocols such as ARM AMBA AXI [9] or Avalon-MM [34]. Functional modules use these to interface with the tool-generated interconnect fabrics.

A secondary, but still popular, use case is streaming generic data between components. For example, a series of individual modules, each performing a different kind transformation on input data, can be daisy-chained together to perform a complex signal processing function. Instead of memory-mapped protocols, streaming protocols like AXI Streaming [8] and Avalon-ST [34] are used. These are less structured, often just consisting of the raw data and handshaking signals for flow control. As a result, the interconnect for such a point-to-point link is often implemented as wires with no logic or registers, and so the degree of automation performed by a tool is low. More sophisticated interconnect must be explicitly instantiated by the user in the same fashion as the functional modules that perform application-related tasks.

2.2 Networks-on-Chip

NoCs [36] are a general class of interconnect. Taken to the extreme, one could claim that any form of interconnect, down to a single wire, can be classified as a NoC. However, NoCs are usually understood to refer to packet-switched networks containing switches, buffers and wires arranged in a nontrivial

topology such as a mesh or a ring. Furthermore, multiple end-to-end communications flows between functional modules share the same physical resources in a time-multiplexed manner. Virtual channels (VCs) [27, 28] and related mechanisms help ensure that no one flow can stall overall forward progress on a shared physical link.

The development of NoCs was motivated by concerns over the scalability of the performance of shared buses in multi-processor systems-on-chip (MPSoCs). The use of NoCs on FPGAs is a distinct, and more recent development than their general usage in ASIC MPSoCs. Projects such as the CONNECT network generator [58] and the Split/Merge [32] NoC are FPGA-specific NoC architectures. Their designers recognized that the relative costs of primitive circuit elements such as wires and multiplexers are radically different when designing for FPGAs versus bare silicon [68]. Thus, rather than simply porting the HDL code for ASIC-targeted NoCs to FPGAs, a ground-up redesign was needed that took these differences into account. Examples of these differences include lower achievable clock frequencies, the ratio of the availability of routing and logic resources, and the expense of large multiplexers and on-chip storage. The general trend of FPGA-targeted NoCs is "less is more" – favouring the composition of simple circuit elements that map well to FPGA resources. FPGA-based NoCs provide important lessons for building our own interconnect architecture, and three examples are described next in detail.

2.2.1 CONNECT

The CONfigurable Network Creation Tool, by Papamichael and Hoe [58], comprises both an FPGA NoC architecture as well as an automated tool for customizing and generating networks built using this architecture. The architecture favours wider and slower links between routers compared to ASIC-based NoCs, preferring to use more of the existing wiring on an FPGA rather than attempt to time-multiplex data over fewer wires at higher clock frequencies. Similarly, routers only have a single pipeline stage, with the stated goal of taking advantage of the lower target clock frequency achievable on FPGAs. In the results of their original paper, they mainly achieved an area reduction over the baseline ASIC NoC design, combined with a frequency reduction. They claim to recover the lost performance by widening their network from 32 to 128 bits while maintaining the same clock frequency. On modern FPGAs, increasing the width causes a clock frequency penalty due to place and route effects, when the topology is sufficiently complex [67]. Recovering bandwidth purely by focusing on width, and neglecting clock frequency, may no longer be as viable a solution. In the design of our network, we still elect to use as wide a bus width as is required to transmit all the user's data signals, plus control signals, as a single word. However, we do this mainly to escape the latency overheads of serialization/deserialization and enable a simple design approaching the semantics of bare RTL bus wires.

In CONNECT, buffers are kept small so that they can fit in an FPGA's distributed RAM resources. These are available at more locations throughout the chip, rather than the comparatively large block RAMs which are few and have their locations centralized to certain columns on the FPGA. The inclusion of buffering makes CONNECT a fairly conventional NoC architecture compared to some recent ones like Hoplite[40].

An important observation made by this work is that the reconfigurability of FPGAs lends well to customizing interconnect to meet a specific application's communication requirements. This is reflected in the web-based generation tool, which allows a user to customize many aspects of the network and architecture – in far more combinations than would be feasible to individually verify for a conservative ASIC-based target. The network topology can be chosen from a predefined list, or a custom one created

with a GUI. Link width and flow control scheme, and buffer depth and allocation policy, are examples of other settings that can be changed.

Ultimately, the generator emits synthesizable Verilog. There are no system integration features – one chooses a topology and the number of endpoints, and only the interconnect is created. From an ease-of-use perspective, this does not fully address the complete end-to-end system generation problem.

2.2.2 Split/Merge NoC

This style of FPGA NoC architecture utilizes two interconnect routing primitives named Split and Merge, which first appeared in [41] and further developed by Huan and DeHon [32]. Split nodes have one input and many outputs, and forward an input packet to one of the output ports based on its intended destination. Merge nodes have many inputs and one output, and arbitrate among many competing inputs to forward one of them to the output. Thus, instead of using a traditional monolithic "router" that contains many input ports and many output ports, the tasks of arbitration and distribution of packets are performed by separate, decomposed primitives. This separation provides more opportunities for pipelining the links between interconnect primitives, yielding higher clock frequencies than the single-stage router design of the CONNECT architecture. Networks with arbitrary topologies can be created by choosing a suitable arrangement of Split and Merge nodes.

In the abstract of the Huan/DeHon paper, the development of this architecture seems to be a direct response to the low clock frequencies achieved by CONNECT. The main focus was to increase the amount of pipelining to achieve a higher clock frequency. Compared to CONNECT, Split/Merge was able to achieve 3x the clock frequency with a 4 cycle pipeline, consuming up to 37% more area, most of which is in memory rather than LUTs. Just like CONNECT, the architecture maintains the NoC tradition of buffering packets at the input of each primitive. Unlike CONNECT, header information is sent in a separate flit preceding the data flits, as with higher frequencies it is no longer a priority to maximize data widths. However, virtual channels are absent, instead favouring the use of separate physical networks, which in theory should be beneficial for reducing hardware complexity and cycle time, but at the expense of area.

Our research is heavily inspired by the Split/Merge architecture, but lacks the mandatory buffering that would otherwise increase baseline latency and consume memory resources. However, we found Split/Merge to be a good starting point for its insight into decomposing network functionality into simple primitives. The original work also did not provide a tool for generating either interconnect or a full system with functional modules.

2.2.3 Hoplite

Hoplite [40] is a minimalist FPGA NoC architecture. It lacks any buffering, instead deflecting traffic through unused channels within a torus topology. A packet can continue circulating within the network until it is able to progress to its original destination. Hoplite routers are designed to be as simple as possible and to map well to the underlying FPGA logic resources. These design choices yield significant area savings and clock frequency gains over the previous two FPGA NoC architectures mentioned in this section. A Hoplite router has 1/26 the LUT usage of a comparable CONNECT router and 1/30 the usage of a Split+Merge router. The clock frequency gains over each were 3.3x and 1.55x respectively.

The Hoplite paper explores the design of the network micro-architecture in detail and ways of design-

ing it around vendor-specific FPGA LUT architectures, which is a level of FPGA-specific targeting that was not addressed in CONNECT or Split/Merge. While this attention to architecture yields impressive clock frequency and area gains, deflection routing has a negative impact on latency. A Hoplite deflection-routed torus has an average latency two orders of magnitude greater than a comparable buffered mesh or torus, at a 0.1 injection rate of uniform random traffic on a 10×10 grid of processing elements. Throughput can be recovered by using the equivalent area to instantiate more Hoplite networks and taking advantage of the low resource and higher achievable clock frequency. One implication of these results was that Hoplite deflection-routed toruses may be a more attractive option than buffered networks for systems where the functional modules themselves are small (at or below 2000 logic elements by their account). We also explore this complexity-matching between functional modules and interconnect in Chapter 6, where we explore building suitable interconnect between finer-granularity functional modules.

A follow-up work, FastTrack [38], is an improvement on the Hoplite architecture, aiming to make even better use of available FPGA wiring than its predecessor. It still uses deflection routing, but routers now have additional connections to non-adjacent routers via "express links" that allow skipping of hops. These links are designed to map to longer wires on the FPGA such that the NoC architecture leverages the diversity of wire lengths present in the underlying FPGA fabric. Throughput gains of 1.5-2.5x are observed over baseline Hoplite at previously-problematic injection rates above 0.1, with average latency reduction in the order of 2-7x depending on injection rate and router configuration.

Deflection routing is feasible only for certain topologies and practically usable at lower injection rates [53]. This is why we will opt for a "primitive decomposition" approach like Split/Merge as a baseline architecture for our work, as there are many viable combinations of primitives that yield functional topologies. However, we also choose to eliminate buffers as a mandatory part of our architecture, adopting this characteristic of Hoplite and deflection-routed networks while diverging from the baseline Split/Merge design. This is necessary to match the latencies achievable with hand-coded RTL in applications where buffering is not necessary. The Hoplite work inspires a strong attention to designing network primitives that map well to FPGA resources, although we do not consider the underlying wiring architecture like FastTrack does.

2.3 Topology Synthesis

A network's topology is defined by the arrangement of its routing primitives and wires. It is a key design parameter in the design of interconnect. Part of our research concerns the development of a new method of automatically generating a network topology based on an application's communication requirements. This will be presented in Chapter 9.

The fundamental problem of topology synthesis is the automation of the design of a network's topology in order to minimize, maximize, or otherwise satisfy some objective. We will see a common pattern in our cited works on this topic in the nature of this objective: the generated topology must satisfy a hard set of criteria (for example, latency or bandwidth) while performing a best-effort minimization of some other metric such as area or power. It should also be mentioned that since networks exist as abstract mathematical entities as well as concrete things at all physical scales, we will limit our interest to those techniques that target on-chip interconnect on either FPGAs or ASICs. Much of the early work on topology synthesis was first in the realm of ASICs, focusing on either packet-switched NoCs or shared

bus architectures, so we will examine those first.

One such early work is by Pinto et al. [59] which does not consider the physical shape or layout of its functional modules or interconnect primitives, and therefore forms the topology synthesis problem in a more abstract mathematical manner. In their formulation, there exists a communication graph representing the desired logical links between functional modules, as well as hard bandwidth requirements for each link. There also exists a characterized properties of interconnect primitives, which are nodes and links with annotated costs (maximum length/bandwidth for links, area/power for switches). The problem becomes one to find an implementation graph, consisting of interconnect primitives, that the logical communication graph can map to. This is stated as an exact problem, and the remainder of the work deals with ways in which to reduce its complexity to make it more computationally tractable. This approach is frequently cited by later works as a pure abstract formulation of the problem.

Srinavasan et al. [66] developed a flow which consumes a list of rectangular functional modules along with a communication graph specifying the flows between the modules as well as their required bandwidths and latencies. This is now an example of an approach that begins to take into account physical dimensions of modules. The objective is to generate a floorplanned system with routers, functional modules, and a mapping of logical traffic flows to physical network links such that bandwidth/latency constraints are satisfied while attempting to minimize power and router count. The topology synthesis flow invokes a floorplanner to lay out the functional modules in a cost-sensitive way. Routers are implied to exist at the corners of the placed functional modules, and the shape/size of the routers is assumed to be negligible. An ILP formulation assigns transmissions to network links. This is an example of a "one-shot" solution that does not need to iterate back to previous steps to evaluate a broader set of topological possibilities.

As a counter-example, Murali et al. [54] address some of the shortcomings of the previously mentioned flow and consider the shape and size of routers, as well as whether or not timing closure is possible in the generated solution. Their approach is iterative: given N functional modules, N different topologies are independently explored, ranging from one in which all modules are connected to one router, to one where each module gets its own router. For each topology, the N nodes are partitioned among the routers by min-cut of required transmission bandwidth, transmissions are assigned to physical links, and the full system is floorplanned to measure its cost and to check for timing violations. Once again, there is a hard constraint to satisfy (bandwidth for transmissions) and a best-effort attempt to minimize a choice of power or transmission hop count. The best out of N topologies is selected, and it is not clear up front which one that will be. This is an important observation – once the physical parameters of interconnect primitives are considered (area, performance, power), as well as how they change with parameterization (for example, the number of ports on a router), it is less clear whether a design with fewer expensive primitives will be better than one with more abundant but simple primitives. This issue of difficult-to-estimate cost impact of topological choices will also be a theme that appears in our work on topology optimization in Chapter 9.

Another example of an iterative process is Xu et al. [71]. The important difference from the previous work is the approach used to construct the candidate topologies. Rather than walking through N possibilities, a greedy hierarchical partitioning is performed of the communicating nodes, where each partition receives a router. Different order traversal produces different results, and more than one is considered and its cost fully evaluated.

One common feature of interconnection networks is the assumption that multiple logical transmissions

will share a physical link. It is then required to ensure that the utilization of that link is not exceeded. Rather than using an average measure such as bandwidth, it is possible to make more detailed time-domain traffic characterizations that consider the possibility that two links may fully utilize the same channel and never compete as long as they do not overlap in time.

One such example of this is the work by Murali *et al.* [55]. Their approach emits a partial crossbar design (rather than a packet-switched NoC, as we have been mostly seeing up to this point) that minimizes the latency caused by temporal overlap of transmissions sharing the network. This is achieved by beginning with a full crossbar and determining which nodes can share network resources while still satisfying performance requirements, so it is also an example of an iterative approach that gradually refines its topology. Temporal overlap is constantly evaluated by referring to traces obtained from an initial time-based simulation of the application. The space of candidate crossbar configurations is tested via binary search.

A similar topology synthesis approach by Cong *et al.* [19] generates networks optimized for power and area containing arbitrary-radix routers. Their flow begins with a topology that exactly matches the logical connectivity, and iteratively refines it such that network cost is reduced at the expense of resource sharing and performance degradation. Groups of transmissions can be mapped to a lower-cost shared bus when the user explicitly marks them as mutually temporally exclusive. This explicit marking of temporally exclusive transmissions, rather than simulation, is the most similar to what we will adopt in our approach.

Rather than beginning with an initial topology that exactly matches logical connectivity, another possibility is to begin at a single switch that handles all communications, and then iteratively split it into several switches until conditions are satisfied. This is what Ho *et al.* [30] proposed. Their technique generates application-specific network topologies that have zero contention between transmissions, guided by knowledge of which transmissions overlap in time. The mapping of logical links to physical links is done with simulated annealing and approximate graph colouring, to reduce the complexity of the exact mapping problem.

Despite being ASIC-targeted, there are useful general approaches that can be found in the works we have cited thus far. Notably, the need to evaluate multiple candidate topologies to estimate the exact impact of topological choices, which seems inevitable once simplifying assumptions are no longer made about the end-to-end system generation flow. There were also many different approaches as to how to iterate over the space of possible topologies to evaluate as candidates. Nevertheless, there do also exist FPGA-targeted topology synthesis approaches.

For example, Kapre et al. have studied the performance/area tradeoffs of Butterfly Fat Tree (BFT) topologies on FPGAs [41, 39]. They build BFT networks using two types of switches (t and π), each consisting of Split and Merge primitives [41, 32] (that are also used in our work) rather than traditional NoC routers. By changing the ratio of t to π switches (controlled by a dimensionless Rent parameter), the bisection bandwidth of the BFT topology can be adjusted to match a particular application's needs. Note that rather than use bandwidth and latency constraints directly, a single abstract parameter is exposed to the user to turn, and to explore the space of generated topologies to evaluate which ones satisfy their needs. This conveniently sidesteps the lack of exact physical control and characterization when synthesizing for FPGAs. There are exceptions, such as the authors' related work on Hoplite, where the interconnect was manually and explicitly placed and routed by hand into a regular predictable pattern.

The ShrinkWrap Compiler [18] is a tool that generates optimized interconnects for applications cre-

ated using the CoRAM framework [17], which automates the creation of memory hierarchies. ShrinkWrap creates the interconnect between the auto-generated memory components and the user's application modules, optimized according to the application's communication patterns, which are explicitly specified by writing C-based control threads. Similar to our work, the ShrinkWrap interconnect uses segregated unidirectional tree-based networks, but the overall flow is specifically tailored to the generation of memory hierarchies.

2.4 Automatic Pipelining

In the GENIE tool described in this research, we automatically insert registers into the interconnect for performance. There is a body of existing work in retiming and automatic pipelining of circuits in more general contexts [23].

Retiming [49] is a technique to move combinational logic relative to sequential elements such that register-to-register combinational delays are balanced. However, it cannot insert new registers in a way that would change the total number of pipeline stages. Similarly, there exist techniques that do not change the number of pipeline stages, but instead provide automation in the form of generating global pipeline control logic [43, 56].

Elastic buffers are FIFO structures that can be implemented using registers, or the two latches within a register [14]. Connecting functional modules via elastic buffers enables one style of latency-insensitive design [13] in which there is no centralized control of pipelines. Rather, handshaking (ready, valid) signals are used to indicate the presence of data or to apply backpressure locally between neighbouring functional units and elastic buffers. If a circuit is designed in a latency-insensitive manner, algorithms and transformations exist [57, 37] that are able to automatically insert or remove elastic buffers, thus automatically pipelining the circuit.

On FPGAs, elastic buffers are more expensive than plain registers. Since the master/slave latches within each register are not directly accessible, at least two whole registers must be recruited to form an elastic buffer. This makes a latency-insensitive design style prohibitive for complexity-sensitive applications. In these situations, another purpose of inserting registers is for maintaining correctness, rather than performance. The underlying *buffer minimization problem* [31, 29] aims to ensure the correct number of cycles of delay between functional units by inserting the fewest number of registers. It can be formulated as an integer programming problem, but has had refinements made by others to improve its asymptotic runtime complexity through decomposition approaches [15] or graph-theoretic reformulations [11]. The buffer minimization problem has also found use in High-Level Synthesis (HLS) tools [21, 12]. There, hardware modules representing operations in a control/data flow graph are scheduled to begin at a certain clock cycle in order to satisfy dependencies, which naturally leads to the problem of determining the optimal locations for delay element insertion.

The cited works on automatic pipelining either insert registers for performance (in a latency-insensitive system), or insert registers for correctness (in statically-scheduled systems). One of our contributions is a combined approach that handles both design styles coexisting in the same system.

Chapter 3

The GENeric Interconnect Engine

This chapter presents a high-level overview of the GENIE interconnect synthesis and system integration tool, which is the concrete realization of the automation and optimization techniques that comprise the contributions of our research.

GENIE builds systems that contain user-created functional modules connected with auto-generated interconnect. Each system is built from an input specification provided by the user, which describes the interfaces of the system's functional modules and establishes logical links that represent transmissions between them. As shown in Figure 3.1, the input is specified programmatically from a script file written in the Lua [60] programming language. In contrast with a purely static data-like representation such as an XML or JSON, executing a programming language as input allows the user to leverage looping constructs, variable declarations, and command line argument parsing to enable complex design space exploration and automated instantiation of modules. Lua was chosen over similar languages such as Python for its ease of embedding – the entire Lua interpreter and standard library consist of several `.c` and `.h` files that can be directly included as part of the GENIE host application.

This host application, labeled "GENIE Tool" in Figure 3.1, is merely a thin wrapper that executes Lua scripts and forwards the calls to an underlying C++ library containing GENIE's actual interconnect synthesis and system building flow. This flow library could conceivably be used by other design tools (such as HLS compilers) that require the synthesis of interconnect as part of their flow. In this case, such a tool would make the C++ library calls directly, bypassing the Lua language layer and the need for passing any files to GENIE.

The GENIE flow implements each system described in the user's input specification. Logical links are realized into interconnect that is optimized for area, subject to optionally-provided performance constraints. Optimization decisions are informed by estimating the interconnect's area and timing characteristics. This is done by querying a database of interconnect primitives, which is stored on-disk and is shipped with the tool. The database contains area and timing models for many parameterizations of each GENIE interconnect primitive. The output of the flow is a SystemVerilog module for each system, containing instantiations of the user's modules and of GENIE's interconnect primitives, which are also shipped with the tool.

The remainder of this chapter describes, in more detail, the input specification, the nature of GENIE's interconnect, and the synthesis flow itself. We begin with an overview of the philosophy of the design choices made when developing the aforementioned aspects.

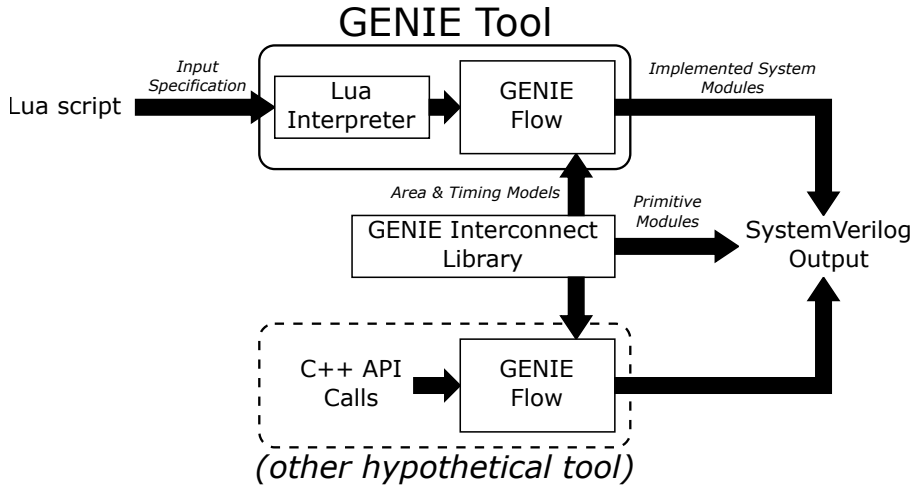

Figure 3.1: High-level overview of the GENIE tool architecture

3.1 Design Philosophy

The development of GENIE was motivated by gaps in existing interconnect synthesis approaches, and by opportunities to explore new combinations of existing approaches and algorithms in the context of a system-building tool. To do so required developing a flexible approach in the generation of the interconnect itself and choosing an appropriate abstraction for the user to specify their design. This section covers the motivation and design of these aspects of GENIE.

3.1.1 Interconnect Generation Approach

Our approach to interconnect synthesis is to build bottom-up from small modules that perform as few functions as possible. We begin by asserting that the ultimate purpose of interconnect is to realize the transmission of data between a set of sources and sinks. In general, this requires some storage and processing in addition to the physical transmission of electrical signals over wires. A non-exhaustive list of such interconnect functionality includes:

- **Distribution:** Selectively directing data to one (or a subset) of many possible destinations, thus implementing one-to-many communication.
- **Arbitration:** Selecting one input from a set of many inputs, implementing many-to-one communication.
- **Storage:** Holding data in-place in order to temporarily match a discontinuity between the presence of data at a source, and the ability to consume data at a sink.
- **Delay Matching:** Delaying data by a specific, fixed number of clock cycles in order to align the arrival of two or more streams of data at a common sink.
- **Pipelining:** Breaking long chains of combinational interconnect logic to achieve higher performance.
- **Clock Crossing:** Passing data between two unrelated clock domains.

- **Conversion:** Performing arbitrary combinational logic to transform interconnected-related control signals, such as addressing information, between different representations.

One possible way to design interconnect building blocks is to create large, monolithic modules that perform several of these functions in a tightly-coupled manner. This is the essence of a traditional Network-on-Chip router, as seen for example in the CONNECT [58] framework for FPGAs. These routers have, in general, many input ports and output ports, meaning they implement both arbitration and distribution, and also contain input and/or output buffers. However, this tight coupling specializes, and thus limits, the kinds of applications that the network can be used for. For example, the number of internal pipeline stages of a router sets a lower bound on end-to-end latency, adding unnecessary overhead to systems that do not require the full functionality and complexity offered. The design philosophy behind GENIE is to generalize, rather than specialize, in terms of supported application design styles, so this approach was not chosen.

One of the earlier attempts to "crack open" the monolithic FPGA router design came in the form of the Split/Merge NoC [41, 32]. Here, two different kinds of interconnect modules were proposed: split and merge nodes. Split nodes are dedicated to distribution, and merge nodes to arbitration. Each has a variable number of possible inputs or outputs, and nodes can be chained together in different ways to create arbitrary network topologies. The original work proposed buffers at the input of every split and merge node, resulting in some coupling of functionality and baseline primitive complexity, but it was an improvement (in terms of clock frequency and area) over CONNECT, and continued the trend of moving away from ASIC-style monolithic router designs.

GENIE's interconnect design takes the next logical step and decouples the functionality within building blocks even further. We propose six different kinds of interconnect modules, each performing as few interconnect-related functions as possible, which will be presented in greater detail in Section 3.3. With GENIE's approach, each module has low design complexity and functionality, and richer functionality is obtained by chaining different modules together. This approach has two key benefits:

- The ability to productively automate the creation of interconnect for target applications of smaller size than would be possible with existing approaches, and thus incurring less performance and area overhead.

- Providing new opportunities for application-specific optimizations, given by the large space of combinations of small primitives. A similar approach has been used to create optimized chains of floating-point operators for FPGAs [46].

3.1.2 Input Specification Design

The choice of specification abstraction is crucial to the ability to easily express design intent, as well as providing a tool enough knowledge about the application to enable it to perform application-specific optimizations.

A system-building tool's design abstraction is structural and is essentially a higher-level version of the structural subset of HDLs like Verilog or VHDL. HDLs use ports (input, output, and bidirectional) that are used as the endpoints for nets. All HDL ports and nets are of the same "type": they are an abstraction for zero-cycle transmission of one bit of data, lacking any additional communication-related usage information. This semantically-neutral communication abstraction makes HDLs a good lowest

common denominator for hardware design, but necessarily limits an FPGA synthesis tool's ability to provide meaningful design automation and optimization capabilities.

Interfaces and logical links are the system-building tool analogies to HDL ports and nets. They serve as endpoints and as specifiers of the existence and direction of transmissions, respectively. These transmissions have richer and more specific semantic meaning than that of the HDL abstraction. A common use case is that of memory-mapped communication, owing to the system-on-chip-centric focus of existing commercial system-building tools, in which a microprocessor is the central actor. The two fundamental operations are reads and writes, using a memory address to indicate a destination. This leads to two types of interfaces: masters, which initiate requests, and slaves, which respond to requests. By specifying logical links between multiple masters and multiple slaves, a tool is instructed to generate all the necessary arbitration and address decoding interconnect, offering a great deal of automation and simplicity to the user – as long as their intent was to create a system with memory-mapped peripherals.

We desire GENIE's communication protocol to be more general and not tied to a specific and narrow use case such as memory-mapping. Similar to our strategy for interconnect building blocks, complex and specialized protocols should be implementable by building upon the simple one that is offered. At the same time, we wish to do this to an extent that maintains a certain level of design automation (otherwise, HDLs already trivially satisfy the first goal). To satisfy both goals, we have chosen the following features to be included in our communication abstraction to the user:

- **Multicast Addressing:** Addressing is necessary for any communication abstraction in which transmissions are sent to different destinations under the runtime control of user logic. Multicast addressing gives a more general capability to send to more than one destination (or all possible destinations) at the same time. Broadcasting is trivially available in the HDL net abstraction and it is limiting to prevent a higher-level communication protocol from offering it as well.

- **Flow control:** The ability to specify valid data only on some clock cycles, and ability for a sink to indicate the availability to receive data. Even if user endpoints do not require flow control for a particular application, these signals are fundamentally necessary to be generated and/or consumed *by the interconnect* for many-to-one communications. Any time two transmissions can contend for the same output, one must be stalled or slowed somehow.

- **Packetization:** Allowing a transmission to span multiple consecutive clock cycles. In theory, support for this could entirely be left up to implementation by the user as a generic data signal on top of the protocol that is provided. After all, multi-cycle transmissions are just many one-cycle transmissions. However, without explicit packetization added to the interconnect protocol, multiple simultaneous transmissions from different sources may arrive in an interleaved manner at a sink that is incapable of maintaining separate buffering and state for reconstructing each transmission. Explicit packetization allows for a policy of preventing this scenario by forcing arbiters, within the interconnect, to wait for an entire transmission to pass through before granting an output to another waiting input.

Given the above requirements, we have developed our own protocol that strives for the required balance of generality and automation capability. The flow control and packetization aspects lead to the existence of signals that resemble low level general streaming protocols such as AXI-Stream [8] and Avalon-ST [34]. With the addition of signals for (multicast) addressing, different transmissions can be

routed to different destinations in the network. We call our resulting protocol *Routed Streaming* (RS) and provide a more descriptive overview in Section 3.2.3.

However, in addition to RS interfaces, auxiliary/utility interface types are also needed for more mundane tasks such as delivering clock and reset signals. Regular single-bit HDL nets would suffice for these (and is, in fact, how they are ultimately implemented in GENIE), but there is still semantic value to identifying a clock signal as a clock signal, as it allows the association of a clock domain to a data-carrying RS interface. Another important and separate type of interface is a "conduit" that acts as a catch-all for any kind of communication that is unsupported by the tool's abstractions. It is a multi-bit data bus that is translated verbatim into HDL nets. A common use of conduits is to interface with elements external to the FPGA such as I/O, peripherals, and off-chip memory controllers.

Our choices for designing the RS protocol aim to be general, but do not cover all possible communication features. Notably missing is support for virtual channels, which is an extension to flow control that the interconnect would explicitly have to be designed to understand to realize its potential as a deadlock-breaker. Using more physical channels is a way of getting around this, as was noted in the work on the Split/Merge NoC[32], which included the argument that FPGAs have enough wires to make this practical.

3.2 Input Specification

Having provided some context for design choices, this section proceeds to provide a more top-down description of GENIE's input specification.

3.2.1 Functional Modules

The purpose of a system integration tool is to instantiate and connect instances of functional modules together. The "functional" refers to the fact that these are user-created hardware modules that perform application-specific functions related to computation or storage, in contrast with interconnect modules that are later automatically instantiated by the tool and are used to facilitate communication. Examples of functional modules include processors, hardware accelerators, peripherals, and on-chip memory blocks – exactly the same kinds of building blocks that existing network-on-chip and FPGA system integration tools were designed to connect together. In addition to these classic examples, GENIE aims to enable the practicality of connecting together finer-grained modules, such as individual state machines and small sections of datapaths.

A functional module definition in GENIE is a thin wrapper around an actual user-provided Verilog `module`. In addition to the associated Verilog module's name, a functional module definition has its own name (which can be different), and a set of interface definitions, which are described in the next section.

3.2.2 Interfaces

An interface consists of one or more HDL signals, each with a prescribed role. The available signal roles, and the requirements for the presence or absence of each role, is dependent on the type of interface. Altogether, GENIE defines four interface types: Routed Streaming for general communication, and three others for utility purposes, as previously described in Section 3.1.2. Each of the four interface types exists

in a "source" and "sink" variant and can exist either as part of a functional module, or as a top-level interface of a system. The available interface types are:

- **Clock:** Contains a single signal of role 'clock' that specifies a clock source or clock sink.
- **Reset:** Contains a single signal of role 'reset' that specifies a reset source or reset sink.
- **Conduit:** Contains one or more arbitrary-width signals that GENIE will realize as wires and otherwise leave untouched for interconnect synthesis. Each signal can have a role of 'in', 'out', or 'inout' that are only used to specify an absolute direction for the signal, and say nothing of its purpose. Multiple signals of the same role must be distinguished by an additional user-assigned "tag" parameter which is an arbitrary, but unique, text string. GENIE will connect signals with matching tags when connecting conduit interfaces.
- **Routed Streaming:** This is the main interface type that GENIE uses to specify module-to-module communications, using the Routed Streaming protocol which is described in the next section. RS interfaces have an associated clock interface, which implicitly specifies a clock domain.

Additional metadata can be defined for some interface types. One example already mentioned is that of RS interfaces having a reference to an associated Clock interface. Additional interface properties will be discussed in detail when describing relevant parts of the interconnect synthesis flow.

3.2.3 Routed Streaming Protocol and Interfaces

One of the design goals of GENIE was to allow a wider range of communication types than existing system integration tools. Existing popular interface protocols such as AXI [9] and Avalon [34] offer a designer two choices (via two flavors of each protocol): express desired communication as memory-mapped reads and writes and receive a high amount of automated interconnect synthesis, or express communication as low-level point-to-point streaming links and receive no such benefits. GENIE's Routed Streaming protocol seeks to bridge that gap. It allows the user to specify unidirectional communication of zero or more named streams of data of arbitrary bit width, with *optional* flow control, addressing, and variable packet length. The presence or absence of the signal roles within an RS interface, associated with each of those functions, dictates the needs of the communication link, and is used by GENIE to generate interconnect that is only as complex as necessary. The protocol is header-less, using separate signals rather than including control information in-line with the data, which takes advantage of the abundance of wiring on FPGAs. A packet/transmission is at least one flit long (one clock cycle in the transmitting clock domain), and its length is dynamically variable via an optional "end of packet" control signal. The following signal roles are available for RS interfaces:

- **Data:** The data being transmitted, of arbitrary bit width. Multiple data signals are allowed within an RS interface, distinguished using a unique user-specified tag, similar to Conduit interfaces. Data signals from a remote interface, sharing the same tag, will be connected by GENIE. Conversely, it is possible to have zero data signals in an interface, for interrupt signal-like communication using the other flow control signals alone.
- **Valid:** Flow control signal, of width 1, specifying valid data this cycle. It is optional, and its absence is assumed to mean that valid data is being transmitted every cycle.

- **Ready:** Flow control signal, of width 1, that is an input to transmitting interfaces and an output in receiving interfaces. Indicates that data is ready to be transmitted this cycle. If absent on a receiving interface, it is assumed the interface is always ready to receive valid data and will not stall incoming transmissions.
- **Address:** Used by transmitting interfaces to select one or more destinations for a transmission, and by receiving interfaces to determine the source of incoming transmissions. Either use case is optional. Addresses are integer IDs that are associated with a logical link that connects two RS interfaces, rather than with an interface itself. By emitting an address, a source interface selects one or more links that share that address, and thus implicitly select one or more remote sink interfaces to send data to. Similarly, a sink interface receives an address, identifying an RS links, and thus implicitly the source of the transmission.
- **EOP:** End-of-Packet signal that is asserted on the last cycle of a transmitted packet. It is optional, and if absent, specifies that each packet is of length 1. It is similar to the EOP signal in the Avalon-MM and Avalon-ST protocols.

RS interfaces are a superset of traditional streaming interfaces used by Avalon and AXI. To implement the use case of memory-mapped communication, a pair of RS interfaces can be used at each end – one for sending read requests and write requests+data, and one for receiving read reply data. GENIE's address scheme cannot be used directly as a memory address, and an extra automation/convenience layer to make this possible can be considered as future work.

3.2.4 System Definitions

A system is a canvas on which functional modules are instantiated and logical links are specified. One or more system definitions serve as the input to GENIE, and the fully-realized HDL implementations of each system are emitted as the output. Each system has a unique name that also serves as the name of the HDL module GENIE will generate for the system. Within a system definition, each functional module instance is also given a unique name. The interfaces of the instantiated modules can then be referred to hierarchically by specifying the name of the instance and the name of the enclosed interface. This is used when specifying the endpoints of logical links.

In addition to connecting the interfaces of functional modules to each other, it is necessary to give the system connectivity to the outside world. To facilitate this, the user is able to add interface definitions to the system itself, as if it were a functional module. The HDL signal names specified in these interface definitions name future Verilog signals that *will be created* as top-level inputs and outputs when the HDL module for the system is generated by GENIE. Within the system, logical links may now also be defined between these top-level interfaces and the interfaces of modules instantiated within the system.

Because GENIE treats a system definition identically to a functional module definition, hierarchical design is possible – systems may be instantiated within other systems.

3.2.5 Logical Links

Logical links connect a source interface to a sink interface of the same type, and they are part of a system's definition. For the three utility-class interface types (clock, reset, and conduit), a logical link simply creates (one or more) HDL nets in the correct direction. Between RS interfaces, a logical link

specifies the existence of a transmission from the source to the sink at some unspecified time during the application's lifecycle.

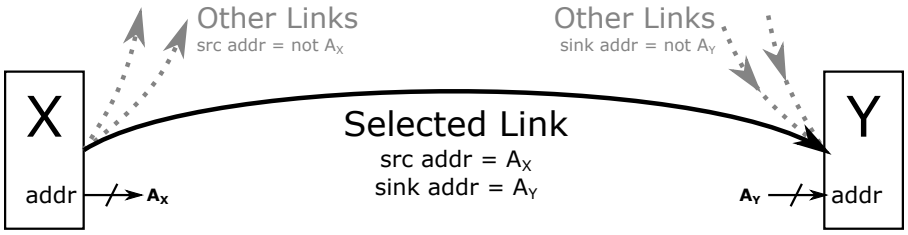

Figure 3.2: Selection of RS links by RS interfaces

Links between RS interfaces have optional *source address* and *sink address* properties that form the basis for GENIE's addressing scheme. Figure 3.2 illustrates an example, in which a source RS interface X and a sink RS interface Y exist, and have many outgoing and incoming links defined. In particular, there exists a link between X and Y, which the user annotated with a source address of A_X and a sink address of A_Y. GENIE interprets this specification to mean: when the address signal belonging to interface X drives the value A_X, the link between X and Y is selected and a transmission between X and Y occurs. When this transmission occurs, if sink interface Y has an address signal (which is optional, and would be an input) then the interconnect will drive it with the value A_Y. Because addresses are annotated on links rather than directly on sources or sinks, this allows a particular sink interface to be known by many different source addresses, each potentially different for every connected source interface (and vice-versa). With this addressing system, source interfaces are able to select a destination, or multiple simultaneous destinations, for a transmission by driving an address value that matches the source address parameter annotated on the links that terminate at the desired destination(s). Similarly, it allows a sink to be informed about which source is sending an incoming transmission, corresponding to a link's sink address parameter. To our knowledge, this is a new functionality not present in existing on-chip communication protocols.

A trivial use case for this information would be for a module to discern the source of a request and direct the reply back to the requester. However, this behavior is no different than memory mapped read semantics, where it is expected that a read request from a master will generate a response back to the source of the request. A more interesting use case would be for the receiving module to perform completely different behavior depending on the source, for example, to prioritize requests from certain sources by directing them to a different internal queue, or to bypass a cache for a transmission from one source but not from another. Again, since addresses are properties of links, different addresses need not even correspond to different physical sources/destinations – it is entirely possible to have one source, one sink, and many logical links between them, differing only in source and/or sink addresses. These logical links may or may not end up sharing the same physical interconnect. The address information in this case effectively acts as an extra data field at the receiver, which could be used to express priority or mode of a request. In another configuration, there can be one source, multiple sinks, and links from the source to the sink having non-unique source addresses. This enables multicast transmission, as when an address is emitted, all matching links are used.

Our addressing scheme is also unique in that it is entirely optional. In the absence of any source addresses annotated on a source interface's outgoing links, broadcast occurs. If the source interface lacks an "address" signal role, then all outgoing links are assumed to be selected when a transmission occurs.

This functionality can recreate simple point-to-point communications which does not require addresses at all, and replicating the semantics of a low-level HDL net, adding to the generality of GENIE's input abstraction.

Other per-link properties exist to inform GENIE of the nature of the application transmissions carried across the links. RS links can have optional *packet size* and *importance* properties that describe the size and desired bandwidth of transmissions. Similar to traditional timing constraints, GENIE can use these specifications to optimize the generated interconnect in an application-specific manner. These will be elaborated in detail when explaining the relevant parts of the GENIE flow.

3.2.6 Lua Specification Example

In this section we provide a small design example that demonstrates GENIE's Lua-based specification front-end, incorporating the elements described above. The system being created is called "TestSys" and is shown in Figure 3.3. It contains four functional modules. A dispatch unit generates a stream of 16-bit numbers and alternatively send them to *either one* of two consumers: an inverter module that complements each 16-bit value, and a reverser module that bit-reverses each received value. These are serialized into a single stream of values that are received by a final "xorer" module that keeps a running XOR of all values received, which is the final output of the system. Each module (including the system itself) has clock, reset, and RS interfaces which are shown and named in the figure.

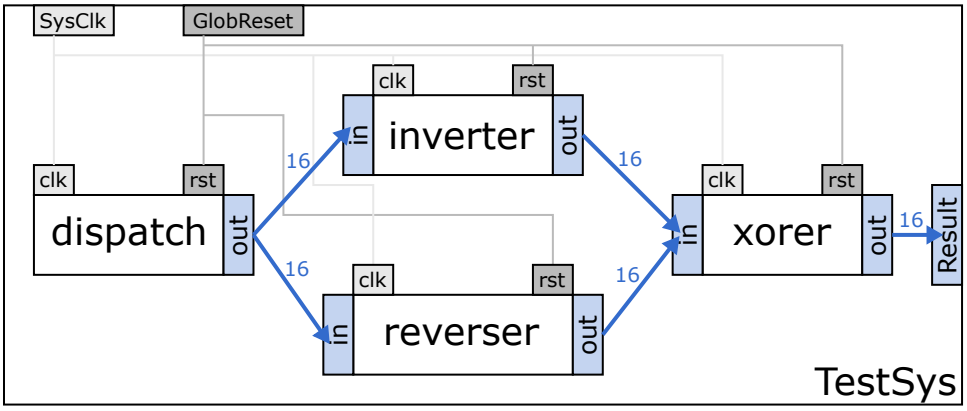

Figure 3.3: Example GENIE System

We will now walk through the Lua code that, when provided to the GENIE tool, will generate this system. First, GENIE's 'builder' package is imported, and a Builder object is created and arbitrarily named ''b''. This provides the user an object-oriented interface to write the remainder of the specification, using Lua's `:` operator to perform method calls on `b` to define modules, systems, links, and so forth.

```
1 -- Create a GENIE builder object
2 require 'builder'
3 local b = genie.Builder.new()
```

This object is a stateful wrapper around a more advanced underlying GENIE API that would normally require a more explicit style of coding. For example, to define a new RS interface, one would have to specify the name of a previously-defined module for which the new interface would belong to. In contrast, by using the Builder wrapper, the target module is implicitly assumed to be the last module that was

defined. By using a Builder object, the imperative set of method calls used to define the hierarchical structure of GENIE objects instead resembles a declarative (for example, XML-based) representation, reducing redundancy and simplifying the syntax for the user.

Next, we define the functional modules and their interfaces, which will later be instantiated within a system. The first module is the dispatcher. The `component` method begins a module definition given its name. By default, this will also be the name of the associated Verilog module. Each of its three interfaces is created and named. The clock and reset interfaces, named 'clk' and 'rst', are by default associated with Verilog ports on the module of the same names. The RS interface, named 'out' (which is associated with the clock interface upon definition) contains many signals of varying roles, which are added on lines 8–11.

```
4  b:component('dispatch')
5    b:clock_sink('clk')
6    b:reset_sink('rst')
7    b:rs_src('out', 'clk')
8      b:signal('valid', 'o_valid')
9      b:signal('ready', 'i_ready')
10     b:signal('data', 'o_data', 'WIDTH')
11     b:signal('address', 'o_addr', 1)
```

Note that the indentation is purely cosmetic and is meant to signify hierarchy. As previously mentioned, during each method call to the Builder object, the target of the operation is implicitly known to be the last applicable parent object – each `signal` is implicitly attached to the last-created interface, which in this case is the RS source called "out". The presence of valid and ready signals (which are optional) indicates full support of flow control and backpressure by this interface. The data signal contains the payload, and the address signal allows the interface to select one of two destinations. The first argument to each `signal` method is the type/role of signal, the second is the name of the Verilog signal, and the third is the size in bits for signal roles that support configurable bit widths. The data signal on line 10 has parameterizable width, and will be set to 16 upon instantiation later. The address signal has width 1 since it needs to select two destinations only.

Lines 12–45 define the other three functional modules and their interfaces.

```
12 b:component('inverter')
13   b:clock_sink('clk')
14   b:reset_sink('reset')
15   b:rs_sink('in', 'clk')
16     b:signal('valid', 'i_valid')
17     b:signal('ready', 'o_ready')
18     b:signal('data', 'i_data', 'WIDTH')
19   b:rs_src('out', 'clk')
20     b:signal('valid', 'o_valid')
21     b:signal('ready', 'i_ready')
22     b:signal('data', 'o_data', 'WIDTH')
23
24 b:component('xorer')
25   b:clock_sink('clk')
26   b:reset_sink('reset')
27   b:rs_sink('in', 'clk')
28     b:signal('valid', 'i_valid')
29     b:signal('ready', 'o_ready')
30     b:signal('data', 'i_data', 'WIDTH')
```

```
31      b:signal('address', 'i_lp', 1)
32    b:rs_src('out', 'clk')
33      b:signal('valid', 'o_valid')
34      b:signal('ready', 'i_ready')
35      b:signal('data', 'o_data', 'WIDTH')
36
37  b:component('reverser')
38    b:clock_sink('clk')
39    b:reset_sink('reset')
40    b:rs_sink('in', 'clk')
41      b:signal('valid', 'i_valid')
42      b:signal('ready', 'o_ready')
43      b:signal('data', 'i_data', 'WIDTH')
44    b:rs_src('out', 'clk')
45      b:signal('valid', 'o_valid')
46      b:signal('ready', 'i_ready')
47      b:signal('data', 'o_data', 'WIDTH')
```

Next, the system itself is defined. Its clock and reset interfaces are created the same as with functional modules, on lines 49 and 50. Lines 51–58 instantiate the functional modules and provide names to the instances. This is where the `WIDTH` parameter is given a concrete value of 16 bits for each instance.

```
48  b:system('TestSys')
49    b:clock_sink('SysClk')
50    b:reset_sink('GlobReset')
51    b:instance('dispatch', 'the_dispatch')
52      b:int_param('WIDTH', '16')
53    b:instance('inverter', 'the_inverter')
54      b:int_param('WIDTH', '16')
55    b:instance('reverser', 'the_reverser')
56      b:int_param('WIDTH', '16')
57    b:instance('xorer', 'xorro')
58      b:int_param('WIDTH', '16')
```

The clock and reset connections are defined next. Top-level clock and reset interfaces, named SysClk and GlobReset, are connected to each of the 4 modules' clock and reset inputs. To automate this process, a loop is used. Interfaces are identified with hierarchical dot notation, and are built programmatically using the Lua string concatenation operator (`..`). For example, the first invocation of line 61 will define a clock link from 'SysClk' to 'the_dispatch.clk'.

```
59    for dest in Set.mkvalues{'the_dispatch', 'the_inverter',
60                             'the_reverser', 'xorro'} do
61      b:clock_link('SysClk', dest .. '.' .. 'clk')
62      b:reset_link('GlobReset', dest .. '.' .. 'reset')
63    end
```

`Set.mkvalues()` is part of a utility package included with GENIE, not part of the base Lua language or libraries, that assists in manipulating sets of objects.

The RS link definitions follow, using the `rs_link` method. This takes two additional optional arguments in addition to the source and sink interface path: the source address and the sink address. This binds the logical RS link to the values emitted or consumed by the interfaces' `address` signals. For example, line 65 specifies that the logical link from the dispatcher's 'out' interface to the reverser's 'in'

interface has a source address of '1', meaning that this transmission will occur when the dispatcher emits the value '1' out of its address signal.

The fourth argument, used on lines 66 and 67, is a sink address. This associates the logical link with an address seen at the *sink*. When the xorro module receives a transmission from the inverter module, the xorro module's address signal will see a value of '0'. A Lua `nil` for the 3rd argument leaves the source address of each link unspecified, since the inverter and reverser can only send to one destination anyway. In general, the source address and sink address associated with each RS link can differ.

```
64  b:rs_link('the_dispatch.out', 'the_inverter.in', 0)
65  b:rs_link('the_dispatch.out', 'the_reverser.in', 1)
66  b:rs_link('the_inverter.out', 'xorro.in', nil, 0)
67  b:rs_link('the_reverser.out', 'xorro.in', nil, 1)
68  b:export ('xorro.out', 'Result')
```

Finally, line 68 introduces a convenient method of both creating a top-level system interface and connecting it to something. The export method creates a top-level interface called Result of the same type, and compatible direction, as the existing xorro module's 'out' interface – which is a Routed Streaming source. This short-hand both creates the Result interface, the appropriate signal definitions (which match those of the exported interface), and creates an RS link. Additional examples of GENIE Lua specification code are available in Appendix A.

3.3 Interconnect Building Blocks

This section describes the nature of the physical interconnect that GENIE creates to implement the user's input specification. It is a preview of Chapter 4 which will explore the circuit design in more detail. Here, we wish to only introduce the building blocks themselves, of which there are six, and the motivation for which was previously laid down in Section 3.1.

3.3.1 Routing Primitives

The task of routing data within a network is a combined effort of distribution and arbitration between an array of sources and an array of sinks. We refer to the modules that perform these tasks, jointly, as *routing primitives*. In GENIE, distribution and arbitration are performed by **split** and **merge** nodes, respectively. Unlike the UPenn/Caltech split and merge nodes, ours are *purely combinational* and contain no pipelining or buffering in their data paths. We keep the advantages offered by having separate split and merge nodes: they can be arranged in arbitrary patterns to implement different network topologies.

Split nodes have a single input and a parameterizable number of outputs. The data payload between the input and outputs is passed through unmodified, and the only logic within a split node is related to flow control: given a one-hot bit vector of size N, a 1-to-N split node will present the data as valid to a specific subset of its output ports. This allows the network to be generalized to support multicast transmissions.

Merge nodes have a parameterizable number of input ports, and one output. The data path is simply a multiplexer. It is controlled by a round-robin arbiter that is capable of selecting a different input every cycle. A stripped-down variant of the merge node exists that eliminates the complex arbiter given the knowledge that none of the expected transmissions through the merge node will temporally overlap. This will be described in more detail in Chapter 4.

3.3.2 Elastic Buffer

The **elastic buffer** primitive is a FIFO with a depth of two words and a latency of one cycle, built out of two registers and a multiplexer. A detailed block diagram is available in the next chapter, presented as Figure 4.5. Depending on the required usage context, it can provide storage, delay matching, and/or pipelining. When no backpressure is present on its connected links, most of the elastic buffer is optimized away by the FPGA toolchain leaving just a register. When backpressure is present, it acts like a FIFO, but from a timing perspective, its latency of one cycle makes it appear equivalent to a single register. Throughout this document, we use the terms "elastic buffer" and "register" interchangeably in the context of interconnect that GENIE generates, since this single primitive can act as both.

3.3.3 Delay Buffer

A **delay buffer** primitive is an inelastic buffer that delays data for a *fixed* number of clock cycles. Its behavior is equivalent to a chain of cascaded registers, but it is designed to instead use the FPGA's distributed RAM when the width and depth are large enough to yield a higher density than the equivalent chained register implementation. Delay buffer primitives are only used for delay matching purposes in applications that use fixed-delay pipelined functional blocks.

3.3.4 Clock Domain Converter

The GENIE **clock domain converter** joins two parts of the network that are synchronized to different clock domains. It is implemented using a FIFO backed by FPGA distributed memory, so it performs storage/buffering as a side effect.

3.3.5 Converter

A GENIE **converter module** is the sixth and final primitive. It is a parameterizable lookup table, which is a generalization of the underlying FPGA's fixed-size lookup tables. It is used for translating between the address signals emitted and consumed by the user's functional modules, and address spaces internal/private to the interconnect that are automatically generated by the GENIE flow. As an example, split nodes expect a one-hot bit vector specifying the intended destination, while a user module uses a sequential fully-encoded integer address. One or more converter modules will be inserted into the interconnect to match these representations.

3.3.6 Inter-Primitive Links

User functional modules, and GENIE-inserted interconnect modules, are woven together using standard FPGA wiring. To aim for generality, GENIE does not prescribe a specific interconnect data width, nor a specific multiplicative width alignment (eg. multiples of bytes or powers of two). Separate wires are used to carry data payload and interconnect control signals, in parallel. Previous work, such as the CMU Split/Merge network [32] used a similar approach, as opposed to multiplexing control information with data using packet headers. This design choice of having separate wires for control signals offers the simplest hardware implementation, avoiding the need for serialization/deserialization logic that would also increase the lowest available latency by two cycles. If we wish for our interconnect synthesis tool to encroach on design scenarios that were previously only available with hand-coded RTL, we must

support a zero-latency combinational paradigm that matches the semantics of a bus of raw RTL nets. There certainly are advantages to packetizing the total data over multiple clock cycles using a narrower link width in scenarios where the extra complexity and latency are tolerable. This is the domain of traditional NoC architectures, and we leave GENIE's support of such types of links to future work.

3.4 The GENIE Flow

This section provides a high-level overview of the entire GENIE flow – the operations that take the user's specifications at one end and produce SystemVerilog output at the other. Individual parts of the flow will be elaborated on in detail in later chapters, together with results that illustrate the efficacy of the various contributions made in this work.

The bulk of the GENIE flow operates on the Routed Streaming (RS) logical links present in the user's input specification. Clock, reset, and conduit links are straightforwardly realized as Verilog nets at the end of the flow – their logical and physical representations are essentially equivalent. The realization of RS links, however, involves the potential insertion of interconnect primitives to realize routing, conversion, and synchronization tasks that are specified or implied by the the logical connectivity of the RS links, their RS interface endpoints, and user-provided communication metadata.

Systems are the objects of largest operational granularity in the GENIE flow – each system is processed independently, and in an order that respects the user's design hierarchy. Within each system, the network formed by RS interfaces and RS logical links is first partitioned into *domains*. A domain is a connected component of the RS network, which itself can be visualized as a directed bipartite graph with RS source interfaces on one side and RS sink interfaces on the other. Figure 3.4 illustrates an example system with three domains. The squares represent RS source or sink interfaces, which although not shown, are either embedded within functional module instances or are top-level interfaces of the system itself.

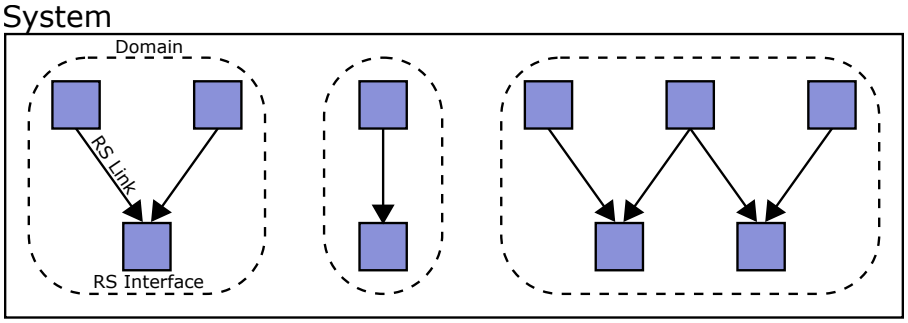

Figure 3.4: A GENIE system whose RS logical links comprise three domains

Domains are automatically extracted by GENIE from the input specification, and are processed independently. There exist two types of domains, which affects the complexity of the downstream processing:

- **Auto Topology:** GENIE will automatically generate and optimize a physical network topology for the logical links in the domain, which determines the arrangement and connectivity of interconnect routing primitives. This is the default.

- **Manual Topology:** Here, the user explicitly instantiates and connects routing primitives as part of the input specification. GENIE will use the specified topology instead of creating its own. A domain is automatically marked as 'manual' if any of its RS interfaces are explicitly connected by the user in this manner.

Domains, by construction, are disconnected from each other at the level of logical links. Generally, this remains true in terms of physical interconnect as well. As will be elaborated on in Chapter 9, the physical topologies generated for auto topology domains remain on separate networks, allowing each to be processed and optimized independently, reducing the size of each problem. However, for manually-specified domains, there is no restriction preventing the user from creating a physical topology that spans multiple logical domains, allowing their logical links to share physical interconnect resources.

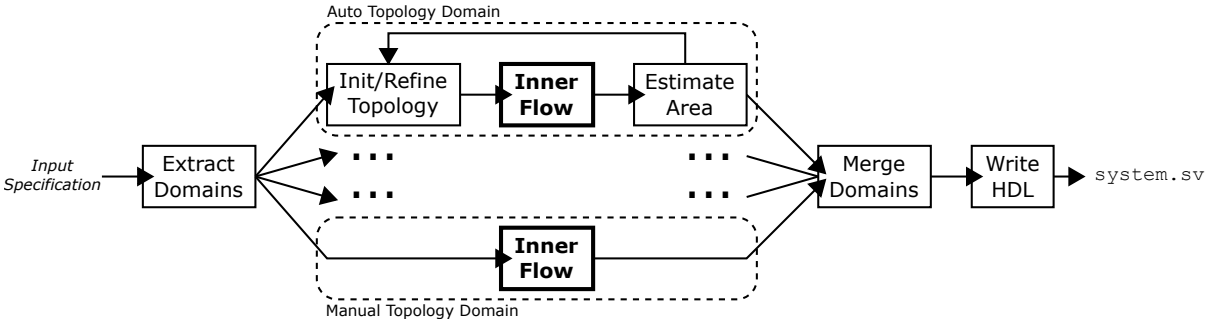

Figure 3.5: GENIE Outer Flow

Figure 3.5 provides an overview of the per-system flow, referred to as the "Outer Flow", that begins at the input specification and ends at generated SystemVerilog. The processing paths for the two different kinds of domains (auto and manual topology) are shown. The automatic topology flow is an iterative optimization loop that will be described in detail in Chapter 9 and its goal is to minimize area while respecting user performance constraints. Common to both domain types is the "Inner Flow" which, given a network topology, builds a detailed representation of a domain's interconnect and itself contains several different optimization problems that we address in this work. The Outer Flow concludes by merging all separately-processed domains and writing the SystemVerilog representation of the entire system (functional modules and interconnect) to a file. By this point, the external interfaces of the generated system have been finalized, and the Outer Flow can repeat for other systems that hierarchically instantiate the original system as a functional module.

The GENIE Inner Flow, shown in Figure 3.6 takes as input a partially-realized domain that has a defined topology, meaning the input already contains a specific arrangement and instantiation of split and merge nodes. The function of the Inner Flow is to insert all the other necessary interconnect primitives to create a functioning fragment of the network. We now give an overview of the different tasks performed by the Inner Flow.

3.4.1 Low-level Topology Refinement

This is a post-processing step performed on the topology given as input to the Inner Flow, and occurs for both manually-specified and automatically-generated topology domains. Here, merge nodes with a large number of inputs are restructured into equivalent trees of smaller-input merge nodes. The maximum

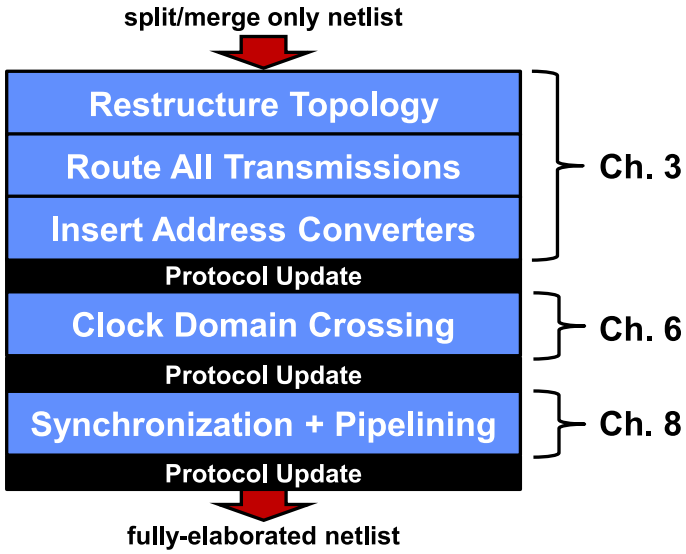

Figure 3.6: GENIE Inner Flow

number of inputs of a merge node in the restructured tree is dependent on the FPGA technology. For example, on 6-input LUT architectures [6, 70], merge nodes of four inputs or less can be implemented using a single level of LUTs in the datapath (which is a multiplexer). Both the original large merge node, and the restructured tree, are still combinational at this point in the flow. The key goal of restructuring is to provide more possible locations into which registers can be inserted later in the flow, versus a single large merge node.

3.4.2 Routing

After a domain's topology is finalized for the present invocation of the Inner Flow, the next step is to route each logical link as a transmission through the interconnect that has been generated so far – which consists of just split nodes, merge nodes, and placeholder physical links between them. Presently, GENIE statically routes transmissions using Dijkstra's shortest path algorithm on a graph with the RS interfaces of user modules, split nodes, and merges nodes being the vertices and the placeholder links being edges, all of equal cost/distance.

Future extensions to GENIE could add an API to specify manual routing to complement the existing ability to manually specifying the topology. Enhancements to the automatic routing could also be made, such that transmissions are routed based on the existing occupancy of physical links by other transmissions and thus avoiding areas of known traffic congestion.

3.4.3 Addressing

After a domain's logical RS links have been statically routed through its network of split and merge nodes, those routing decisions are realized in an Addressing phase. Here, an internally-visible address is assigned to each logical link and "converter" primitives are inserted into the interconnect to and from this representation.

Addressing and routing are related due to the role of split nodes in GENIE. Split nodes are the only locations where a transmission can enter and then leave out of multiple possible output ports, hence

steering the transmission in the correct predetermined direction, or directions. This is decided based on a bit vector, called a *split mask* that is an input signal into the split node and is supplied with the incoming transmission. Each bit in a split mask corresponds to one output of the split node, indicating that the input should be replicated to that output. This in theory allows GENIE (albeit in future versions) to support any style user-visible address representation – memory-mapped, X/Y mesh, and so forth, as long as that representation is convertible into a split mask for each split node.

Implementation of a larger range of addressing protocols on top of the split mask interface is left as future work. In the current version of GENIE, the user-facing RS protocol implements only one of many possible addressing schemes as an initial proof-of-concept of this overall design philosophy. As previously described in Section 3.2.5, each RS link has up to two possible addresses associated with it: a source address and a sink address. The address signal at an RS source or sink interface is matched up with a logical link in order to implicitly select a destination, or identify a source. Effectively, from the point of view of an individual RS sink (source), all remote sources (sinks) have a well-defined address. However, from the point of view of the domain as a whole, conflicts could arise between the local points of view of individual sources or sinks. For example, the address "2" could be used by source X to identify sink Y, but source W could use "2" to identify sink Z. The interconnect would have no hope of routing traffic unambiguously to either sink.

To resolve these possible ambiguities, the Addressing stage of the Inner Flow needs to generate an internal *domain address mapping* for the domain. An address mapping here is defined as an association between an address and one or more logical links. This results in a total of three different types of address mappings within a domain:

- **User mappings:** The mappings local to each RS source or sink interface. They need not globally agree with each other. This mapping is entirely determined by the user.
- **Split masks:** The bit vectors entering each split node can also be considered address mappings, as they associate a numeric value with one (or a group) of transmissions that share a particular output port on the split node. These mappings are local to each split node, and depend on the domain's topology and routing.
- **Domain mapping:** An intermediate address mapping that is designed such that it is unambiguously convertible between all present user mappings and all split masks. There is one per domain, and it is generated by the Addressing stage of the Inner Flow.

Each address mapping is encoded by an address signal of the appropriate width, and more than one such mapping/signal can be present at any point within the interconnect. After determining the domain mappings, GENIE inserts converter primitives at boundaries between different mappings.

Figure 3.7 shows an example domain with four RS interfaces (sources A/B and sinks C/D) connected with a shared bus topology consisting of one merge node and one split node. There are six total address mappings, each represented by a different colour. This is the most general case – if sinks C and D did not care about the sources of their transmissions (as would be decided by the user), they would not have or need user address mappings, bringing the total down to four. Converter primitives are inserted near each RS source or sink to convert each user address mapping to/from the domain mapping (in purple). There is a fifth converter inserted immediately before the split node which generates a split mask (green) from the domain mapping. While the split mask is consumed at the split node, the domain mapping must continue to pass through the split node in order to reach the converters located before C and D.

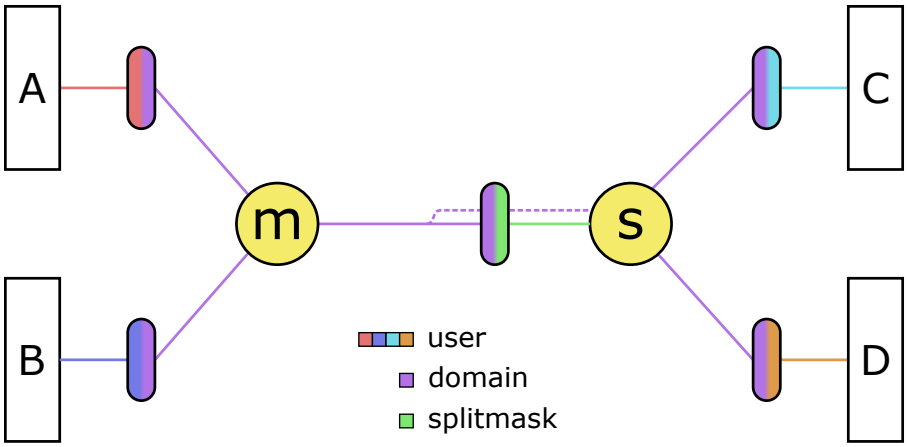

Figure 3.7: Address mappings and locations of converter primitives (rounded rectangles)

Logical Link	User-Defined				GENIE-Generated	
	A	B	C	D	domain	splitmask
A→C	2		8		0	2'b10
A→D	3			5	1	2'b01
B→C		16	9		2	2'b10
B→CD		17	10	6	3	2'b11

Figure 3.8: Example address encodings for each logical link

Figure 3.8 provides example encodings for each address representation shown in Figure 3.7. The six color-coded address representations appear as columns in the table. Here we assume four transmissions (logical RS links): A to C, A to D, B to C, and a multicast transmission from B to CD. Each is initiated by one of the sending interfaces (A or B) by asserting that interface's encoding of the desired transmission, which corresponds to the "source address" of the RS link during link definition. For example, if A wishes to send to C, it asserts address "2", and if B wishes to send to (only) C, it asserts address "16". The sink encodings of each transmission are observed at receiving interfaces C and D. The unicast transmission from B will appear as address "9" at C, and the multicast transmission from B will appear as "10" at C and "6" at D.

The user-defined encodings are fixed and are part of the input specification. GENIE must determine an appropriate internal address representation (in general, there could be many co-existing simultaneously in different parts of the system, although we currently do not explore this possibility). At each split node, the correct split mask must be generated to allow transmissions to be routed correctly. Since the shown split node has two outputs, each corresponds to one bit of the mask.

Many opportunities exist for implementing optimizations for reducing the required number of converter primitives. For example, if sinks C and D did not require receiving sink addresses, the internal domain mapping could simply be merged with the split mask. Of course, this may not be possible if there existed more than one split node, which fed off the same address representation. Rotation of split node outputs could be used to coerce favourable splitmask encodings. If A and B's user mappings happened to agree on their labellings for transmissions to C and D, the two converters adjacent to A and B could be replaced with a single converter *after* the merge node. Furthermore, if the user decided

to implement one-hot source address encodings directly, neither the split mask nor the internal domain mapping would be required. Clearly, there is a rich optimization problem here, and we leave exploration of this space to future work. For our initial solution to this problem, we create a domain mapping with unique integer IDs for each transmission, and a splitmask conversion occurs preceding each split node. Some optimization may occur during combinational logic synthesis in the downstream FPGA toolchain.

3.4.4 Protocol Carriage and Width Determination

GENIE views the connections that it creates between interconnect primitives as "physical links", in contrast to the logical RS links that the user specifies as input. A single physical link is really an abstraction for a bundle of separate control and/or data signals. Different physical links will carry a different subset of possible signal types. Each signal is ultimately produced and consumed by an interconnect primitive or user functional module. In order for a signal to be delivered to its point of consumption, it must be carried by all upstream physical links. This requires that the physical links be sized appropriately, and that intervening interconnect primitives be parameterized correctly to accommodate the cumulative width of all signals passing through them.

The Protocol Carriage step of the Inner Flow determines which physical links must carry which signals, based on the locations of producers and consumers of each signal type in the interconnect. In Figure 3.6, it appears several times as "Protocol Update", since incremental updates may be necessary after any other flow step that inserts or modifies interconnect primitives.

For the links and primitives that simply pass a signal through without producing or consuming it, a consistent packing of signals into a single bit vector is performed. This is not trivial – a single set of wires can be used to carry different signal types at different times, as a result of upstream multiplexing performed by merge nodes. After this stage is performed, an important result is that each physical link now has an exact known size in bits. This allows algorithms present in the proceeding stages of the Inner Flow to make optimization decisions based on link width.

3.4.5 Clock Domain Crossing

GENIE supports designs that have multiple clock domains. During the Inner Flow, clock domain converter primitives are inserted into the interconnect at boundaries between different user clock domains. Establishing these boundaries is an optimization problem in which the goal is to minimize the total area cost of clock domain crossing hardware. It arises from the fact that while the associated clock domains of the user's RS interfaces are fixed and known, the clock domains of the intervening interconnect primitives are free variables. Each possible assignment of clock domains to interconnect primitives can yield a different number of clock domain crossing points, with each such point potentially having a different link width in bits and therefore a different cost.

This optimization problem has been studied in the context of application-specific NoCs [44], and is an instance of a *multiterminal cut problem* on a directed graph, which is NP-hard in the general case [25]. GENIE uses an approximate heuristic approach that assigns one clock domain at a time. Further details on the algorithm will be presented in Section 6.2.3.

3.4.6 Register Insertion

Up to this point in the GENIE Inner Flow, all interconnect primitives that have been inserted (except for clock domain crossing units) are purely combinational in the parts of their data paths that forward signals from an input to an output. The path from any user functional module's source RS interface, to a sink RS interface, incurs 0 cycles of latency. Following GENIE's design philosophy of composition of elemental interconnect functions, the addition of registers is accomplished separately and optionally. Inserting registers (actually elastic buffer primitives, as discussed in our interchangeable use of the terms in Section 3.3.2) serves two distinct purposes: synchronization, and interconnect pipelining.

Synchronization is related to the functional correctness of the user application, and has nothing to do with performance. One of the design styles that GENIE aims to generalize and provide automation capability to is the set of designs that eschew the use of backpressure-based flow control and instead contain pipelined functional modules that have known and fixed processing latency from inputs to outputs. In such designs, removal of flow control makes the functional modules (and the resulting interconnect) cheaper in terms of area and complexity. The trade-off is that pipeline delays must now be balanced across different functional modules, such that multiple branching paths through the interconnect re-converge with identical *total* latencies. To support this, GENIE allows a designer to annotate functional modules with known fixed latencies, specified from each RS sink interface (module input) to RS source interface (module output). Another specification extension, a set of *synchronization constraints*, allows the designer to specify the desired latency relationships between chains of RS logical links that span multiple hops of intervening fixed-latency functional blocks. GENIE then inserts the correct (and minimal total) number of registers into the interconnect to realize the constraints. This is done by transforming the constraints and specifications into an integer linear program (ILP), much the same as in an HLS scheduling problem [31, 15, 11].

The other purpose of inserting registers into the interconnect is to pipeline it and reduce the maximum total combinational logic delay. To do this, GENIE creates a timing graph of the interconnect generated thus far, before any register insertion. This graph contains delays measured in units of LUT delays, specific to an FPGA device family. These are stored in a database containing characterized area and timing information for each of GENIE's interconnect primitive types. It is populated during an offline characterization phase that synthesizes each primitive, in many parameterizations, through a full back-end FPGA CAD flow. Based on the timing graph for the domain's interconnect, an optimization problem is formulated that aims to determine the *locations* where elastic buffers should be inserted such that no buffer-to-buffer combinational delay exceeds a specified threshold. A trivial solution is to pipeline every possible link, so the optimization goal is set to minimize the total number of elastic buffer stages, weighted by each link's total bit width.

Both of the register insertion sub-problems (pipelining and synchronization) generate ILP constraints that are solved and optimized *simultaneously*. The result of the solution provides the numbers, and locations, of the required elastic buffers. Details are found in Chapter 8.

3.5 Summary

This chapter provided a high-level overview of the GENIE tool, including a rationale for our overall interconnect synthesis methodology, the design of the user input specification, and our choice of primitive module building blocks. In the next chapter, we describe these building blocks in much more detail. A

complete Lua design example was also given to demonstrate how the user interacts with the tool. The brief tour of the inner and outer synthesis flows in Section 3.4 is a preview of upcoming chapters that focus on specific interconnect transformations and optimizations.

Chapter 4

Interconnect Microarchitecture

This chapter elaborates on the quick overview of GENIE's six types of interconnect primitives previously described in Section 3.3. Here, we discuss the design of each primitive in detail at the RTL level. Each has many possible parameterizations and optional capabilities, in order to provide the generality and complexity needed in some user designs, and simplicity and low overhead for designs that do not. A change of parameters affects a primitive's area and timing, which must be modeled and predicted to enable interconnect optimization. We begin with a discussion of this modeling infrastructure, as it is common to all primitives.

4.1 Area and Timing Modeling

In order to make tradeoffs during various decisions, any optimization process needs a reasonably accurate way to measure the impact of each decision. To enable this within GENIE, we store the resource utilization and combinational logic depth of each primitive, under different parameterizations, in a database. There are two types of parameters: those that directly correspond to Verilog module parameters in the primitive's source code (for example, a data bus width), and those that indicate the presence or absence of certain input/output control signals. That is, when an input port is tied to a constant, or an output port is left unconnected, this causes the FPGA synthesis tool to optimize away logic within the primitive, yielding changes in area and timing that should be captured. One common example is the presence or lack of a 'ready' signal used for backpressure, as not all user designs need this functionality.

For each primitive, a specific setting of parameters yields a data point in a multi-dimensional space. This data point contains area and timing values that are originally obtained by synthesizing each primitive for a specific FPGA architecture using the back-end CAD software and extracting from the compilation report in an automated manner. To avoid sampling the entire parameter space, which would be costly in terms of time and storage, the LUT-based implementation of each primitive is studied to identify opportunities for parameter decoupling and using interpolation or extrapolation. These will be explored in the sections pertaining to each primitive.

4.1.1 Area

Area values are used primarily by the topology optimization loop in GENIE's Outer Flow. The area of an entire domain (after passing through the Inner Flow) is estimated and compared to that of other

topology choices. The area information at each data point in a primitive's multi-dimensional parameter space contains three values: the number of combinational LUTs, the number of registers, and the number of embedded RAM cells (no GENIE primitives use block RAM resources).

The effect of some types of parameters on area usage is linear and predictable. For example, varying the width of a data path usually incurs a replication of existing resources, with a fixed marginal per-bit cost. Since we place few restrictions on the user's data width, this is advantageous, as it would be impractical to store each possible width parameterization for every primitive. Instead, we can maintain database entries for widths of 1 and 2 bits (while keeping all other parameters constant), and extrapolate the area usage to any desired bit width beyond 2 bits.

4.1.2 Timing

Timing values are used for automatic interconnect pipelining in GENIE's Inner Flow, as briefly described in Section 3.4.6. There, registers are inserted *between* modules to limit the longest chain of combinational LUT stages seen in the overall system. The timing information at each data point in a primitive's multi-dimensional parameter space is a table of variable size in which each row stores one point-to-point delay. A row contains three entries: source name, destination name, and the logic depth in number of FPGA LUTs from that source to that destination. The source and destination are names of input and output ports (respectively) of the primitive module. Either one (but not both) can also be a special name called "INT" which refers to an internal unspecified register within the primitive. This allows the timing modeling to capture three types of delays: a specific input port to a specific output port, a specific input port to any internal register, any internal register to a specific output port.

The combinational delays are stored as a number of LUT stages as these vary less across FPGA compile runs than absolute nanosecond values and are independent of device speed grade. It would be a straightforward modification to store absolute delays instead, as the algorithm that consumes these timing values (covered in Section 8.2), just sees them as integer cost values.

4.2 Merge Node

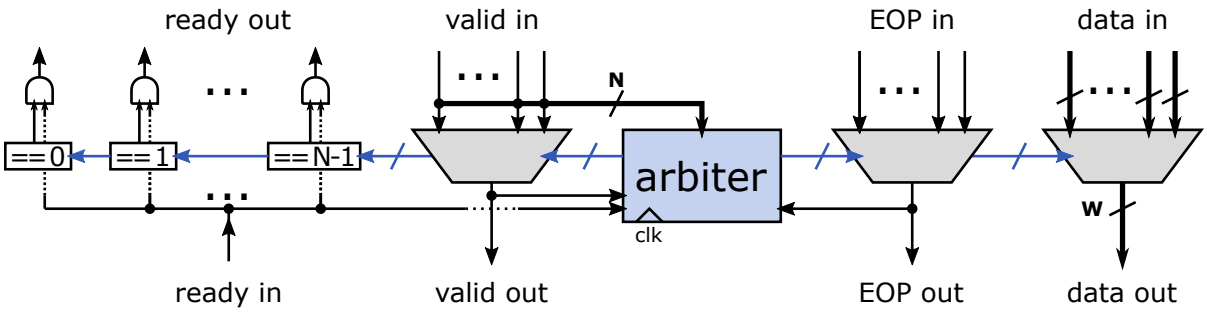

Figure 4.1: Full-featured merge node design

A merge node forwards one of its N inputs to its output, stalling the rest. Each input is a bundle that contains these types of signals:

- Data, of W bits wide

- Valid
- End-of-Packet (EOP)
- Ready (travels in reverse direction)

When multiple inputs are valid, the merge node will select the current input to be serviced based on a round-robin arbitration policy. Once an input is selected, it remains selected until it raises its EOP signal. As mentioned in Section 3.1.2, this is sufficient to ensure that multi-cycle transmissions arrive at their final destinations without being interleaved with parts of other simultaneous transmissions bound for the same destination, removing the need for user modules to maintain extra buffering and state. Future work may loosen and generalize this policy to enable more aggressive traffic balancing within the interconnect.

Figure 4.1 illustrates the merge node architecture when all signals are present. The data path that implements the actual selection and forwarding of the input to the output is just an N-to-1 multiplexer, which is the simplest possible FPGA-based implementation for this task. The duty of the merge node's control logic (the arbiter) is to provide the multiplexer select signal, which is coloured blue in the figure. This is a function of which inputs are currently requesting access (the input valid signals), the arbiter's internal state, and the output-side EOP and ready signals. The incoming ready signal is back-propagated only to the currently-selected input (zeroing the others), which has the effect of stalling the non-selected inputs.

4.2.1 Arbiter

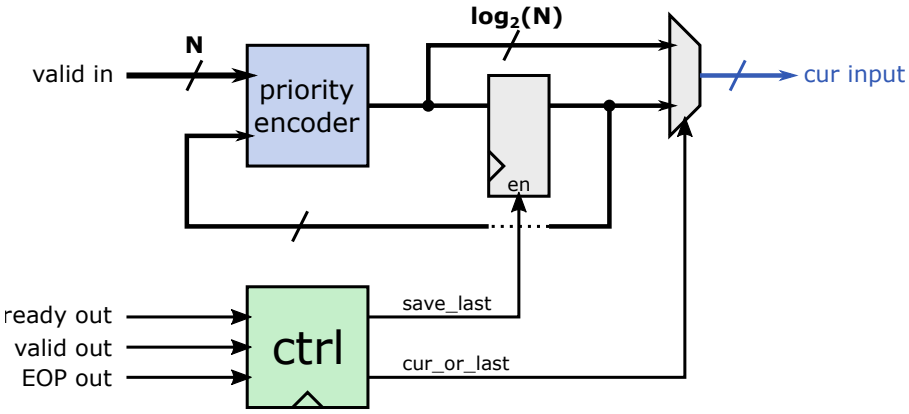

Figure 4.2: Round-robin arbiter design

The arbiter block within Figure 4.1 is shown in detail in Figure 4.2. It uses a priority encoder to identify the first of the N inputs that is high, yielding an index $log_2(N)$ bits wide. To avoid a lower-indexed input from monopolizing and starving the others, the priority encoder also takes the index of the last-serviced input, which it uses as a starting point to begin scanning the valid signals. It does so in a circular manner, wrapping around through index 0.

The last-serviced input, held in the register, is updated at the end of the initial clock cycle of every transmission. On subsequent clock cycles, the registered value also serves as a means to lock the output of the arbiter at the same fixed value until the transmission ends. A state machine controls this behaviour

as well as the updating of the register. It monitors the merge node's sink-side ready, valid, and EOP signals, the congruence of which determines when data is actually sent and received, and additionally when the end of the transmission is reached. Its two states represent being in the initial cycle of the transmission, and the state of being in the subsequent cycles.

If a particular domain does not use end-of-packet signals or backpressure, GENIE will tie these signals within the domain's interconnect blocks (and specifically, its merge nodes) to constant values. This will cause the FPGA synthesis flow to optimize out parts of both the merge node and the arbiter.

4.2.2 Parameters and Area Usage

Merge nodes have the following parameters:

- **N:** The number of inputs.
- **W:** The width of the data port.
- **BP:** Whether the `ready` signal is used or not.
- **EOP:** Whether the `eop` signal is used or not.

The area and timing database contains parameterizations of N from 2 up to a fixed number. Merge nodes with more inputs than this hard maximum are restructured into trees during the Inner Flow. When N becomes large enough to require implementing the priority encoder and multiplexers using multiple levels of LUTs, the area consumption of the merge node becomes hard to predict, hence the need to empirically record each parameterization of N. However, parameterizations of W only need exist for values of 1 and 2, with extrapolation used to estimate other values. This is because W only affects the mux for the `data` signal, with each extra data bit having the same marginal area cost.

For example area usage, a parameterization of $N, W, BP, EOP = \{4, 2, 1, 1\}$ consumes 12 6-LUTs[1] and 3 registers, with a marginal cost of 1 LUT per extra W. The largest combinational delay is 4 LUT stages, from the `valid` input to internal registers.

4.2.3 Conflict-Free Merge Node

If the user can guarantee that certain transmissions will *never* overlap in time, and some subset of these transmissions happen to be routed by GENIE through a merge node, then a large optimization can be made to its design. Specifically, since only one input can be valid at a time, there is no need for an arbiter – each valid signal can be ANDed with its corresponding data signal and the result ORed to generate the output data. The resulting simplified merge node design is shown in Figure 4.3.

Note that the EOP signal is no longer consumed by the merge node, so it and other such signals are combined into what is now called the 'data'. Also, while the merge node itself no longer produces stalling (via a lowered ready signal), it can still *propagate* downstream ready signals upstream. This backpressure functionality costs no area, and combined with the irrelevance of the EOP signal leaves only the N and W parameters for this version of the merge node. The area usage for $N, W = \{4, 2\}$ is reduced to seven 6-LUTs and no registers, with the maximum combinational delay reduced to two LUT stages.

[1] Arria 10 device

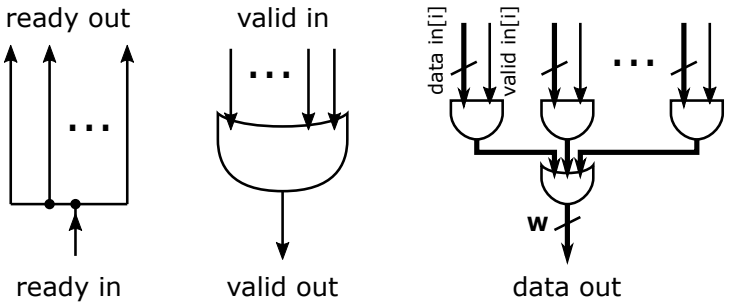

Figure 4.3: Conflict-free merge node design

4.3 Split Node

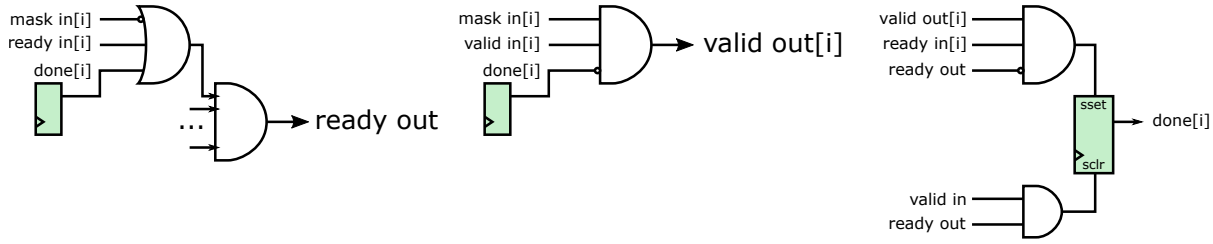

Figure 4.4: Split node design with multicast

The split node has one input and N outputs, and its function is to replicate its input to one or more of its outputs. The input contains data of W bits wide that is forwarded unmodified to each output. All of the complexity of the split node is in gating and manipulating the valid and ready control signals that belong to the input as well as each of the N outputs. An N-bit-wide *split mask* signal accompanies the input, each bit of which selects an output to forward the input data to. The split node's logic is trivial if it is only required to support unicast operation, in which at most one output is enabled at a time. The logic expressions for each downstream valid signal, and the upstream ready signal, become:

$$valid\,out_i = valid\,in\ \&\ mask_i$$
$$ready\,out = (ready\,in_0 \mid !mask_0)\ \&\ (ready\,in_1 \mid !mask_1)\ \&\ \cdots\ \&\ (ready\,in_{N-1} \mid !mask_{N-1})$$

However, this is not sufficient for multicast, since not all selected outputs may necessarily be ready to accept the data simultaneously. Without modification, the result would be incorrect behavior in which outputs that *are* ready will continue to receive duplicate copies of data until every output signals that it is ready. One option is to withhold all the output valid signals until every selected output is ready. However, this will result in deadlock with downstream merge nodes, which need incoming valid signals to be present in order to generate upstream ready signals.

The solution used is shown in Figure 4.4. Here, a *done* register exists for every output port. It is set to 1 if its associated output port has successfully received the data *and* other selected outputs have not yet done so. It is used to gate each output's valid signal to avoid the duplicate transmission of data on subsequent cycles. It is also taken into account when generating the upstream ready signal – outputs marked as 'done' do not contribute to the generation of backpressure. Thus, the output ready signal

goes high only when every selected output has received the data, in either the current or a previous clock cycle. When this occurs (signaled by valid in and ready out being high), all the *done* registers belonging to each output are cleared in preparation for the next transmission. The described behaviour is shown as synchronous clear and preset signals on each register in the figure for clarity, and in reality is implemented using D and enable signals.

4.3.1 Parameters and Area Usage

Split nodes have the following parameters:

- **N:** The number of outputs.
- **BP:** Whether the `ready` signal is used or not.
- **NO_MULTICAST:** Whether to disable support for multicast transmissions.

The width of the data port, W is also a parameter to the actual HDL module, but since output data is just replicated from input data, it has no effect on the area or timing of the split node.

The Inner Flow and area/timing database support split nodes up to 32 outputs, after which they are broken up into trees of smaller nodes. A full-featured 16-output split node with of N, BP, $NO\_MULTICAST$ $= \{16, 1, 0\}$ consumes 43 LUTs and 16 registers. Removing the ready signals ($BP = 0$) reduces the usage to 23 LUTs and no registers. Further removing support for multicast yields 16 LUTs (one per output) and no registers, and this is the same case as with multicast but no backpressure (removal of ready signals optimizes away the same circuitry).

As future work, it may be possible to model the area and performance of the split node at an even finer level of detail, taking into account downstream traffic conditions per-output. If it is known that a particular output will never experience a stall, then its "ready in" signal will be tied high and not contribute to the combinational logic driving the associated "done" register or the "ready out" signal.

4.4 Elastic Buffer

In literature [14], elastic buffers are FIFOs of arbitrary finite capacity used to implement *elastic circuits*, where local decentralized flow control through Ready and Valid signals allows a series of functional modules to synchronize the production and consumption of data, thus enabling a latency-insensitive design style.

GENIE's elastic buffer primitive is a FIFO of depth 2 implemented using registers. It can be used to perform pipelining, delay matching, and buffering, depending on the usage context. The input side and output side each have three signals: data of parameterizable width W, valid, and ready. Data and valid travel forwards from input side to output side, and ready travels backwards. The ready signal is optional and can be omitted, which reduces the entire elastic buffer primitive to two sets of registers: one for the data (containing W registers) and one register for the valid signal.

Figure 4.5 shows the design of the fully-functional elastic buffer. There are two pairs of registers, each holding a copy of the incoming valid and data signals: `data0`/`valid0` and `data1`/`valid1`. In the normal mode operation (when the incoming ready is 1) the incoming data and valid signals flow through just the `data0`/`valid0` registers and to the output, and the `data1`/`valid1` contain a copy of the same valid/data

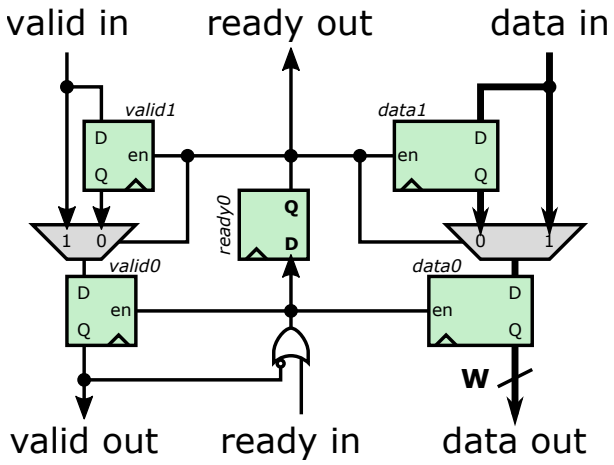

Figure 4.5: Elastic buffer design.

but are unused. However, once the output side becomes stalled, `data0`/`valid0` must hold their value, while `data1`/`valid1` continue to accept one more token. Figure 4.6 illustrates this behaviour.

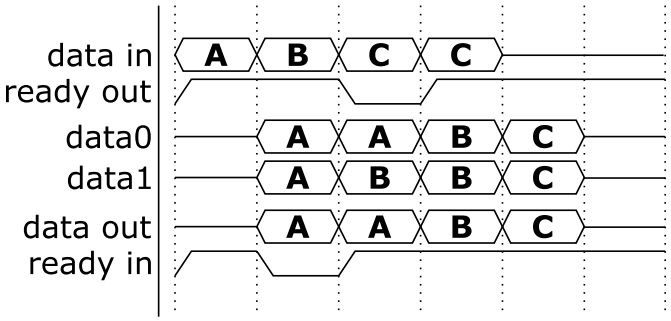

Figure 4.6: Elastic buffer timing diagram for stall handling. Valid signals are high only when the associated data signals are shown to have a value.

After the stall has cleared, `data0`/`valid0` receive the held token from `data1`/`valid1` before resuming normal operation. The design accomplishes two things: it pipelines both the forward valid/data signals as well as the backward-traveling ready signal, and does so without the loss of data during a stall. For Intel/Altera architectures, the actual design of the elastic buffer avoids using combinational logic to implement the 2-to-1 multiplexer in the figure. Instead, the extra `SLOAD` and `SDATA` register control signals are used. This optimization, shown in Figure 4.7, is important when the link width W is large, as each bit of data would normally consume combinational LUTs for the multiplexing.

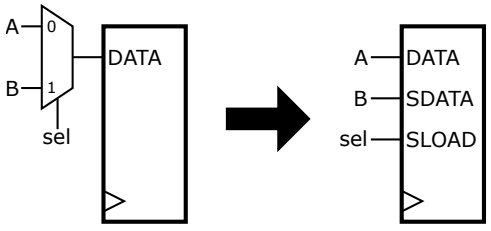

Figure 4.7: Using register control signals to implement a 2-to-1 multiplexer

4.4.1 Parameters and Area Usage

The elastic buffer's only parameter is the data width W. The area/timing database only needs to store entries for a baseline value of W since the resource usage for arbitrary values of W can be easily extrapolated. When using the ready signal (in full elastic buffer mode), each additional data bit consumes 2 registers (one within `data0` and one within `data1`). When not using the ready signal, only 1 extra register is consumed per data bit, and in this mode, the entire elastic buffer consists only of the `valid0` and `data0` registers and no combinational logic. The delay through the elastic buffer is 1 clock cycle, and it incurs no combinational delays beyond the OR gate, which is a single 2-input LUT unless optimized away during logic synthesis by merging it with upstream/downstream logic.

4.5 Clock Crosser

A clock crosser primitive is a dual-clock FIFO. The flow control signals (valid/ready) at both the input end and the output end are used to derive the 'read', 'write', 'empty', and 'full' FIFO control signals. Currently, the FIFO is instantiated from the FPGA vendor's library of primitives, as it is well-optimized for the target technology. The only constraint that we impose on the FIFO is to prefer the use of distributed RAM rather than block RAM, for timing purposes – distributed RAM is spatially abundant and is thus easier to place closer to upstream and downstream interconnect. The trade-off of using distributed RAM is lower capacity, but since our need is simply to provide clock-crossing ability, any depth is acceptable.

4.5.1 Parameters and Area Usage

The only parameter is the data width W. It affects the number of parallel distributed RAM blocks that must be used. On Altera architectures, the distributed RAM is a maximum of 20 bits wide [33], so one extra block is used for every 20 bits of W. Additional logic and registers are used to maintain FIFO read and write pointers, convert between binary and Gray encoding, and send the pointers from one clock domain into the other safely. These are fixed costs independent of W.

4.5.2 Future Work

Using a dual-clock FIFO is the most general solution that is guaranteed to provide correct behaviour but may not necessarily be optimized for certain applications. If, for example, the two clocks are synchronized and related in frequency by an integer (or rational) multiple, knowledge of this relationship could be used to use just logic and registers and avoid the need for a memory-based FIFO. This opportunity would present itself in applications that use double-pumped clocking schemes for parts of their design. Different circuits would be used if the source was the faster or the slower of the two clock domains, to avoid losing data during the worst case of continuous usage.

Additionally, if it was known that transmissions occur infrequently (and never reach continuous usage), a low-cost clock crossing solution using a full handshake could be used, even if the clock domains were unrelated. All of these optimization opportunities have yet to be explored in GENIE's interconnect synthesis flow.

4.6 Address Converter

The Address Converter is a generalized sparse lookup table of arbitrary size that maps input values of W_{IN} bits wide to output values of W_{OUT} bits wide. They are inserted during the Inner Flow to convert between different address representations (as previously discussed in Section 3.4.3). A converter has N total input-to-output mappings, and not every possible input value needs to have a mapping (it is allowable for $N \leq 2^{W_{IN}}$). When an unrecognized input value is presented, undefined behaviour occurs. This is caught with an assertion during RTL simulation of the generated SystemVerilog, and is intended to catch erroneous behavior by user modules that emit undefined addresses.

Aside from the table geometry parameters already mentioned (N, W_{IN}, and W_{OUT}), the contents of the table itself are also provided as vector-type parameters to each Address Converter module instance. The design synthesizes to pure combinational logic. In addition to being dependent on the table size, the nature of the generated LUTs will be dependent on the table contents as well. Thus, it is difficult to estimate the number of LUTs used, and the combinational logic depth, of the synthesized implementation without ourselves replicating the FPGA logic synthesis and optimization algorithms. Currently, we estimate the LUT usage with a worst-case bound: $log_k(W_{IN}) \cdot W_{OUT}$, where k is the FPGA LUT size. This often is an overestimation, as in addition to finding a more compact representation, logic synthesis finds ways to merge the converter logic with upstream or downstream logic as well.

4.7 Delay Buffer

Delay Buffer primitives are a drop-in replacement for long chains of consecutive Elastic Buffers that were inserted during the register insertion phase of the Inner Flow. When such chains of registers are inserted, it is not for pipelining, but for synchronizing and matching delays with fixed-latency functional blocks. Rather than using registers, the Delay Buffer is implemented using RAM, preferably of the distributed, rather than block, variety. The replacement of register chains with Delay Buffers is intended as an area-saving optimization, as a single Intel LAB block configured as distributed RAM can hold 32 20-bit words or 64 10-bit words in the same area that normally would contain a single 20-bit word if using registers.

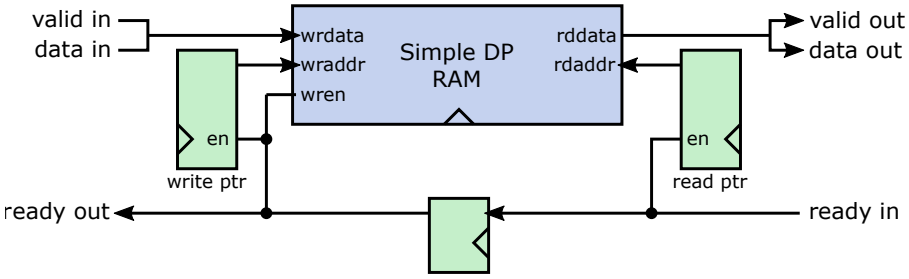

Figure 4.8: Delay buffer design

The Delay Buffer takes incoming {data, valid} signals as input and delays them by N cycles as the output. Figure 4.8 illustrates the design. It uses read and write pointers to index a RAM, like a FIFO. However, unlike a traditional FIFO, the pointers are always incremented in tandem and are a fixed (the maximum) distance part. The effect is to delay both valid *and* non-valid (empty) inputs, without internal compaction of non-valid entries. This reflects the primitive's intended as a delay element for

fixed-latency systems. Nevertheless, a ready signal is supported for edge cases where the segment of physical interconnect occupied by a Delay Buffer is used for both fixed-latency and latency-sensitive communications. In this case, the ready signal is pipelined, and is used to disable the incrementing of the read/write pointers, stalling the entire RAM delay chain contents in lockstep.

This can be seen in the fact that the valid signals are combined with the data signals and not treated specially for control purposes.

4.7.1 Parameters and Area Usage

The Delay Chain's parameters are the data width W and the number of cycles to delay, N. The presence or absence of the ready signal is also considered when looking up area usage and timing in the primitive database. Without the ready signal, the read and write pointers remain, and increment forever at a fixed distance apart.

Including the Clock Crosser, the Delay Buffer is the only other GENIE primitive that uses RAM resources. Unlike the Clock Crosser, the Delay Buffer's RAM usage is dependent on two parameters, N *and* W, and both affect the usage in a stepwise manner. On Intel devices, distributed RAM blocks contain 640 bits arranged as either 32 words of 20 bits of 64 words of 10 bits. Multiple such blocks will need to be chained together in parallel to cover the necessary width (W), while any extra depth (N) is handled by additional downstream Delay Buffer instances with separate read/write pointers.

The total resources consumed by a Delay Buffer include the necessary number of distributed RAM blocks along with the pointer management logic and registers. When deciding to replace a chain of Elastic Buffers with one or more Delay Buffers, we weigh the cost of both options to decide whether the replacement is fruitful or not. However, in corner cases where the product of width and depth is sufficiently small, it is entirely possible that the back-end CAD software will choose to implement the distributed RAM block as registers anyway. We found that this occurs, on the Arria 10 FPGA, when the width-depth product is less than 32. Several entries in the primitive database exist for such parameterizations (sparsely, in increasing powers of two for width and depth), and the nearest largest entry is chosen during lookup. Such area lookups will show zero memory usage, and only logic and registers, as reported by Quartus synthesis results when the database was built. This will help GENIE avoid inserting Delay Buffers in these corner cases, in a somewhat data-driven way.

Chapter 5

Example Applications

In this chapter, we introduce two real applications that will be used as the basis for the experiments performed in Chapters 6–9 to measure and validate our research contributions. We choose to concentrate our efforts into investigating two fully realized applications in great depth and detail rather than using a broader set of benchmarks and only lightly discussing their inner workings. Examining the detailed design problems that arise within these applications is required to fully appreciate our contributions. The designs of the applications will be revisited in later chapters as each relevant design problem and contribution is examined.

5.1 LU Decomposition

Our first application is an LU decomposition engine, originally based on a design by Zhang et al. [73] and used in our previously published work on GENIE [61, 62, 63]. Its function is to receive a square matrix A as input and decompose it into a lower-triangular matrix L and upper-triangular matrix U such that $L \times U = A$. The matrices are stored in off-chip DDR SDRAM memory in IEEE 754 floating-point format and are operated on by a configurable number N of Compute Elements (CEs). The off-chip memory is accessed through a configurable number M of independent memory controllers. Coordination of the entire system is performed by a control unit. Figure 5.1 illustrates the top-level design of the LU decomposition engine, which includes these components. This application, and the way we choose to implement it, has many features that yield interesting and diverse interconnect-related design problems:

- Multiple independent external memory interfaces.
- Two independent clock domains.
- Productive use of broadcast and multicast communication.
- Combining fixed-latency and latency-insensitive design styles.
- Network topology exploration and optimization.
- Fine-granularity system and interconnect design.

The last point, related to fine-granularity design, is a key contribution of our work, and refers to the design of each Compute Element. In addition to constructing the system shown in Figure 5.1 (the

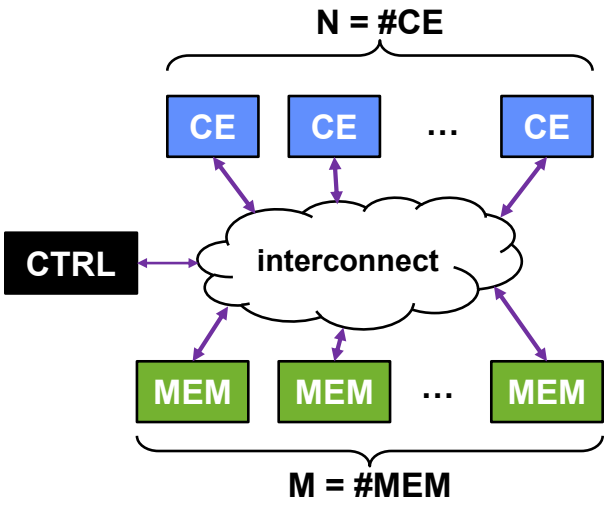

Figure 5.1: Top-level block diagram of the LU decomposition engine

so-called *coarse-grained* context), we are able to automate the construction of the smaller system within each CE – a context in which existing system and interconnect building tools are not well-suited for, as we will show in Chapter 6. Furthermore, a more detailed description of the connectivity between the components of Figure 5.1 will show that the topology of the network implementing the interconnect can be customized to take advantage of the application's communication patterns. Chapter 9 will discuss another key contribution in which these topologies are automatically generated and optimized for area, based on a user-provided description of the underlying communication patterns. Some of these details will be covered later in this chapter, but first we will describe the functionality of the LU decomposition engine in order to motivate the interconnect design problems.

5.1.1 Operation

The input matrix A is divided into blocks of 64×64 elements, such that there are a total of $B \times B$ blocks. Blocking improves the spatial locality of data access and reduces off-chip memory bandwidth [16]. Matrix A is processed in-place by the algorithm such that the output matrices L and U end up occupying the same space originally occupied by A. The outer loop of the algorithm iterates over square sub-matrices of A of progressively smaller size, with the first sub-matrix being A itself, with bounds $(0, 0)$ to $(B-1, B-1)$ in units of 64×64 blocks. Subsequent sub-matrices have bounds $(1, 1)$ to $(B-1, B-1)$, $(2, 2)$ to $(B-1, B-1)$ and so forth, with the top-left corner of the current sub-matrix indexed by k in general.

Algorithm 1 describes this outer k loop in more detail. For each sub-matrix indexed by k, there is one serial round of computation followed by multiple parallel rounds of computation. A round delegates the processing of matrix columns to individual CEs. The serial round processes the first column of the sub-matrix, and the parallel rounds process the remaining columns, with each parallel round processing up to N columns at a time. This partitioning arises due to the data dependencies between blocks: processing each block (x, y) within a sub-matrix requires a *left* block at (k, y) as input, which must be processed first. Therefore, we process the first column at $x = k$ serially, and then use the full parallelism afforded by the hardware to process all remaining columns in parallel. The processing of the first column requires extra steps and extra hardware, and we decided to give only CE0 that hardware, which is why

it is always tasked to do the serial round.

ALGORITHM 1: Outer loop of blocked LU decomposition algorithm

```
// Process every sub-matrix
for k = 0 to B − 1 do
    // First serial round
    Dispatch just CE0 to process column k from y = k to y = B − 1
    // Subsequent parallel rounds
    x = k + 1
    while x < B do
        Dispatch all N CEs to process columns x through x + N − 1 from y = k to y = B − 1
        x = x + N
    end
end
```

Figure 5.2 illustrates several aspects of the algorithm on a matrix with dimension $B = 7$ blocks, or 448×448 individual elements. The hardware is parameterized to contain $N = 4$ CEs and $M = 2$ memory controllers. Subfigure 5.2a) shows the first three sub-matrices corresponding to the first three values of k for the outer loop. Subfigure 5.2b) delineates the three rounds of computation that occur for outer loop iteration $k = 1$, along with which CEs are assigned to which columns. The first serial round, round 0, is always assigned to CE0. The next round, round 1, is the first parallel round, and can use all 4 available CEs to process the next 4 columns. The final round, round 2, is also a parallel round, but since only 1 column remains, it is processed by a single CE.

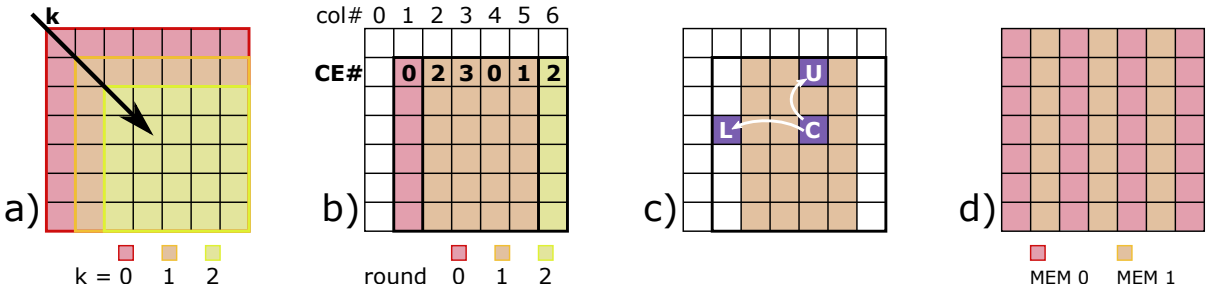

Figure 5.2: **a)** Outer k loop passes 0, 1, and 2. **b)** Computation rounds 0, 1, and 2 and respective CE-to-column assignments for pass $k = 1$ with $N = 4$ CEs. **c)** Locations of L and U blocks for a C block at $(4, 3)$. **d)** Matrix column to memory controller mappings for $M = 2$

In the parallel rounds, a column x is assigned to the CE with index x **mod** N. As shown in Subfigure 5.2d), each column x is also held by memory controller x **mod** M. This striped column-to-memory mapping evenly divides the memory bandwidth between memory controllers when performing parallel computation rounds. Together with the column-to-CE mapping, this also limits the set of memory controllers that any CE (except CE0) must communicate with, which will simplify and segregate the resulting design of the interconnect.

Subfigure 5.2c) is a re-statement of the data dependencies that each block has on previously-processed blocks, for a block C within a parallel computation round. In addition to C depending on its left block L, it also depends on a *top block*, U. The dependency on L is satisfied with the serial/parallel computation round partitioning, and the dependency on U is satisfied by each CE processing its assigned column of blocks in a serial top-to-bottom fashion. The U block is cached within each CE after it is created as output. Note that the block L will be used by *every* block in the same row, as input, and therefore each

CE processing those columns will be requesting it for reading. Rather than reading the same L block up to N times, we will read this block once and broadcast it to waiting CEs, saving bandwidth and providing a compelling use case for multicast-capable interconnect.

5.1.2 Communication Patterns

Here, we describe the transmissions that occur between the control unit, memory controllers, and CEs. Understanding these communication requirements provides the basis for creating suitable interconnect, and in Chapter 7 will provide the necessary context for understanding the design space of possible network topology optimizations. We begin by summarizing all types of traffic in Table 5.1, which we then explain in further detail.

Table 5.1: Logical communication links used in the LU decomposition engine. Each line of the table represents a family of links, differing in the type of traffic, the source, and/or the destination.

Type	**Source**	**Dest**	**Data Bits**
Writes	CE 0	MEM (any)	272
	CE n $>$ 0	MEM n % M	
Read Requests	CE 0	MEM (any)	20
	CE n $>$ 0	MEM n % M	
	CE (any)	CTRL	
	CTRL	MEM (any)	
Read Responses	MEM (any)	CE (all, *broadcast*)	272
	MEM n % M	CE n	
Go Messages	CTRL	CE (any)	25
Done Messages	CE (any)	CTRL	0

There are two broad categories of transmissions that occur in the system: control messages (Go and Done) between the control unit and CEs, and memory traffic (reads and writes) used to access the memory controllers. The control messages are used to dispatch work to the CEs from the outer algorithm loop and indicate work completion back to the control unit. The Go messages contain a relatively small data payload (25 bits) indicating the assigned matrix column and row bounds. Done messages contain no data payload. Both message types are infrequent and low-bandwidth compared to memory traffic.

Memory accesses are used to read or write off-chip SDRAM in units of one matrix block at a time. A matrix block is a 64 $\times$ 64 array of 32-bit floating-point numbers, represented as 512 words of 256 bits to match the memory controller interface. There are three types of memory traffic: read requests, read responses, and writes. Read requests are relatively small (20 bits), as they only contain an address. They are sent to a memory controller to initiate a read of matrix data. Read responses complete a read request and deliver matrix data back to one or more CEs, and write transmissions deliver a block of matrix data from a CE to a memory controller, together with address information. Read responses and writes contain the block data payload, and are relatively large since they span 256 bits (plus control information) in space and 512 cycles in time.

Writes are unicast transmissions: they are always sent from one CE to one memory controller. However, reads exist in both unicast and multicast varieties. A multicast read, as mentioned above, is used to read a single L block from a memory controller and send it to *all* CEs simultaneously instead of reading the same block N times in a unicast manner. Figure 5.3 summarizes all of the possible types of memory traffic. Subfigure 5.3a) shows unicast read request, read response, and write transmissions.

These can be generally used for all block types. Subfigure 5.3b) depicts the special read requests and responses that occur when a CE requests an L block.

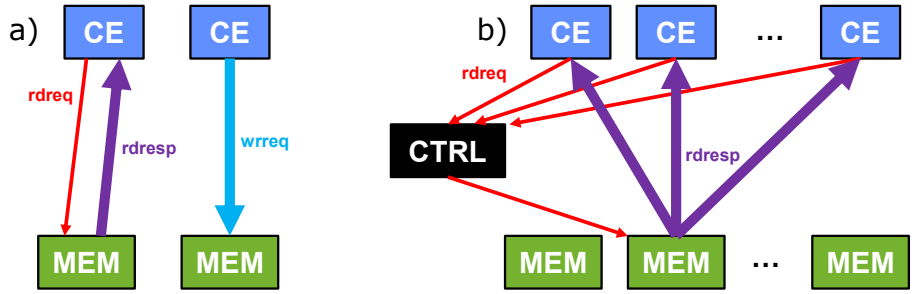

Figure 5.3: **a)** Unicast reads and writes. **b)** Multicast read request and response sequence.

When reading L blocks during a parallel processing round, CEs will direct their read requests to the *control unit* instead of a memory controller. The control unit, which knows how many outstanding CEs are currently dispatched in the processing round, waits for all such read requests to arrive. It then forwards a single copy of the read request to the appropriate memory controller with a special flag set within, such that when the read reply is generated, it is broadcast to *all* CEs simultaneously. This scheme is possible in part due to the active CEs operating nearly in lockstep. The CEs process their assigned columns of blocks top-to-bottom and the block processing times are constant. Figure 5.4 is a more detailed view of the difference between regular unicast read requests/responses and multicast read request/responses for a system with $N = 16$ and $M = 4$.

As previously mentioned, each CE (except CE0) will only ever be responsible for processing certain columns of the matrix, and each memory controller only holds specific matrix columns (see Figure 5.2). This has the effect of segregating the interconnect required to support memory transmissions. For example, for performing writes, not every CE needs to be connected to every memory controller. Figure 5.5 illustrates the necessary connectivity for $N = 8$ CEs and $M = 4$ memory controllers: only CE0 needs to reach all memory controllers when performing writes, but the other CEs only need to write to one memory controller. Read requests have a similar segregation effect, although the control node is included. Read replies *do* require every memory to be able to reply to every CE, but only as a result of having

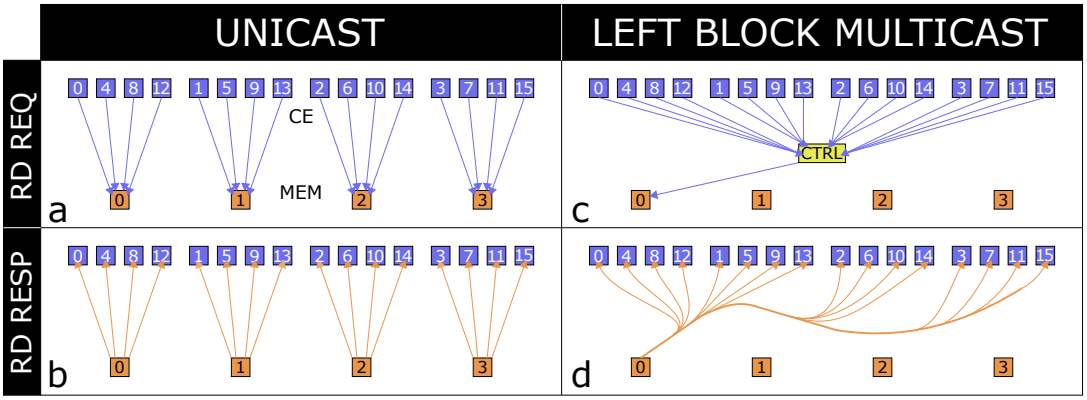

Figure 5.4: **a/b)**: Unicast read requests and read responses. **c)** Read requests for left blocks aggregated by the Control Node and forwarded to a single memory controller. **d)** Left block contents are multicasted from a memory controller to all CEs.

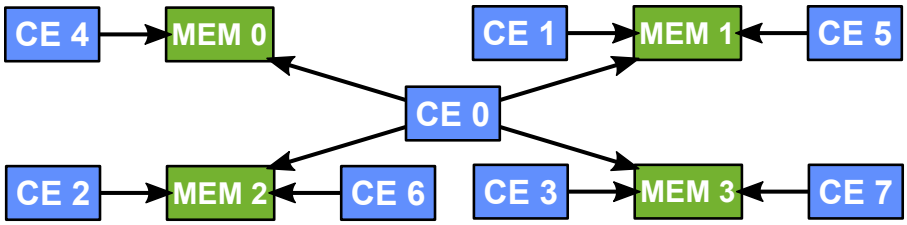

Figure 5.5: Required connectivity between CEs and memory controllers for Write transmissions in a system with $N = 8$ and $M = 4$

to broadcast replies to L block reads. Figure 5.4 provides an example of these segregation effects for all the types of read request and reply traffic. In Chapter 7, these special communication requirements of each type of memory traffic will be exploited to create optimized interconnect.

5.1.3 Compute Element Design

In addition to the outer LU system, we are interested in exploring the automated generation of interconnect for the smaller, finer-granularity system within each Compute Element. Each CE contains several communicating components, shown in Figure 5.6. Of note are five internal caches, each of which can hold one matrix block. They are operated on by the compute pipeline that implements the innermost loops of the LU decomposition algorithm. CE 0 has a more complex pipeline than the other CEs, as it must perform floating-point division. A local control unit orchestrates computation and the reading and writing of matrix blocks, operations that are handled by a marshaller unit.

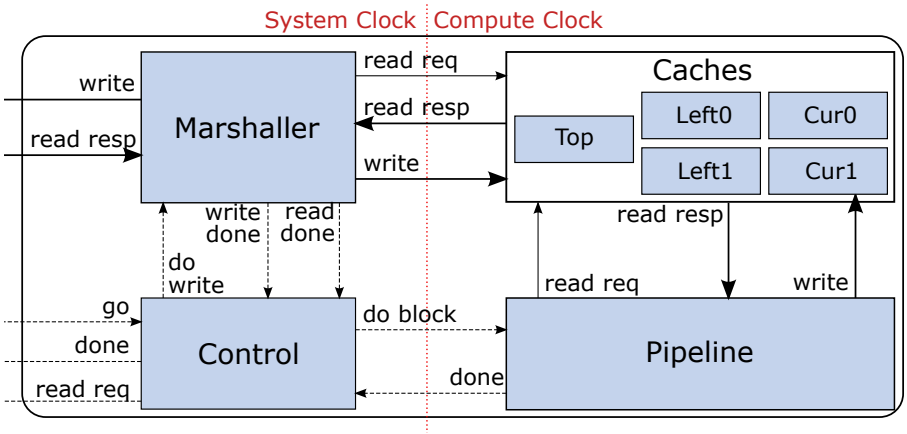

Figure 5.6: Compute Element architecture

The caches are named based on the type of block they hold: Left, Current, or Top. There is only one Top cache, but two Left caches and two Current caches, in order to implement double-buffering. While the pipeline operates on cache blocks L_i and C_i, the marshaller is filling or writing back the other two cache blocks, L_{1-i} and C_{1-i}. The sharing of the caches between the pipeline and marshaller yields rich opportunities for interconnect design, making use of multicast (to fill multiple caches simultaneously) and leveraging user-decreed guarantees of temporal exclusivity between the two components that access the caches. These will be explored in more detail in Chapter 6.

The pipeline and the caches also operate on their own Compute clock domain, separate from the

System clock that is used by the rest of the CE and outside of the CE. Having to also cross clock domains between the marshaller, pipeline, and caches adds additional complexity and optimization opportunities to interconnect design.

The main difference between the inner CE system and the outer LU system lies in the sizes of the communicating functional modules relative to the interconnect. We will show in Chapter 6 that our approach of building interconnect bottom-up from simple primitives yields superior results, with far less interconnect area overhead than existing approaches.

5.2 Convolutional Neural Network

Our second application is a Convolutional Neural Network (CNN) implementation for image classification on FPGAs. It will be used in Chapter 8 as the motivating example for one of our major contributions, which is the introduction of a new type of user specification that allows an interconnect synthesis tool to synchronize a network of fixed-latency functional units while optimizing the generated interconnect for area usage. First, we will present some necessary algorithmic background for the application.

5.2.1 Background

CNNs are a machine learning technique [48] and have been effectively used for speech recognition [4], playing the game of Go [72], and the image classification application [42] that we focus on. CNNs operate in two modes: training and inference. Training 'teaches' the network to classify inputs, and involves feedback. Inference is a feed-forward process that uses the trained neural network to classify inputs. Our hardware supports the inference mode only, expecting the network to have been previously trained offline.

For image classification, each input image is split into its three color channels, and these two-dimensional slices are stacked to form a three-dimensional volume. This volume undergoes a chain of different computation stages ("layers"), each producing an intermediate volume that represents progressively higher-order features of the original image. The final output is a low-dimensional array that directly represents the probabilities of different image categories. The most time-consuming [20] processing stages are the *convolutional layers* from which CNNs derive their name. Our hardware only implements the processing of these convolutional layers.

Each convolutional layer takes its input, convolves it with N_k different kernels, and produces its output. The input, kernels, and output can be visualized as three-dimensional volumes. Each kernel volume is populated by weights that were calculated during the offline training process, and are constants during the inference operation of the convolutional layer. To produce one element (voxel) of the output volume, a dot product is calculated between one of the N_k kernel volumes and an equally-sized subvolume of the input. This is then summed with a per-kernel constant *bias value* that was also produced offline during training. The process of producing one output voxel is illustrated in Figure 5.7, and stated in a more precise mathematical manner in Equation 5.1.

$$out(x_o, y_o, j) = bias_j + \sum_{x=0}^{k_w-1} \sum_{y=0}^{k_w-1} \sum_{z=0}^{k_d-1} kern_j(x, y, z) \cdot inp(x + x_0, y + y_0, z) \tag{5.1}$$

To generate the entire output volume, multiple such dot products are calculated. Each time, a

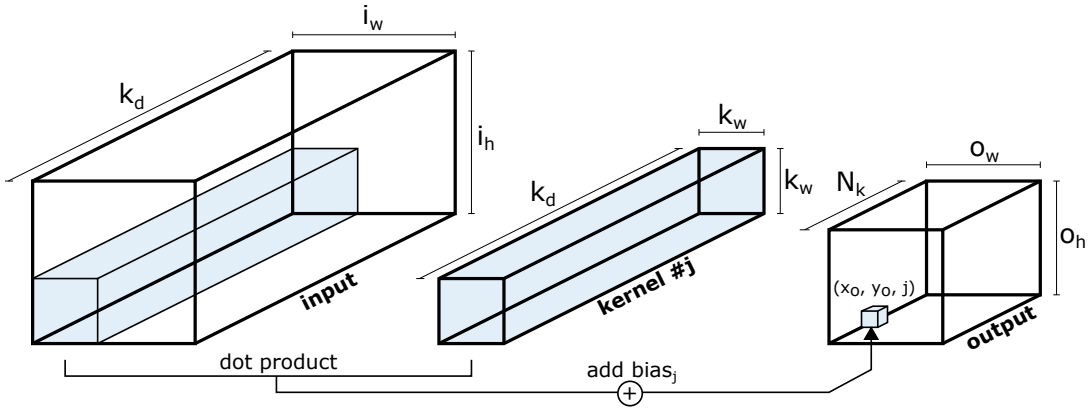

Figure 5.7: Visualization of the dot product between a kernel and a kernel-sized subvolume of the input. This produces a single output voxel.

different kernel and/or a different image sub-volume is chosen. The choice of kernel changes the z coordinate of the output voxel. Moving the boundaries of the sub-volume 'window' within the input volume, in the x and y dimensions, also moves the location of the output voxel in the x and y dimensions. The amount by which the input volume window moves each time in the x or y dimension is called the *stride*. Equation 5.1 assumes a stride of 1 for simplicity, and our hardware also only implements a stride of 1 as a proof-of-concept. Note that during the production of the output volume, many dot product operations share either the same kernel or the same region of the input volume. This will be exploited to produce many output voxels in parallel while sharing data through the use of multicast communication.

5.2.2 Hardware Design

Figure 5.8a shows the top-level design of the CNN implementation. It is capable of processing *one* convolutional layer at a time, which requires storage for the input volume, kernel volumes, and output volumes. All three are held off-chip in DDR3 SDRAM. Working subsets of the image volume and kernel volumes are cached on-chip in the blocks marked "Input Bufs" and "Kernel Bufs" respectively. The "Iterator" blocks generate streams of addresses for reading their respective Input and Kernel buffers, thereby implementing the loops of the triple summation of Equation 5.1. As can be intuited from that equation, many reads of input and kernel volumes are required to produce a single output voxel – there is much more reading than writing. Therefore the output is left un-cached and written directly to off-chip memory by the "Output Writer" block.

The compute array performs many dot-product operations in parallel using $N \times M$ *dot product units* (DPUs), each of which produces one output voxel. Figure 5.8b provides a close-up view of the compute array and connections between the DPUs, iterators, and buffers. The N parameter indicates the number of input buffers, and M the number of kernel buffers. Changing N and M affects the parallelism and computational capability of the hardware, and is unrelated to the sizes of the input, output, and kernel volumes, which are runtime-adjustable parameters. Each DPU receives input and kernel data every cycle, which are multiplied and internally accumulated into a single output voxel, implementing the volume dot product operation. At the end of this operation, each DPU shifts out its output voxel to the right of the figure, towards the Output Writer for storage into off-chip memory.

Input, kernel, and output voxels are 16-bit fixed point values. Every connection in Figure 5.8 between

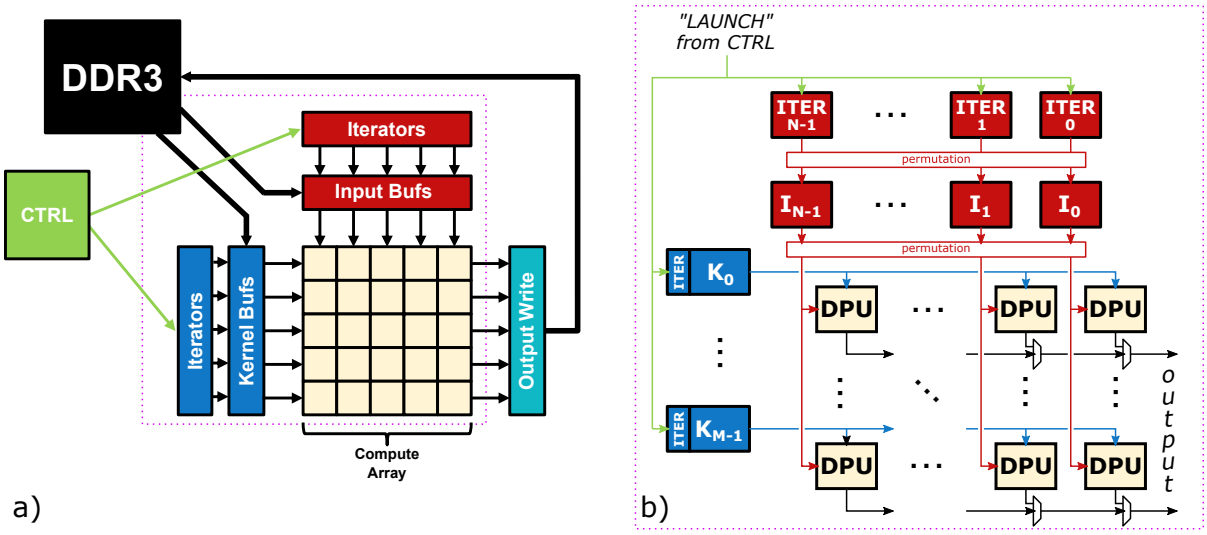

Figure 5.8: **a)** Top-level block diagram of the CNN engine. **b)** Detailed view of the iterators, buffers, and compute array.

the input/kernel buffer outputs and the DPU inputs is 256 bits wide, carrying 16 voxels. Thus, each DPU receives 16 kernel and 16 input voxels per cycle. Internally, a DPU is an array of 8 independent Arria 10 DSP blocks configured in multiply-accumulate mode. Each DSP block performs two 16-bit multiplies: one image voxel multiplied with one kernel voxel, and another image voxel multiplied with its corresponding kernel voxel. The two products are summed internally within the DSP block, producing a total of eight partial sums, one for each DSP block. These eight, 16-bit partial sums are fed through a soft adder tree to produce the final 16-bit output voxel value. The final addition of the bias value is performed outside the DPUs, in the Output Writer block that writes to off-chip memory.

5.2.3 Operation

A dot product operation iterates over two 3D volumes of voxels of equal size. One volume is the entirety of a kernel, and the other is a kernel-sized subvolume of the input volume. For correct operation, a voxel from the kernel, and a matching voxel from the input, *must arrive simultaneously* at a DPU at a given clock cycle. This is complicated by the fact that the image and kernel data may take paths of unequal length to reach each DPU. We will now explain these paths in detail, as they are central to the design problem that will be later explored in Chapter 8.

The beginning of a dot product compute phase is globally synchronized by a 1-bit "launch" signal generated by the control unit, and is received by all input and kernel iterator units. The iterators then proceed to generate one 9-bit voxel read address per cycle. These addresses are directed towards an input, or kernel, buffer. Reading kernel buffers is straightforward – there is a 1-to-1 connection between an iterator and a kernel buffer. However, reading image buffers is more complicated. To provide the correct input voxel to each DPU, any given iterator reads from a *different* image buffer every cycle, in a permuted fashion: during clock cycle x, iterator y reads from image buffer $I_{(x+y)\%N}$. The need for such connectivity is illustrated in Figure 5.8b by the "permutation" blocks in the input volume data paths, which are shorthands for (logical) full crossbars. After being read, every buffer (kernel and input) produces a 256 bit wide output containing 16 sequential 16-bit voxels, and these are broadcast to a row

or column of DPUs within the compute array.

Due to the asymmetrical complexity of the kernel and input volume data paths, it may be necessary to pipeline the latter to mitigate the cost of the *two* $N \times N$ full crossbars, one of which is 256 bits wide. Doing so would introduce skew between the kernel and (now pipelined) input delivery paths to a DPU, which would have to be corrected. There are many ways to accomplish this, each with a trade-off in area usage and design complexity for the user. In Chapter 8, we will study this synchronization problem and use it to motivate one of our contributions: a method for an interconnect synthesis tool to automatically provide fixed-latency synchronization given high-level data delivery constraints from the user, while simultaneously attempting to minimize area.

The important transmissions occurring within the CNN engine are summarized in Table 5.2. Two "loader" modules, one for kernel and image data, are not shown in Figure 5.8 and represent the interface to OCM (off-chip memory) and periodically update the contents of the kernel and image buffers. Note the many broadcast transmissions for kernel and image data from on-chip buffers to DPUs. Each image buffer sources N possible transmissions, with each transmission broadcasting to one of the N columns of DPUs. No two image buffers broadcast to the same column simultaneously, and this is reflected through mutual exclusivity constraints on the transmissions.

Table 5.2: Logical communication links used in the CNN inference engine. Each line of the table represents a family of links, differing in the type of traffic, the source, and/or the destination.

Type	**Source**	**Dest**	**Data Bits**
OCM Fill	K Loader	K Buffer (any)	256
	IMG Loader	IMG Buffer (any)	
Launch Signal	CTRL	ITER (all)	0
		K Buf (all)	
Read Address	ITER (any)	IMG Buf (any)	9
K data	K Buf i	DPUs (all in row i)	256
IMG data	IMG buf (any)	DPUs (all in column 0)	256
		DPUs (all in column 1)	
		..	
		DPUs (all in column N-1)	
Result	DPU (any)	Output Writer	16

Chapter 6

Fine-Grained Interconnect Synthesis

This chapter demonstrates the first of our three contributions to FPGA interconnect synthesis: the ability to automatically generate and optimize interconnect for *fine-grained* systems. Originally, this work was presented in two papers [61, 62]. The results in this chapter have been updated to use newer versions of tools (both GENIE and Qsys) and synthesized for a newer FPGA architecture.

We begin this chapter by defining the fine-grained design context and how it differs from the traditional (coarse-grained) context that is the main use case for existing automatic interconnect synthesis. This will be followed by a more detailed examination of features within the GENIE synthesis flow that provide benefit for fine-grained designs. Then, we will show the effectiveness of our approach by using GENIE to build the interconnect for the LU decomposition engine introduced in Chapter 5.1, and compare it against two other implementations: one created by an existing tool (Intel Qsys Pro[7]), and one in which the entire system, including interconnect, is created manually in Verilog. In addition to measuring area and clock frequency, we are interested in attempting to qualitatively, and quantitatively, measure the *ease of use* experienced by the designer in creating the application using each of the three approaches.

6.1 On Design Granularity

We use the term 'granularity' to refer to the size and complexity of functional modules that communicate over an interconnect fabric. Originally, the desire to create a distinction between 'fine-grained' and 'coarse-grained' functional modules arose from attempting to characterize the typical use cases for existing interconnect synthesis tools. In order for a functional module to participate in automated interconnect synthesis, its interfaces must conform to a protocol expected by the tool. The types of required signals specified by a protocol will impose a minimum complexity for any module that implements them. If this complexity is unacceptable or inconvenient, it discourages the use of the tool to build interconnect in an automated fashion.

For example, for a tool which generates memory-mapped interconnect, modules that can express their communications as memory reads and writes will be more suitable than ones that don't. In this case, processors and peripherals would be a natural fit, but using the same tool to connect together *sub-modules* of the processor, such as its register file and control logic, would not be a productive use case. Instead, such sub-components would be connected together with ad-hoc, tightly coupled interconnect

that does not have the assumptions and requirements of a heavyweight interface protocol – they, and the interconnect that joins them, are of *finer granularity* than their parent processor module in the design hierarchy.

Another more general feature present in communication protocols is backpressure, in the form of a 'ready' signal or a credit system [26]. This allows modules to operate correctly in the face of variable communication latency, either due to dynamic traffic congestion or due to the implementation and topology of the interconnect itself. Backpressure allows modules to temporally decouple, on a cycle-by-cycle basis, from the operation of other modules in the system, and enables modularity and design re-use. However, it too imposes a minimum baseline complexity for the design of a module. If an automated interconnect-building tool's protocol demanded the use of a backpressure signal, an application would be required to implement it even if it did not have use for it (for example, if the communication modules used tightly-coupled cycle-accurate global synchronization). Even if this is as simple as tying a ready signal to a constant '1', the interconnect complexity would be greater than necessary for the task. This may discourage the designer from using an automated tool to build interconnect, and they may instead opt to use ad-hoc fine-grained interconnect, rather than the latency-insensitive interconnect generated by the tool.

To add to protocol-related concerns, the other aspect of a module's granularity is its size. Even if a module is a natural fit for a communication protocol imposed by an interconnect synthesis tool, it may still represent an unproductive use case if the expected area overhead of the generated interconnect is sufficiently large compared to the area of the module. From these considerations, we can summarize some characteristics of fine and coarse granularity. Coarser-grained modules and interconnect:

- Use more complex interface protocols that require signals that either are specific to one style of communications (with memory-mapping as an example), or require backpressure signals that temporally decouple the module from the interconnect.
- Tend to be larger, both due to the required protocol-induced complexity and due to avoiding interconnect area overhead relative to module size.

In contrast, finer granularity is characterized by:

- Simpler protocols with fewer mandatory signal types, enabling a wider range of communication styles. The theoretical lower limit would be the un-typed wires provided directly by HDLs.
- Smaller in size, with a single lookup table forming a lower limit.

These are not hard definitions, but intuitively observed trends. It is possible, for example, for there to exist a memory-mapped protocol that does not use backpressure and requires the user to specify a fixed round-trip read latency through their modules, but still generate large and heavyweight interconnect that would be classified as coarse-grained. Conversely, it is possible for an otherwise-lightweight protocol to require the use of backpressure, but for the associated tool to generate interconnect with low enough area overhead to be classified as fine-grained interconnect and permit productive use in automatically connecting smaller modules.

The case for GENIE's interconnect being fin*er*-grained than that of existing tools was made in Section 3.1 through the bottom-up approach of combining simple interconnect primitives. What we show in this chapter is that this approach allows productive automation of interconnect synthesis for systems with finer-grained components.

6.2 Fine-grained GENIE Flow Features

The original papers [61, 62] covered by this chapter were our first published works on GENIE and its interconnect synthesis flow. We introduced and described features of this flow that simplified and aided the creation of fine-grained interconnect. Three of these features were Latency Introspection, conflict-free Merge nodes, and the automated insertion of clock crossing logic, each of which is described in turn in the following sub-sections.

6.2.1 Latency Introspection

The fine-grained design context is associated with a tighter coupling of functional modules with interconnect. One form that this coupling takes is cycle-accurate knowledge of communication latencies. When backpressure is not used, functional module design is simplified as it does not need to accept or respond to stalls generated by the interconnect. However, without backpressure, the functional module design and interconnect design must now closely cooperate to ensure correct behavior. Our initial solution to this problem was a feature called Latency Introspection, which allows a functional module, at compile time, to query the latency in cycles of generated interconnect paths. The functional module could then use this knowledge to pipeline and delay its *internal* signals by the correct amount of cycles to re-align with returned data from the interconnect.

In GENIE, Latency Introspection is achieved with an API call within the user specification. As parameters, it takes a previously-declared Routed Streaming link between two functional module interfaces, and the name of a Verilog parameter. Upon interconnect generation, the flow declares that parameter within the auto-generated Verilog and assigns it the latency, in cycles of the requested path. The parameter can be passed into functional modules in the system so that they can adjust their internals accordingly.

Later, GENIE was augmented with Synchronization Constraints and automatic register insertion, which supersedes some of the use cases of Latency Introspection. Instead of informing functional modules of fixed interconnect latencies and forcing the designer to compensate, these new features allowed GENIE, and the generated interconnect itself, to compensate for fixed latencies outside of functional modules. These features will be later covered in Chapter 8.

6.2.2 Conflict-Free Merge Nodes

GENIE provides the user the ability to specify extra hints about their application's communication behavior. One of these hints is the assignment of transmissions to mutual-exclusion groups. This specifies that transmissions belonging to one group will *never* occur at the same time as transmissions from another group, guaranteed by the user's functional module design. If a user does specify mutual exclusion groups, and if temporally mutually-exclusive transmissions are present at the inputs of a generated Merge node in the interconnect, the GENIE flow will convert that Merge node into its no-conflict variant (see Section 4.2.3). The Compute Element design explored in this chapter contains such transmissions, providing an opportunity for interconnect optimization. It is another example of the cooperation of application and interconnect design that is characteristic of fine-grained systems.

6.2.3 Automatic Clock Crossing Insertion

One of the stages of the GENIE flow is the automated insertion of clock crossing logic. It is driven by an algorithm that optimizes clock domain boundaries within generated interconnect in order to reduce area. While relevant to all design contexts, it can be especially important in fine-grained designs as interconnect represents a potentially significant fraction of total system area. This is the case within the Compute Element design being studied, as it includes a clock domain transition.

When a design contains multiple clock domains, there is an interesting optimization problem that arises when crossing between any two domains: where in the generated interconnect network should the transition occur? For example, consider a GENIE Split node that has its inputs driven by a functional module belonging to one clock domain, and its outputs connected to functional modules of a second clock domain. Placing a clock crossing at the input of the Split node would be cheaper than inserting two crossings, one at each output. When the network contains a complex topology of Split and Merge nodes, the optimal choice may not be obvious. GENIE intelligently chooses the point at which the minimum total number of signals undergo the crossing, because each signal incurs a non-trivial cost. The process occurs in two phases: determination of clock crossing boundaries, followed by the insertion of clock crossing primitives (described in Section 4.5) at the determined clock domain boundaries.

Algorithm 2 represents the work of the first phase, determining the clock domain boundaries within a GENIE interconnect domain (which is a connected component of the entire system). It represents the interconnect domain as a directed graph $G = (V, E)$. Vertices represent Routed Streaming Interfaces belonging to both designer-specified functional modules and those of internally-generated interconnect primitives. $C(v)$ represents the clock domain of each vertex, and initially only some vertices will have this assignment, from a set of domains K. The objective of the algorithm is to find the best clock domain assignment for each of the remaining unlabeled vertices.

Directed edges between the vertices represent physical connectivity between the corresponding module interfaces. Each edge e has a weight $W(e)$ that represents the cost of placing a clock domain crossing there, and this weight is initialized at graph creation time. The weight/cost is a function of the number of data bits on that link plus the nominal fixed overheads of maintaining a dual-clock FIFO such as read/write pointer registers. The clock domain assignment problem is formulated as a multi-way-cut [24] problem: to partition the graph into connected components (one for each clock domain) while minimizing the total weight of the boundary edges between them. This problem is NP-hard for more than two clock domains, and Algorithm 2 is based on a greedy approximation which, instead of considering all clock domains simultaneously, looks at two clock domains at a time.

The algorithm requires that each clock domain be represented with a single *terminal vertex*. We create each terminal in the set T by merging together all vertices that share the corresponding clock domain. The vertex merging procedure is explicitly laid out in Algorithm 3. We then assign one clock domain at a time by repeatedly finding a minimum cut between one of the terminal vertices t_i and a merged vertex representing all other terminal vertices s_0. This is accomplished with a standard min-cut (dual of max-flow) algorithm [22] which returns the total weight of the minimal cut $cost_i$ and a residual graph R_i. A greedy decision is made to choose the terminal vertex that yields the least-cost two-way cut. The vertices in that terminal's partition are assigned its corresponding clock domain, and are removed from the graph. With one fewer unassigned clock domain, the process repeats until all vertices are assigned a domain.

Finally, after the algorithm terminates and all RS Interfaces have an assigned clock domain, GENIE

ALGORITHM 2: Area-Minimizing Clock Domain Assignment

inputs : connectivity G=(V,E), edge weights W, *partial* clock assignments C, clock domains K
output: clock assignments C for *all* $v \in V$
$T := \{\}$
`// Collapse all vertices that share a clock domain, add them to` T
foreach $k \in K$ **do**
  $U_k := (u_0, u_1, u_2, \cdots \in V \mid C(u_i) = k)$ `// all ports driven by clock` k
  MergeVertices(G, U_k)
  $T := T \cup \{u_0\}$ `// one terminal vertex per clock domain`
end
`// Assign clock domains one at a time, removing them from a copy of G as we go`
$H := G$
while $|T| > 1$ **do**
  `// Try the terminal for each remaining unassigned clock domain`
  foreach $t_i \in T$ **do**
    `// Merge the terminals for all the other domains into a single vertex` s_0
    $H' := H$
    $U := (s_0, s_1, \cdots \in T \mid s_j \neq t_i)$
    MergeVertices(H', U)
    `// Find the min-cut between source` t_i `and sink` s_0
    `// Memoize the residual graph` R_i `and total cut weight` $cost_i$
    $R_i, cost_i :=$ MinSTcut(H', W, t_i, s_0)
  end
  `// Choose the clock domain terminal that yielded the smallest-weight cut`
  $cost_{best} :=$ smallest $cost_i$
  `// Assign all reachable vertices the corresponding clock domain`
  foreach $v \in R_{best}$ *reachable from* t_{best} **do**
    $C(v) := C(t_{best})$
    remove v from H
  end
  $T := T \setminus \{t_{best}\}$
end
`// Only one clock domain remains, do a trivial assignment`
foreach $v \in H$ **do**
  $C(v) :=$ C(the one member of T)
end

ALGORITHM 3: MergeVertices(G, U)

inputs : G=(V,E), list of vertices to merge U=$(u_0, u_1, u_2, \dots)$
output: updated G with vertices from U merged into u_o
$E := E \setminus \{(x, y) \mid x, y \in U\}$ `// remove edges between members of` U
`// Transfer outgoing edges to` u_0
foreach $(x, y) \in \{E \mid x \in U_k \wedge y \notin U_k\}$ **do**
  $E := E \setminus \{(x, y)\}$
  $E := E \cup \{(u_0, y)\}$
end
`// Repeat for incoming edges`
`// ...`
$V := V \setminus \{u_1, u_2, u_3, \dots\}$

inserts clock crosser primitives at clock domain boundaries. It is worth noting that the problem of minimizing clock crossing in networks-on-chip has been previously studied. Kulkarni et al. [45] have examined the runtime and accuracy trade-offs of using brute force, exact (using an ILP formulation), and heuristic approaches to solving this optimization problem. Our heuristic differs from theirs, and

considers the varying cost (due to data width) of each interconnect link, rather than minimizing the number of clock crossing points alone.

6.3 Compute Element Design

This section is a continuation of Section 5.1.3 which briefly introduced the design and functionality of a Compute Element (CE) within the larger LU Decomposition Engine example application. The CE will serve as the design to demonstrate GENIE's effectiveness at building interconnect for fine-grained systems. We will now describe in more detail the logical transmissions that occur within a CE and provide context for the fine-grained interconnect that will be built to carry these transmissions. Later, we will compare different implementations of this interconnect, built using GENIE and other tools.

6.3.1 Structure and Functionality

We begin with an overview of how the CE and its components operate, as this is important for understanding the communications between them. A CE processes a specific column of blocks from the matrix being operated on by reading the blocks from external memory and writing back transformed data in their place. Figure 6.1 (a duplicate of Figure 5.6) shows the communicating functional modules within a CE, of which there are eight:

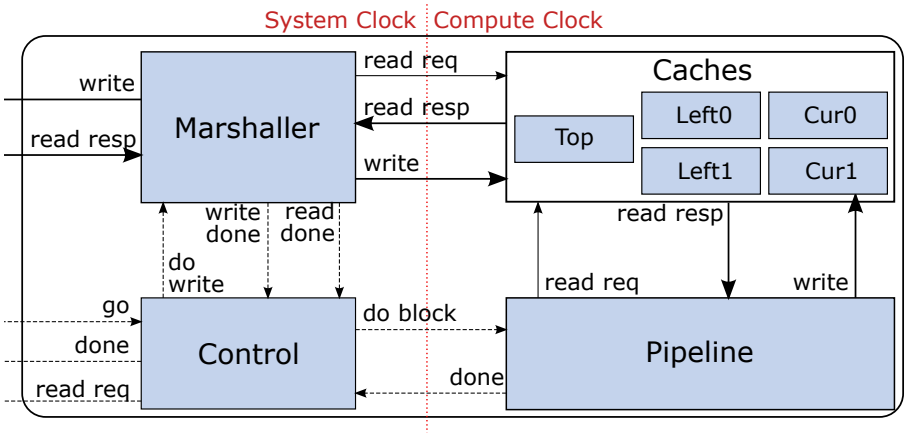

Figure 6.1: Compute Element architecture, restated)

- A Control unit to orchestrate the fetching, processing, and writing back of blocks.
- Five Caches that store the matrix blocks being operated on locally within the CE.
- A computation Pipeline, which reads from and writes to the caches to produce the processed results.
- A data Marshaller to transfer matrix blocks to and from the caches and external memory outside the CE.

There are five independent dual-ported cache blocks in total: Top, Left0, Left1, Current0, and Current1. They are named after the types of blocks they store during processing, and relate to the

spatial relationships between the cached blocks within the larger matrix. The Left and Current blocks are also double-buffered for increased performance, with the numerical suffix indicating which buffer it belongs to. While Cur_i and Left_i are being accessed by the Pipeline, the Marshaller accesses Cur_{1-i} and Left_{1-i} for transferring blocks from or to off-chip memory. The Top block is rarely written to (once by the Marshaller, and once by the Pipeline at the very top of the column of blocks) and does not require double-buffering.

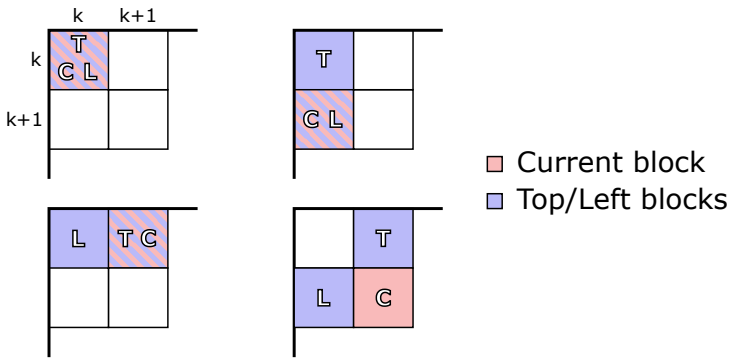

Figure 6.2: Aliasing of Top, Left, and Current blocks within the current submatrix, whose top-leftmost block has coordinates (k, k).

The CE's purpose within the LU Decomposition Engine is to process all the blocks within its assigned column of blocks from the current sub-matrix of the outer loop of the algorithm (see Algorithm 1). To process one block at coordinates (i, j), the CE requires three as input: the existing block at (i, j) called the Current block, the block at (i, k) called the Left block, and the block at (k, j) called the Top block. The Marshaller is responsible for filling the associated caches before the Pipeline can operate on them. Afterwards, it writes the transformed Current block back to off-chip memory. For edge cases where $i == k$ or $j == k$ these blocks can alias to one another as shown in Figure 6.2. Due to this aliasing, the Marshaller and Pipeline will sometimes write the same data to two or three different caches. In the Marshaller's case, when it writes duplicate data it also writes to the same addresses of the target caches, which is behavior that maps well to a single multicast transmission.

The CE has two clock domains in order to decouple the performance requirements of processing matrix blocks and transferring them to and from off-chip memory. Processing a block requires $O(n^3)$ accesses to matrix elements by the Pipeline, while reading and writing back a block's contents to off-chip memory is $O(n^2)$ where n is the block size (64 in our case). The Pipeline and Caches operate using the "Compute Clock" and the rest of the design uses the "System Clock", including the coarse-grained interconnect linking the CE with the greater LU Decomposition system. The boundary between the two clock domains is shown in Figure 6.1.

6.3.2 Communication Behavior

There are two classes of communications present within the CE shown in Figure 6.1: low-throughput control messages (shown as dashed arrows), and high-throughput matrix block read requests, read replies, and writes (shown as solid arrows). The former, while being point-to-point and not performance-demanding, can still benefit from automated interconnect synthesis rather than being implemented by hand, either because of the need to cross clock domains (Control to Pipeline), or the potential need to

pipeline the links to close timing later in the design cycle.

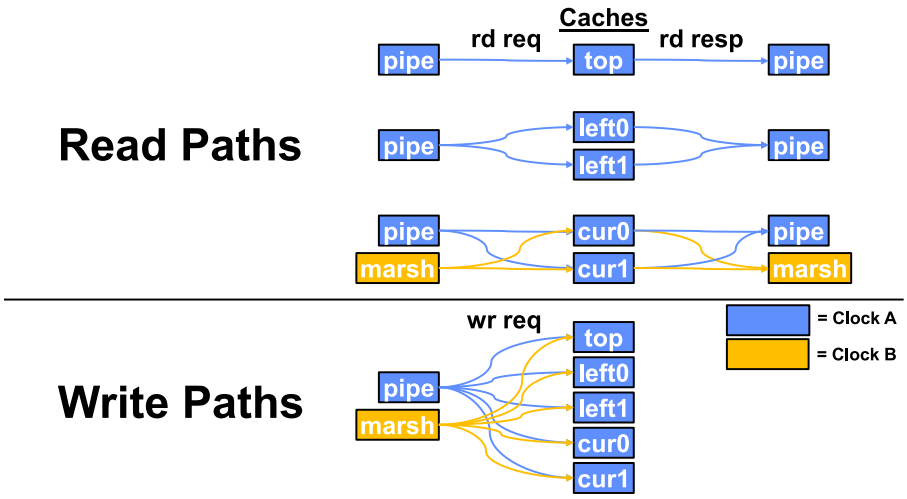

Figure 6.3: High-throughput read and write transmissions between the Marshaller, Pipeline, and Caches

In contrast, the high-throughput communications links, whose logical connectivity is depicted in Figure 6.3, require high-performance and non-trivial interconnect. They send data words (or requests for data words) every cycle, and originate or terminate at the read or write port of one of the five Caches. The communication requirements for each block type (Top, Left, and Current) are asymmetrical:

- The Top cache (of which there is only one) is only ever read by the Pipeline.
- The two Left caches also only read by the Pipeline, but only one at a time.
- The two Current caches can be read by both the Pipeline and the Marshaller simultaneously, but in a non-overlapping fashion.
- All of the five caches can be written by either the Pipeline or the Marshaller. Either can write to multiple caches from the sets {Top, Cur0, Left0} and {Cur1, Left1} simultaneously, but the Pipeline and Marshaller will *never* write to both sets.

These communication requirements represent different levels of complexity, ranging from what amounts to trivial direct links (for the Top cache reads) to the complex spatial and temporal relationships of the write paths. For all but two links (Pipeline to Top Cache reads), some mix of one-to-many or many-to-one communications is needed, requiring distribution or arbitration hardware within the interconnect. The Pipeline and Marshaller perform writes of identical data to some subset of the Caches, which changes at runtime, and thus requires either multicast capability or multiple write ports. The double-buffering of the Caches is explicitly managed by the application and guarantees that the Marshaller and Pipeline *never* compete for the same Cache's read or write port. This application-specific behaviour presents opportunities to optimize the design of the interconnect to reduce area and increase performance.

Read requests are 12 bits wide, and specify an address within a cache. Read replies are 256 bits wide, and carry multiple words of data to feed the Pipeline's SIMD datapath. Writes contain both an address and data and are 268 bits wide. It is important to mention the relatively large width of these connections, since it makes the interconnect's area usage that much more sensitive to its architecture.

6.3.3 Three CE Variants

To evaluate GENIE's fine-grained interconnect synthesis capability, we create three different implementations of the CE and its interconnect: one generated by GENIE, a manually-written and optimized reference design, and one generated using Intel's Qsys[7] system integration tool. This allows us to compare GENIE against the best possible hardware (at the expense of design time) and against an existing automated synthesis tool (at the possible expense of one or more performance or area metrics).

Each variant is a different realization of the CE system shown in Figure 5.6. In the Qsys variant, two different communication protocols are used: Avalon-MM (memory-mapped), and Avalon-ST (streaming) [34]. Connections to the Caches map naturally to random-access reads and writes, so we implement those using Avalon-MM, using an extra address bit to select between buffers of double-buffered Caches. The remaining connections, which are point-to-point and have no memory-like semantics, are implemented using Avalon-ST.

In the GENIE variant, all connections in Figure 5.6 are implemented using Routed Streaming or Conduit interfaces. The Marshaller and Pipeline interfaces that write to the caches use multicast addressing: every possible combination of destination caches (which is fewer than the maximum of 2^5) is mapped to a unique Source Address on the initiating interface. The *memory* address (for writes, and for read requests) is actually part of the data payload as an (additional) signal of type `data`.

The manual variant's interconnect is designed by hand in SystemVerilog, using application-specific design optimizations. Some optimizations, and interconnect design, match what GENIE is capable of producing automatically, which is already quite minimal. However, some optimizations are not available in GENIE, making the manual variant theoretically better-performing. Notable examples include:

- **Clock crossing:** The manual variant directs *all* traffic flowing from one domain to another (and with similar backpressure requirements) into a *single* clock crossing FIFO, rather than separate FIFOs for each transmission as GENIE does. While the total number of bits undergoing crossing remains the same in both cases, combining all signals into fewer FIFOs amortizes the cost of FIFO control logic and achieves better packing of data into FPGA distributed RAM blocks.

- **Centralized double-buffering control:** Some of the caches are double-buffered. When the Marshaller and Pipeline write to, or read from caches, the choice of *which* buffer to access is contained within the read or write request in the GENIE version – the Split and Merge nodes in the interconnect are controlled locally by the data stream. In the manual version, we the designers recognize that buffers are swapped rarely and not on a cycle-by-cycle basis, so some of the buffer steering interconnect can be controlled *centrally* from the Main Control unit as it prepares to launch a computation or data marshalling phase. This blurring of functional module and interconnect boundaries is possible when the entire design is created by hand.

Additional differences between the three variants are described in the next section.

The manual variant existed before the development of GENIE and served as an inspiration for the kinds of automation and optimization features that would be desirable in a new tool. However, we attempted to capture and "factor out" a design paradigm in a generic way and not to specialize GENIE for perfectly re-creating the manually-generated interconnect of the CE, as described above in the differences between the manual and GENIE-generated variants.

6.3.4 Variant Implementation Comparison

Here, we highlight some important differences between the interconnect implementations of each variant in order to give some context to the results in Section 6.4. The goal of automation is to improve designer productivity while generating hardware with acceptable area and performance. To that end, we also hope to provide a *qualitative* picture of the design effort required to create each variant. We will focus on how each variant handles the following aspects of the CE design, since they required the greatest interconnect complexity:

- Clock domain crossing
- Marshaller-to-Cache read path
- Pipeline-to-Cache read paths
- Cache write paths

Clock Domain Crossing

Both the Marshaller-to-Cache connections and the Control-to-Pipeline connections cross clock domain boundaries, which is handled differently among the three variants. In the manual variant, there exist three clock-crossing FIFOs for the whole design: one for connections travelling from Compute Clock to System Clock (read replies) and two in the other direction (one for read requests, and one for all other non-backpressured signals combined together into a single bus). There is also a full-handshake-based clock crosser for the single-bit 'done' signal that is emitted from the Pipeline after every block is processed. Its inability to be combined into the data portion of the three other FIFOs meant it would otherwise have to have its own FIFO – which is unnecessary overhead for an intermittent single-bit signal.

Qsys performs automatic clock crossing for Avalon-MM connections, inserting dual-clock FIFOs when a master and slave are on different clock domains. However, it inserts FIFOs *after* routing traffic to multiple destinations, causing each destination path to have its own FIFO, including 9 FIFOs which must accommodate the cache read/write data width (256+ bits). Finally, no automatic clock crossing is performed on the Avalon-ST connections for the low-bandwidth control messages, requiring manual instantiation of clock crossing adapters from the Qsys component library.

The GENIE implementation has one clock-crossing FIFO for each connection (for a total of five), rather than the two used in the manual variant. All Marshaller-to-Cache write paths share a single FIFO, which GENIE inserts *before* a split node that broadcasts to up to five caches. This was determined using the algorithm described in Section 6.2.3. The total number of FIFO memory bits is thus identical to the manual variant, but there is extra logic overhead since each FIFO requires its own read/write pointer and metastability protection registers. The upside is that all Routed Streaming connections receive automated clock crossing, with no designer intervention needed.

Marshaller-to-Cache Reads

Cache reads from the Marshaller need to be able to stall if the system outside the CE is unable to accept the outgoing read reply data. Both the GENIE and Manual variants use the backpressure generated

outside the CE, in the form of a `ready` signal, as a means to stall the reading of the Marshaller-Caches-Marshaller read pipeline on a cycle-by-cycle basis.

The Avalon-MM protocol has backpressure for read requests in the form of the `waitrequest` signal role, allowing slaves (the Cache read ports) to stall the Master (the Marshaller). However, there exists no signal that allows the Marshaller to stall read data returning from the Caches. Our solution was to add a FIFO to the Marshaller to buffer this data until it can be sent outside the CE, and reserving space in this FIFO before sending any read requests to the Caches. Note that this missing functionality in Qsys requires extra effort for the designer to mitigate, while also costing area. This is not a general limitation of memory-mapped protocols, as, for example, AMBA AXI[9] has backpressure support for request and reply paths, but is itself cumbersome to use since many signals are mandatory.

Pipeline-to-Cache Reads

Read and write access to (some) of the cache blocks is shared between the Pipeline and Marshaller. However, due to double-buffering of the Caches, and careful orchestration by the Control logic, the design of the CE guarantees no competition between the Pipeline and Marshaller for the same buffer. This is ideal, because in theory it allows the Pipeline to operate as if it has sole point-to-point access to the Caches with the benefit of deterministic latency, simplifying the design.

This is the case in the hand-made variant. The Pipeline's access to the two buffers of the caches is achieved with a multiplexer controlled directly by the Control logic. It switches buffers at the correct times, ensuring that the Pipeline and Marshaller never compete for cache accesses. The round-trip latency of the read path through this interconnect and through the caches themselves is fixed and known by the Pipeline, allowing it to use simple register chains to delay other signals, instead of a more expensive latency-insensitive construct such as a FIFO.

GENIE's implementation of the read path is similar to the hand-made variant. The specification for the read request/response paths contains a hint that the Marshaller and Pipeline will never simultaneously compete for the same Cache ports, resulting in simplified Merge nodes that are equivalent to the muxes of the hand-made variant. The effective difference is that these muxes are controlled locally (by the incoming Valid signals) rather than centrally by the Control logic. Using Latency Introspection (described in Section 6.2.1), the Pipeline is able to know the exact fixed latency of the GENIE-generated read path interconnect and can avoid using FIFOs, just like the hand-made version.

The Qsys interconnect inserts arbiters that allow both the Marshaller and Pipeline to access the read ports of the two 'current' caches Cur0 and Cur1, which are the only caches among the five which can be read by *both* the Pipeline and Marshaller. Qsys has no way of knowing our application's dynamic behavior, and the inserted arbiters are designed for the worst/general case in which simultaneous competition is possible. This increases the interconnect complexity to more than what is necessary. It also forces us to make changes to the Pipeline module. Even though we, the designers, know that the Pipeline should never encounter competition for reading the Caches, Qsys refuses to generate the system unless we add the Avalon-MM `waitrequest` signal as an input to the Pipeline's master interface, to handle the potential stalls that are expected in the general case. Even after satisfying this requirement of adding the signal, it became evident that we could not even *ignore* its presence – stalls were being inexplicably generated by the interconnect despite the transmissions facing no competition. This is visible in the simulation fragment shown in Figure 6.4 as a ready signal that is deasserted for one cycle.

The stall always manifests in this manner: a deassertion for one exactly one cycle. There are two

Figure 6.4: Unexpected stall generated by Qsys interconnect

methods to handle this stall within the Pipeline. The most correct and general way is to insert a FIFO, which increases cost and complexity. The other method, which will only work in the case of *this* particular observed stall behavior, is with a pair of registers and a multiplexer, which have less impact on area and timing than the FIFO. Both versions will be explored as sub-variants of the Qsys CE design variant when we report results in Section 6.4.

The preceding issue affected only reads to the two Current Block caches. The other caches either do not require the addition of a stall signal, or the signal can be truly ignored without further modifications to the affected functional modules, as no stalls of the type in Figure 6.4 occur. The issue for those is now the determination of the fixed round-trip read latency through the caches. Without a Latency Introspection mechanism like the one GENIE has, we were forced to either study the generated Verilog, or perform simulation, to discover these latencies.

This detailed study into the differing implementations of the Pipeline-to-Cache read request and read reply networks illustrates the extra analysis, simulation, and overall design effort required to use an automatic interconnect synthesis tool in a use case for which it is not designed.

6.3.5 Cache Write Network Topology

The GENIE interconnect's Split and Merge nodes can be arranged to form many different topologies. As described in Section 3.4, each interconnect domain in a system can have its own topology, and can either be generated automatically or be customized manually. By default, GENIE creates a 'sparse crossbar' topology automatically based on each domain's user-specified logical connectivity. At the time of the writing of this chapter's original representative papers, GENIE did not yet have the ability to further optimize an auto-generated sparse crossbar topology to minimize its area, based on application communication patterns. However, manual specification of topologies was still possible. In this section, we apply this ability to the fine-grained CE system, exploiting an opportunity to reduce the area of one of the network domains: the Marshaller-to-Cache and Pipeline-to-Cache write requests. By doing so, the GENIE-generated interconnect is able to take advantage of the CE's unique communication patterns to reduce area in the same way that the manual variant of the CE does. Doing so will require extra effort from the designer, but less so than creating such specialized interconnect by hand.

The default topology used by GENIE is its built-in *sparse crossbar* topology. It is programmatically generated according to two rules:

1. Every source with multiple sinks generates a Split node.

2. Every sink with multiple sources generates a Merge node.

This scheme is also known as *slave-side arbitration* [34] and it has the property that competition for network bandwidth occurs only at the sink, since there exists a dedicated physical path for each source-to-sink logical path. This is the scheme used by Qsys interconnect, with different arbitration/distribution primitives in place of GENIE's Split and Merge nodes, and is the only available option.

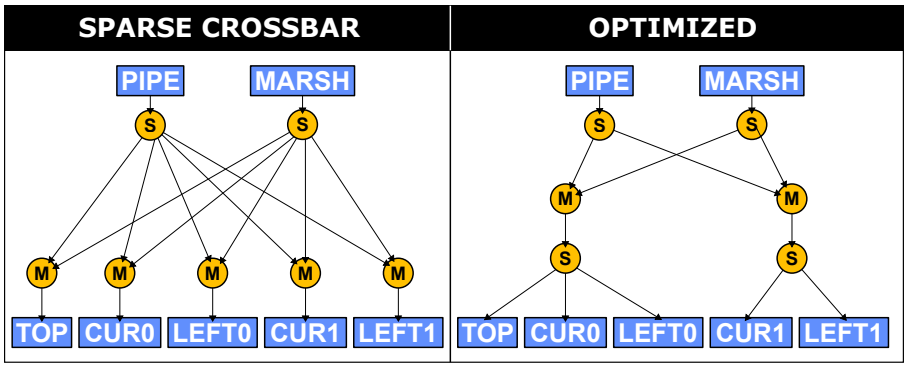

Figure 6.5: Sparse Crossbar (default) and Optimized (application-specific) topology implementations for Write Requests. The networks are built from (S)plit and (M)erge nodes.

The Pipeline and Marshaller are guaranteed to never compete for Cache access within the CE, and this can be taken advantage of to create the area-optimized topology for Cache Write Request links shown in Figure 6.5. The Top cache, although not itself double-buffered, logically belongs to buffer set 0, as it holds the top-most block in the column – the buffer index that holds a particular block in a column alternates every row and begins at 0. This is an application-specific nuance that allows such an interconnect optimization to be possible.

Compared to a crossbar topology, the custom topology reduces the number of Merge nodes from five to two without creating any contention points in the network. In both topologies, the Merge nodes are of the simplified conflict-free type (Section 4.2.3), as the logical transmissions through them have been explicitly marked as temporally exclusive in the user's input specification. Even so, write requests are 268 bits wide, and the removal of three of them still represents non-trivial savings, which will be quantified in Section 6.4. The additional two Split nodes incur minimal overhead as their cost is independent of payload width.

This fine-tuning of topology design would normally only be possible with hand-crafted Verilog, as is the case with our manual CE variant. GENIE provides a much simpler alternative via manual topology specification while *re-using the same logical connectivity specification*, thereby allowing fine-grained design optimization and topology exploration without giving up the convenience of automation.

6.4 Results

In this section, we quantitatively compare the three Compute Element variants described in Section 6.3.3 in order to judge the efficacy (and ease of use) of GENIE in generating fine-grained interconnect. Automation should increase productivity and make life easier for the designer. The implementation issues discussed in Section 6.3.4 give a qualitative view of the designer effort required. In this section we measure the amount of source code (and tool specification code) line counts as a first-order quantitative approximation of the difficulty of creating each CE variant. Automation should also strive to produce a high-quality interconnect implementation. We obtain the area and F_{max} of each variant after being synthesized, placed, and routed on an FPGA. After comparing the three CE implementations, we also measure the effects of the different interconnect optimizations that GENIE offers and how much each of them contributes to the GENIE CE variant's comparisons against the other two variants.

6.4.1 Methodology

Among the three CE variants (Manual, GENIE, Qsys), the GENIE and Qsys variants each have several configurations of interconnect that we will test and compare. Four configurations of the GENIE variant were created, differing in the combinations of two interconnect optimizations that can be selectively turned on or off for the synthesis flow. The first of these optimizations is enabling the use of the manually-specified application-optimized topology for the CE's write request network instead of the default sparse crossbar topology (both options are shown in Figure 6.5). The second optimization is to allow GENIE to infer conflict-free Merge nodes within the interconnect that implements accesses to the double-buffered caches. This is normally on by default, but can be explicitly disabled as a flow option. The resulting four GENIE variant configurations are:

- **GENIE_BOTH:** Application-optimized topology for cache writes, conflict-free Merge nodes are permitted.
- **GENIE_TOPO:** Application-optimized topology for cache writes, standard Merge nodes used throughout.
- **GENIE_MUTEX:** Sparse crossbar topology for cache writes, conflict-free Merge nodes are permitted.
- **GENIE_NONE:** Sparse crossbar topology for cache writes, standard Merge nodes used throughout.

These four configurations will be compared against each other in Section 6.4.4. In the comparison between the three CE variants (Manual, GENIE, Qsys), the GENIE_BOTH configuration will be used.

For the Qsys variant, two configurations will be tested. They differ in their handling of the single-cycle stalls generated by the Qsys interconnect and experienced by the Pipeline when it issues read requests to either of the Cur Caches. The two possibilities were described in detail in Section 6.3.4:

- **QSYS_FIFO:** The Pipeline uses a memory-backed FIFO to absorb the stall generated by the Qsys when a stream of cache read requests is issued. This is a more general and technically correct solution, requires more complex hardware, but the hardware in question can be an off-the-shelf vendor-provided FIFO implementation.
- **QSYS_REG:** The Pipeline uses a custom purpose-built circuit, backed by registers, that implements a 2-deep FIFO capable of absorbing exactly one stall cycle generated by the Qsys interconnect. This option only became available after detailed manual examination of simulations. It requires less complex hardware, but requires additional HDL code to be written.

Qsys offers some limited parameterization of its interconnect generation flow. One important option changes the amount of pipelining inserted into the interconnect, and has four possible settings: 1, 2, 3, or 4 registers. We used the maximum setting of 4 registers for both QSYS_FIFO and QSYS_REG.

6.4.2 Source Code Line Count

Our first set of results investigates the number of lines of source code (including scripting input lines for the two tools) required to create both the functional modules and interconnect for each CE variant.

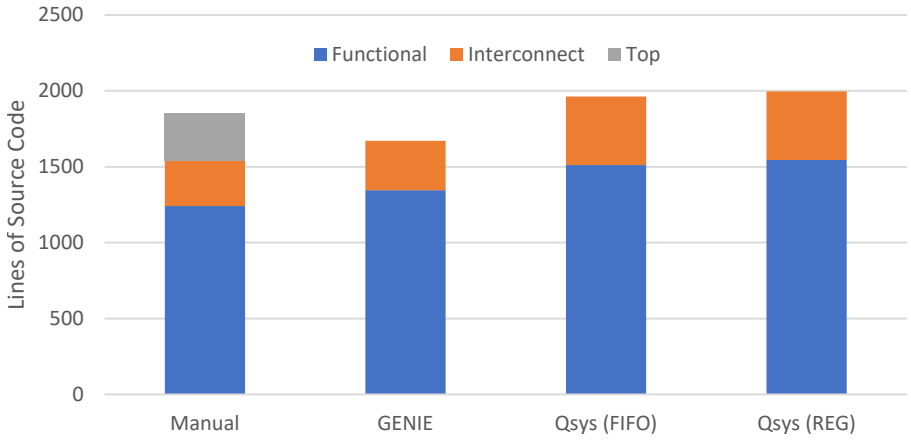

Figure 6.6: CE Code Line Count

Line counts were obtained using the CLOC [2] tool, which ignores comments and blank lines. For the interconnect portion, we are interested in the size of the specification directly written by the designer. In the Qsys and GENIE variants, this would be the size of the scripts (written in TCL and Lua, respectively) that are given as input to the tools to describe the system's communicating components and logical connectivity. The manual variant's interconnect is written in Verilog, as are the functional modules in all three variants. Table 6.1 compares the source code line counts between the three variants, divided into interconnect and functional module categories.

Table 6.1: CE Code Line Count

Variant	I/C + TOP	FUNC	TOTAL
Manual	612	1242	1854
GENIE	326	1346	1672
Qsys (FIFO)	453	1511	1964
Qsys (REG)	453	1544	1997
GENIE vs. Manual	-47%	+8.4%	-9.8%
GENIE vs. Qsys (FIFO)	-19%	-11%	-15%

Note that the manual variant's 612 lines of interconnect source code include 316 lines solely dedicated to the top-level Verilog module which instantiates all the other modules; this is referred to as "TOP" in the table heading. This glue code does not specify any true functionality, yet comprises a large portion of the source code base. Figure 6.6 gives it its own category to provide a better comparison of "real" interconnect specification size, which is 296 lines for the manual variant. Nevertheless, using either system integration tool spares the designer from having to manually write the top-level instantiation code, so we include it together with interconnect in the "Interconnect + Top" category in Table 6.1.

The design of the functional modules is also affected by the choice of interconnect synthesis tool, in order to be compatible with protocols or mitigate lack of features, as described in Section 6.3.4. The GENIE variant requires 8.4% more functional code than the manual variant, with minor architectural changes to the Cache modules. It also required 11% less functional code than the smaller Qsys variant (the smaller, FIFO-based version), which required the more significant changes described in Section 6.3.4.

This highlights an important qualitative component of design effort that is not completely captured

by line counts alone. The difference between the two Qsys variants is only 33 lines of code out of almost 2000. However, as described in Section 6.3.4, a significant amount of simulation, analysis, and design effort (which are also likely not transferable to other designs) was required on the part of the designer to write those 33 extra lines. Without this extra effort, yielding the Qsys FIFO variant (which still requires a relatively intrusive modification to the Pipeline module) achieves a poor clock frequency compared to GENIE, as will be shown in the next set of results.

If a designer were to create the Compute Element with GENIE in mind from the very beginning, they would need to write 9.8% less total source code than with no automation at all, with an even greater reduction of 47% if considering just the interconnect and top module. The respective savings over Qsys are 11% (total) and 19% (interconnect only), using the more optimistic comparison against the FIFO-based configuration. Implementing the two-register stall handler for the Qsys variant requires 33 extra lines of code, and much more time looking at simulation output.

The reduction in source code line count versus entirely manually-written Verilog is small, but even if it were zero, there would still be other benefits to automating the creation of fine-grained interconnect: the reduction of design, testing, and debugging time. As we will show next, GENIE also produces interconnect that is on par quantitatively, in terms of frequency and area, with the manually-written Verilog variant.

6.4.3 Area and Clock Frequency

Next, we measure the achieved clock frequencies of each variant. These are measured for the entire synthesized CE, not just the interconnect in isolation. Synthesis of each variant was performed using Intel Quartus Prime Pro version 17.0, targeting a large Arria 10 `10AX115N2F45E1SG` device, with the expectation of low congestion and device utilization. All top-level non-clock inputs and outputs terminate at Virtual IOs (dead-end LUTs) instead of real device pins. Both clock domains in the design were over-constrained to 1 GHz, and frequency results were geometrically averaged over six random seeds. The raw, uncapped frequencies reported by the TimeQuest timing analyzer were used, rather than the "Restricted F_{max}" which is limited by the minimum pulse width of the device.

Table 6.2: Compute Element Clock Frequency

Variant	Compute Clock	System Clock
Manual (MHz)	493	641
GENIE (MHz)	476	619
Qsys (FIFO) (MHz)	291	512
Qsys (REG) (MHz)	460	533
GENIE vs. Manual	-3.4%	-3.4%
GENIE vs. Qsys (FIFO)	+64%	+21%
GENIE vs. Qsys (REG)	+3.5%	+16%

Table 6.2 shows the achieved frequencies for both clock domains for each variant, and a relative comparison of GENIE against the other two variants. Against the manual variant, the GENIE-generated CE achieves clock frequencies that are only 3.4% worse. In both cases, the critical paths for the Compute clock are in the floating-point divider that is part of the Pipeline component. This is ideal, as it signals that the interconnect is not the bottleneck of the design. Clock crossing circuitry that enters a distributed

RAM based FIFO is the location of the critical paths for the System clocks of both the manual and GENIE variants.

The comparison of GENIE against Qsys differs significantly between the two possible configurations of the Qsys CE Pipeline, which uses the Compute clock domain. The critical path within the FIFO-based Qsys variant is within the FIFO, achieving a clock frequency of only 291 MHz – GENIE achieves 64% higher. This is not the fault of the interconnect, as the FIFO lays within the CE's compute modules. For a more fair comparison, the register-based Qsys variant's critical path is once again within the floating-point divider, which is the same as in the GENIE and manual variants. Here, GENIE only has a 3.5% advantage in Compute clock frequency. This floating-point divider is fed directly from the interconnect, which may explain why a change in interconnect implementation is changing the F_{max} despite the critical path appearing within a functional module.

For another interesting comparison, it is possible to reduce the number of registers that Qsys inserts into its interconnect from the maximum of 4 (which we used) to the default value of 2. In this case, the critical path *is* within the Qsys interconnect, achieving a clock frequency of just 358 MHz, even in the 'fairer' register-based version. However, even though Qsys is capable of achieving similar Compute clock frequencies, we will see (below) that it requires significantly more resources to do so. In the System clock domain, GENIE outperforms the FIFO- and register-based Qsys configurations by 21% and 16% respectively. The difference could be attributable to noise, as the System clock side does not change between the two configurations.

Table 6.3: Compute Element Area Usage

Variant	**ALM**	**M20K**
Manual	2521	40
GENIE	2508	40
Qsys (FIFO)	8110	108
Qsys (REG)	7987	101
GENIE vs. Manual	-0.5%	0%
GENIE vs. Qsys (FIFO)	-69%	-63%
GENIE vs. Qsys (REG)	-69%	-60%

Table 6.3 provides the area usage of the three variants, in terms of Arria 10 Adaptive Logic Modules (representing logic, registers, and distributed memory) and M20K memory blocks. All variants also use 8 DSP (hard multiplier) blocks in addition to what is shown.

The GENIE-generated CE is on par with the manual variant in terms of area. Slight differences in logic and register usage exist, but not enough to significantly change the observed result after packing into ALMs is performed. These differences are mainly attributable to where multiplexers and registers are placed in the interconnect, and the number of clock crossing FIFOs.

The Qsys interconnect contains an over-abundance of clock-crossing FIFOs (as discussed in Section 6.3.4), as well as additional FIFOs used to buffer cache read data. The increased number of FIFOs, and the fact that the GENIE and manual variants use distributed memory instead of M20Ks for their FIFOs, explains the high observed M20K usage for Qsys. In terms of logic, registers, and distributed RAM, GENIE uses 69% fewer ALMs than either Qsys configuration. Predictably, the FIFO-based Qsys configuration uses seven more memory blocks than the smaller and less complex register-based configuration.

Table 6.4: Clock Frequency, Area, and Source Code Effects of GENIE Optimizations

Configuration	Compute (MHz)	System (MHz)	Area (ALMs)	Code (Lines)
NONE	411	629	3157	292
TOPO	459	651	2757	302
MUTEX	471	614	2902	316
BOTH	476	619	2508	326
TOPO vs NONE	+12%	+3.5%	-13%	+24
MUTEX vs NONE	+14%	-2.4%	-8%	+10
BOTH vs NONE	+16%	-1.6%	-21%	+34

Recall that both Qsys configurations are created by instructing Qsys to use the maximum amount of interconnect pipelining (4 stages). If we reduce this setting to the default of 2, the ALM usage is significantly reduced, to 4437 and 4373 for the FIFO- and register-based configurations, respectively. These are still much larger than GENIE's CE.

6.4.4 Effects of Application-Specific Interconnect Optimizations

Up to this point, all GENIE results have used the GENIE_BOTH configuration which enables both the custom cache write topology and conflict-free merge node optimizations. Here, we will analyze in detail how much each optimization, in isolation, contributes to GENIE's clock frequency, area, and source code line count results. Table 6.4 presents the clock frequency, area, and source code line counts of four GENIE-generated Compute Elements.

The two optimizations being examined mainly affect the Merge nodes within the interconnect: the optimized topology (GENIE_TOPO) reduces the total number of Merge nodes in the Cache Write network domain, while the no-conflict optimization (GENIE_MUTEX) simplifies the remaining ones. We see a total design area reduction caused by each optimization separately: 13% fewer ALMs due to the optimized topology, and 8% fewer ALMs resulting from the no-conflict Merge nodes. The total result is additive, reducing the CE area by 21% when both optimizations are enabled. The Cache Write network domain has a 268 bit wide data path, and reductions and simplifications of Merge nodes in this part of the interconnect have great effect.

In terms of clock frequency, only the Compute clock domain should be affected. Any effects on the System clock can be attributed to noise, and serve as a comparison for the significance of the Compute clock results. The replacement of fully-functional Merge nodes with simpler, no-conflict Merge nodes yields a slightly better clock frequency improvement (GENIE_MUTEX, 14%) compared to reducing the number of fully-functional merge nodes (GENIE_TOPO, 12%). Together, both optimizations yield a 16% overall clock frequency improvement over the baseline.

The two optimizations studied here require extra specification code to be written by the designer, totaling 34 lines of Lua code. In Chapter 9, we will introduce an algorithm that is capable of *automatically* creating the optimized topology of the GENIE_TOPO configuration given only the transmission mutual exclusivity hints used in GENIE_MUTEX.

6.5 Conclusion

We have shown GENIE's applicability in a fine-granularity design space that has been neglected by existing tools. Using a realistic design example, we demonstrated significant clock frequency and area improvements over a commercial system-building tool, and comparable results to hand-designed interconnect. Qualitative differences in ease-of-use were also examined, related to the effort required by the designer to interface functional modules with the interconnect. Due to a mandatory requirement for supporting stalls generated by the interconnect, the Qsys version of the system required unwelcome changes to the design that either degraded clock frequency and area efficiency or required significant simulation effort to mitigate. GENIE did not require such changes. Quantitatively, GENIE required 9.8% fewer lines of total source code to describe the complete system, versus a hand-made implementation. This figure rises to 47% if considering only the code to design the interconnect. The cost of this productivity gain was a 3.5% degradation in clock frequencies, and less than a percent difference in ALM usage. This demonstrates that the automation and ease of use provided by the tool, our primary goal, does not detract from the interconnect's performance in a frugal fine-granularity design context. Ultimately, our crude measurement of number of lines of source code neglects the fact that not all source code is equally difficult to write – high-level scripting code requires a different amount of design and debugging effort than HDL.

Against Qsys, GENIE achieved clock frequency gains of 64% and 21% in the Compute Element's two clock domains, and a 69%/60% reduction in ALM usage and RAM usage respectively. These gains demonstrate the efficacy of GENIE's automatic clock crossing insertion algorithm and lightweight Split/Merge interconnect microarchitecture in a fine-granularity design.

We examined two of GENIE's optimizations that depended heavily on exploiting application-specific communication patterns: mutually-exclusive sharing (which creates simplified Merge nodes), and customizable topologies (which can reduce the number of expensive Merge nodes). Our single design example was able to take advantage of both of them, together providing a 16 % clock frequency improvement and a 21 % reduction in area. These optimizations took advantage of double-buffering, which is a common enough design technique and communication pattern that we foresee these optimizations being useful in other applications as well.

The next chapter will focus on coarse-grained interconnect synthesis while re-using the CE design studied here as a sub-module.

Chapter 7

Coarse-Grained Design Exploration

After demonstrating GENIE's fine-grained interconnect synthesis capabilities in Chapter 6, we now examine its application to larger, coarser-grained systems in this chapter. This is the domain of traditional interconnect synthesis tools [7, 69], and Network on Chip architectures [36]. For our benchmark application, we use the full LU Decomposition Engine design previously described in Chapter 5.1. It consists of *many* instances of the Compute Element sub-system that the last chapter focused on in detail, combined with off-chip memory controllers and a central control unit. The communication between these components was designed to be latency-insensitive and is thus more amenable to existing communication protocols.

This chapter is based on our paper from ICFPT 2015 titled *Automatic FPGA System and Interconnect Construction with Multicast and Customizable Topology* [63]. Much of the requisite (re)introduction to GENIE, its features and synthesis flow, and the LU application, will not be necessary, as those topics have been covered in previous chapters. Instead, the focus will be on exploring the design space of the LU application using GENIE's automated interconnect and system-building capabilities. As we have done in the previous chapter, we will be investigating how enabling and tuning GENIE flow parameters affect performance and area usage. We will also be comparing the LU application built using GENIE's interconnect versus two other versions, built with Intel's Qsys tool [7] and the CONNECT [58] packet-switched FPGA Network-on-Chip interconnect. This will allow us to evaluate GENIE's ability to build large systems in the same coarse-grained design context that existing tools have been designed to service. The comparison to CONNECT is new, and was not in the original paper.

7.1 System Design

The LU decomposition engine's high-level design and operation are described in Section 5.1. In this section, we provide additional implementation details that are both common among, and specific to, the three different interconnect methods – GENIE, Qsys, and CONNECT – being compared.

7.1.1 Common Implementation Details

The LU application contains one or more Compute Elements (CEs). In this chapter, a common CE design will be used irrespective of the coarse-grained interconnect that connects the CEs and other system components together. This CE design will use GENIE-generated fine-grained interconnect and

corresponds to the best-performing configuration examined previously in Chapter 6. When generating the CE, GENIE is configured to use an optimized write request network topology and conflict-free Merge nodes. It also uses a "maximum logic depth" setting of 5 levels, which ends up leaving the interconnect data paths unregistered everywhere except for the cache write request network, which receives one level of registers. Of the two clock domains present in the LU application, the Compute Clock is internal to the CEs. Therefore, using a common CE design should *ideally* yield minimal variation in the Compute Clock domain frequency when varying the outer coarse-grained system interconnect. We will show that this is not the case, and offer a hypothesis related to the nature of FPGA placement and routing.

In addition to CEs, the LU system also contains a variable number of memory controllers. For internal design reasons, this number M must be a power of two, as must the number of CEs (N). Increasing M increases the available memory bandwidth and permits the addition of more CEs to the system without experiencing a plateau in performance. Each of the M memory controller nodes does not directly communicate with off-chip memory, but rather buffers and converts the large read and write requests of 64×64 matrix blocks to bursts of appropriate size that can be consumed by actual, vendor and device-specific off-chip memory controllers, **which we do not instantiate in our design**. Instead, each of our M nodes terminates in a set of Virtual I/O pins (implemented as logic cells) that carry the request/reply signals that would normally be forwarded to a "real" off-chip memory controller provided by Intel in the Quartus IP library. The purpose of this design choice is to allow us to experiment with values of M that would not be feasible to synthesize on an actual FPGA due to pin and bank placement constraints (and fixed number of hardened memory interface PHYs available). However, such designs may still be physically realizable in future work by using external memory solutions that communicate over the more plentiful high-speed serial interfaces on FPGAs, such as FBDIMM[35] and Hybrid Memory Cube[52].

The CEs, memory controller nodes, and central control unit are designed to communicate over a GENIE-generated interconnect fabric. In order to use a different system-level fabric (such as Qsys or CONNECT), protocol conversion modules are inserted at each system-facing interface. The design and area overhead of each type of interface will be discussed in the respective sections.

7.1.2 GENIE Interconnect

The GENIE-built version of the LU application will serve as the baseline for comparing all others. However, before doing so, we would also like to investigate the effects of a variety of parameters and features *within* the design space of GENIE-generated systems.

The most important of these is the performance improvement yielded by the "Left Block broadcast" optimization that is specific to the LU application, in which certain matrix blocks that are requested by all running CEs can be read just *once* from a memory controller and broadcast to the CEs, instead of being read redundantly multiple times. More details were previously described in Section 5.1.2. GENIE is the only one of the three interconnects being compared that is capable of implementing the multicast transmissions necessary to make this optimization possible. We will be comparing the total application runtime of the system when using this optimization, versus having this optimization forcibly turned off, in which CEs request all matrix blocks as unicast transmissions directly from memory controllers, resulting in redundant reads. This comparison will be made across multiple values of N and M, as it is expected the benefits of the optimization will only be seen when memory bandwidth demand exceeds supply. The version of the LU system with the Left Block broadcast optimization forcibly turned off

also provides a fairer baseline against which to compare the application cycle counts for the other two, non-multicast-capable, versions of the system.

Another adjustable parameter for GENIE is the maximum interconnect logic depth. This controls the amount of registers automatically inserted into the interconnect for pipelining. This parameter D indicates to GENIE that elastic buffer primitives should be inserted no more than D consecutive logic levels (in LUTs) apart. It includes the combinational logic depth "looking in" to user's functional modules through Routed Streaming interfaces, which the user can specify with an additional parameter that defaults to 0, indicating that the interface is immediately registered. Since the CEs that comprise the LU system are universally built with GENIE in this chapter, using GENIE to build both the coarse-grained LU system and fine-grained CE contents presents opportunities for cross-hierarchy-boundary optimization in this regard. Specifically for logic depth, it should be possible for the GENIE-generated CE to automatically annotate its system-facing interfaces with the correct logic depth looking into the CE. Unfortunately, this is not yet the case, and is future work. For this reason, the logic depths of the CE interfaces have been manually annotated to their correct values to mimic this future functionality. GENIE's default maximum logic depth is 5 levels, but we will also sweep other values to produce a performance/area trade-off, which will better inform GENIE's comparison against the other two interconnect types later. The details of the automatic pipelining mechanism will be explained in Chapter 8.

7.1.3 Qsys Interconnect

In order to build the LU system using Qsys, the interfaces of the CEs, memory controllers, and control node must be made to use the Avalon-MM [34] protocol. Instances of a protocol conversion module are inserted to make this possible, specially made for the LU application's five types of transmissions. Logically, the largest change is mapping integer destination addresses to byte addresses within a global memory map. For communications that were not memory-like to begin with, such as the Go messages from the control node to each CE, this is a minor inconvenience. Expressing the Done messages that are sent from all CEs to the control node as memory writes (to a single address) is necessary to receive the benefits of automatically-inserted interconnect that handles arbitration and contention of those messages.

The three memory-like transmission types (read request, read reply, write request) map very naturally to Avalon 'read' and 'write' transactions, as this is what the protocol was designed for. Read requests and replies are not considered as two distinct types of transactions by Qsys, which slightly simplifies the interfaces of the memory controllers compared to the GENIE and CONNECT versions. Specifically, read replies do not experience backpressure – the Qsys interconnect contains memory buffers that hold the replies, and only grants read requests access to the memory controller if there exists sufficient space in the reply buffers. As a result, memory controllers do not have to receive backpressure at the reply side, nor do they need to remember the destination for each read reply, as that is kept track of within the interconnect. However, these buffers and additional interconnect complexity will incur an area cost, as we will show in our results. The convenience of not having separate request and reply transmissions also makes the Left Block broadcast optimization impossible with Qsys.

Qsys provides some parameterization of its memory-mapped interconnect. The "Limit interconnect pipeline stages to:" setting permits Qsys to insert between 0 and 4 (inclusive) pipeline stages into the interconnect data path. This parameter, which we will shorten to R, acts like GENIE's D setting in that it enables a trade-off between performance and area usage. Unlike D, increasing R increases the number of registers. The four possible nonzero values of R correspond to the four locations within the Qsys

interconnect fabric that the designers decided that registers are best placed. A more detailed control of (even additional, beyond four) register placement is possible manually within the GUI, in a way that could be specific to each application, but we did not explore these additional settings. Qsys' default value for R is 2, but we also consider using the maximum value of 4 when benchmarking Qsys against GENIE and CONNECT.

7.1.4 CONNECT Interconnect

The CONfigurable NEtwork Creation Tool (CONNECT) [58], which was mentioned in Section 2.2.1 is a packet-switched NoC architecture designed with FPGAs in mind. Unlike GENIE and Qsys, it only generates the interconnect itself and does not create the full system that includes the instances of the functional modules. The tool itself is available publicly as a web-based interface [51] that permits the user to configure the network's parameters, after which it sends an email containing synthesizable Verilog source code. CONNECT's architecture uses monolithic routers that contain input and output buffers and support virtual channels. Figure 7.1 illustrates the router architecture.

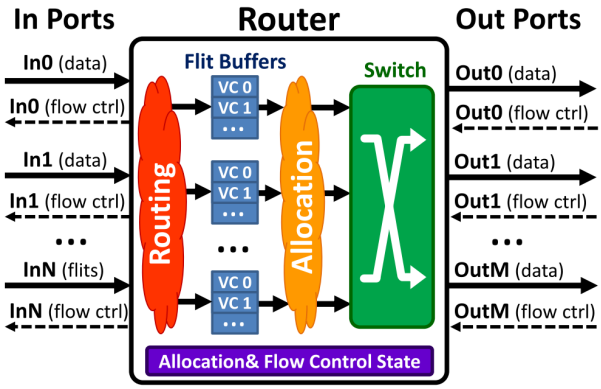

Figure 7.1: CONNECT's Router Architecture, sourced from [58]

Unlike the GENIE and Qsys versions of the LU system, which each create five separate interconnect domains for the five types of traffic, the CONNECT version contains a *single* network that carries all possible transmission types. This is done to respect the use case that CONNECT was designed for. As a result, the protocol conversion modules in front of each CE contain additional multiplexing and arbitration to create a single output interface from three previously-separate interfaces (done messages, write requests, read requests). Neither the memory controllers nor the control node require such multiplexing, since they only send one type of transmission. In order for the network to carry all possible transmission types, we introduced a "transmission type" field within our data payload, which was already made wide enough to contain the largest of the transmission types (272 bits, used for write requests and read replies), bringing the total data width to 275 bits. Additionally, CONNECT requires fields within each flit to indicate the destination address, virtual channel number, and an "end-of-packet" flag, yielding a final flit width of 283 bits. The entirety of a data word is contained inside each flit to stay true to CONNECT's design philosophy of utilizing many parallel wires for links. Splitting each large data word into multiple flits would penalize maximum throughput in this minimally-pipelined router architecture.

The network was configured with $(N+M+1)$ bidirectional endpoints, representing the CEs, memory controllers, and the control node. Every endpoint is attached to a router, and the routers can be arranged

in a multitude of ways to create different network topologies. Our goal is mainly to evaluate GENIE, so evaluating many topologies for our CONNECT network would represent unnecessary effort. However, in order to have a fair comparison against GENIE, the topology that we do choose for the CONNECT network should be a good representative of that architecture's capabilities and intended use. To that end, we will evaluate the area and performance of two different topologies that represent opposing points of the design space:

- **Bidirectional Ring:** There exists one router per node and the routers are connected with two rings carrying data in opposing directions. Each memory controller node is placed after a group of N/M CE nodes in the ring. Each group of N/M CEs are the CEs that will mainly communicate with the neighbouring memory controller node, effectively partitioning the ring into locally-communicating regions that better utilize available bandwidth. This design point maximizes the amount of routers, and minimizes the amount of network ports (and buffers) and complexity of arbitration internal to the router.
- **Hub:** Shown in Figure 7.2, this topology contains one router per memory controller node, but no routers associated with the CEs or control node. The N/M CEs that primarily communicate with each memory controller are attached to that same router, allowing efficient local communication. These M routers are connected via a bidirectional ring, whose network links are only used for some phases of the application computation loop. The control node is arbitrarily attached to one of the routers. This topology minimizes the number routers, but each router has many ports.

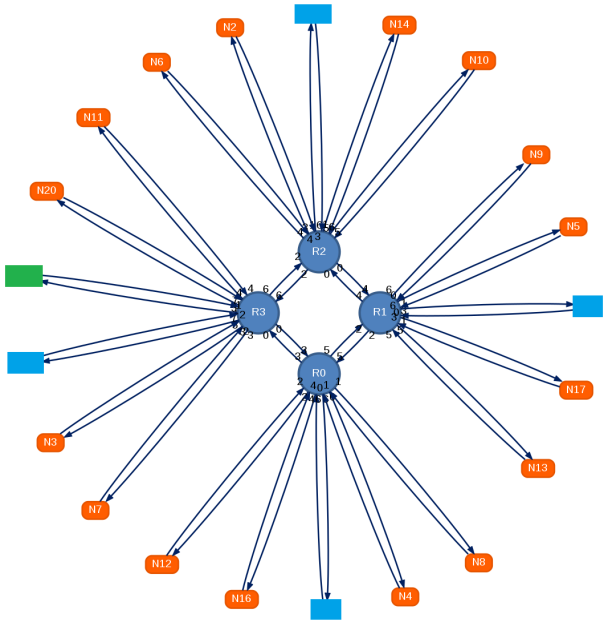

Figure 7.2: CONNECT "Hub" topology for $N = 16$ and $M = 4$. Orange nodes are CEs, blue rectangles are memory controllers, and the green rectangle is the control node. Only 4 routers are used, shown in the center.

In order to avoid deadlocks and improve performance, especially in the ring topology, we will configure CONNECT with two virtual channels: VC 1 will carry read replies, while VC 0 (which has higher priority) will be used for all the other types of traffic. As with the Qsys interconnect, CONNECT does

not support multicast transmissions. This makes the Left Block broadcast optimization impossible to implement.

The set of parameters used to generate our CONNECT networks is shown in Figure 7.3. The topologies explored were Double Ring as well as our custom hub-based one. The number of endpoints equals $N + M + 1$ for any given configuration. Two virtual channels were used to separate requests and responses. "Peek Flow Control" was selected as an alternative to the default credit-based flow control, and provides a similar valid/ready handshaking scheme as used by the protocols of GENIE and Qsys. The advanced settings were left to their defaults except for "Use Virtual Links", which prevents fragmentation of long multi-cycle transmissions at their destinations, also matching the behavior of GENIE and Qsys.

Parameter	Value
Network Topology	
Topology	Double Ring
Number of Endpoints	21
Network and Router Options	
Router Type	Virtual Channel (VC)
Number of VCs	2
Flow Control Type	Peek Flow Control
Flit Data Width	275
Advanced Options (click to expand)	
Flit Buffer Depth	8
Allocator	Separable Input-First Round-Robin
Use Virtual Links	☑
Debug Symbols	None

Figure 7.3: Settings used to generate the CONNECT network. The topology and number of endpoints will vary. Here, they are set up for $N = 16$, $M = 4$, and a bidirectional ring topology.

7.2 Results

In this section, we will measure the clock frequency, area usage, and simulated application runtime of the various design points within the design spaces previously described in Section 7.1. Our goal here is to explore each of the three interconnect design spaces in isolation first, before performing a head-to-head comparison in the next section. The large number of parameters merits a separate examination and commentary on the performance of each interconnect architecture and how they scale with the size of the application. First, we will describe our experimental methodology.

7.2.1 Experimental Methodology

There are two classes of results we are collecting: synthesis for area and clock frequency, and simulation for application run time. For synthesis, we use Quartus Prime 17.0 Pro Edition and target an Arria 10 `10AX115N2F45E1SG` device. This is the largest device, at the fastest speed grade, available for this

Intel device family. Clock frequencies are measured for *both* clock domains, using a geometric average over four placement seeds. When measuring area, we separately include combinational lookup table and register usage. Distributed memory usage is only reported for the CONNECT interconnect, and it is not used by any of the other interconnect variants. No block memory or DSP blocks are used by any interconnect variant, so these are not reported at all. We also include the Adaptive Logic Module (ALM) count, which better represents total packed area utilization of combinational logic, registers, distributed memory (where applicable). Area is measured for the interconnect only, while clock frequencies are reported for the entire system.

Simulations are carried out in ModelSim 10.4c using a testbench that simulates off-chip memory controllers. The controller model is simple: it adds 30 clock cycles of latency to each request (read or write) before carrying it out, simulating the row and column access overheads of DDR SDRAM. The incoming requests are delayed in a pipelined fashion rather than serially adding 30 cycles to each request.

The LU application processes an input matrix, and outputs a matrix of the same size. We generate a random square matrix whose dimensions are multiples of 64 element blocks. Correctness was verified against a software reference implementation, using spot checks on various sizes of matrix dimensions and application size (N and M parameters). This is a lengthy process, requiring up to a day per run. In order to expedite data collection, the LU application was configured for "Performance-only Simulation", in which no data is computed, but each Compute Element simply waits the correct number of cycles in order to simulate the computation that would have occurred. This reduces the required simulation time to approximately 20 minutes. Computation time is deterministic and not data dependent, which makes this simplification accurate. Only control and handshaking signals are properly simulated, and this is enough to measure application performance.

Even if the processed data is meaningless in this performance-only simulation mode, it is still crucial to choose an input matrix size that will exercise all available hardware and interconnect. For a system with N CEs, this number is $(N+1) \times (N+1)$ blocks. Recall that the outer loop of the LU algorithm (Algorithm 1) first performs a serial pass using CE0 on the first column, then utilizes as many CEs as necessary to process the remaining columns. This parallel computation pass will thus need N remaining columns of matrix data to utilize all N CEs. In general, matrices containing $(1+kN)$ blocks will exhibit this optimal use of CEs, on their first outer loop pass.

7.2.2 GENIE: Benefits of Multicast

One of the key advantages of GENIE over the other two types of interconnect is the Left Block broadcast optimization that is possible due to multicast transmissions. Here, we will measure the resulting performance improvement when it is enabled, versus when it is not. The goal of this optimization is to reduce the memory bandwidth utilized by CEs. Naturally, this only affects performance if memory (and interconnect) bandwidth is the bottleneck, rather than the computation. Thus, we expect to observe a performance improvement only for greater ratios of N to M. Additionally, since the CEs' compute pipelines operate on their own separate Compute Clock, the ratio of this clock frequency to the System Clock frequency (which governs the memory and interconnect) will also affect the memory bandwidth demands of the CEs.

In Table 7.1, we show the total application runtimes of LU systems of various sizes, built using GENIE, and with the Left Block broadcast optimization enabled ("BCAST") and forcibly disabled ("NO BCAST"). The simulations are run with a 400 MHz Compute Clock and two different frequencies

for the System Clock: 300 MHz and 200 MHz. These frequencies are representative of the range of achieved clock frequencies seen in the remainder of this chapter.

Table 7.1: Simulated runtimes in milliseconds and resulting speedups achieved from enabling Left Block broadcast in GENIE-generated LU systems of various sizes.

	400 MHz/300 MHz			400 MHz/200 MHz		
Size	**BCAST**	**NO BCAST**	**Speedup**	**BCAST**	**NO BCAST**	**Speedup**
N1 M1	991	991	1.000	993	993	1.000
N2 M1	520	520	1.000	522	521	0.999
N4 M1	284	285	1.003	285	286	1.004
N8 M1	166	167	1.006	167	168	1.010
N16M1	105	106	1.013	107	130	**1.210**
N32M1	76	94	**1.227**	93	122	**1.309**
N4 M2	284	285	1.004	285	286	1.006
N8 M2	165	166	1.008	166	168	1.011
N16M2	104	106	1.013	105	108	1.025
N32M2	71	79	**1.101**	72	95	**1.319**
N16M4	104	105	1.012	104	106	1.019
N32M4	71	73	1.026	71	86	**1.205**

We observe that runtime decreases as N is increased, as expected. Increasing M only further decreases runtime when memory bandwidth demand is high due to an initially large N/M ratio, for example, going from N32M1 → N32M2 → N32M4. The speedup due to enabling Left Block broadcast is also included in Table 7.1, and we can see that it too only helps runtime when memory bandwidth demand is high. Instances of significant reductions of 10% or more are bolded, and are more numerous when the memory and interconnect clock runs at 200 MHz rather than 300 MHz. Figure 7.4 plots just this speedup for both clock frequency configurations and illustrates these trends more clearly.

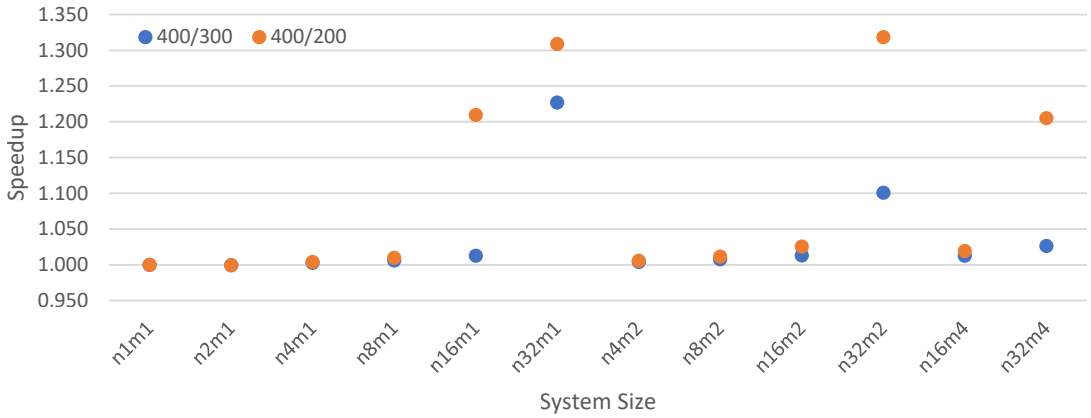

Figure 7.4: Speedup due to enabling Left Block broadcast for two clock frequency ratios across many system sizes.

The goal of this comparison was to provide a detailed real-world use case of how multicast-capable interconnect can benefit an application designer. In addition to improving performance, one can use it to reduce system complexity while maintaining performance: at a 300 MHz System Clock and with 32 compute elements and only one memory controller, the effect of enabling this multicast-powered application optimization yields a better run time (76 ms) than adding a second memory controller

(79 ms).

7.2.3 GENIE: Maximum Logic Depth

Here, we vary the amount of automatic pipelining performed by GENIE by varying the parameter D, which is the maximum allowable logic depth. Four different values of D are tested: 5, 4, 3, and 2, with decreasing values corresponding to more pipelining. This generates several frequency and area curves, presented in Tables 7.2/7.3 and Figures 7.5/7.6.

The results illustrate the ability to trade off performance for area. Decreasing D has the effect of increasing the System clock frequency, which is the clock domain used for the interconnect at this level of the application's design hierarchy. Figure 7.5 illustrates that the effect is more pronounced for systems of sufficiently large size, with 8 or more compute elements. A point of diminishing returns is reached as D continues to decrease, where the clock frequency no longer increases.

Area usage, however, continues to increase as D decreases, as shown by the ALM and register usage curves in Figure 7.6. The intent of this particular experiment is to choose "high-performance" and "low area" GENIE system variant to allow a fair comparison against Qsys, which is also able to vary the amount of interconnect pipelining. Based on the frequency and area results seen here, a reasonable "high-performance" value of D would be 3, as it provides similar performance to $D = 2$ but with lower area cost. The $D = 5$ and $D = 3$ parameterizations will be used in Section 7.2.6 for the final comparison against the other tools.

Table 7.2: Achieved System and Compute clock frequencies in MHz for GENIE-generated LU systems versus system size and maximum logic depth.

	D5		**D4**		**D3**		**D2**	
Size	**System**	**Compute**	**System**	**Compute**	**System**	**Compute**	**System**	**Compute**
N1 M1	360	474	364	498	371	490	368	501
N2 M1	377	468	365	426	383	433	376	448
N4 M1	364	461	375	452	385	459	369	458
N8 M1	321	430	377	455	369	460	376	451
N16M1	318	411	323	416	331	418	329	390
N32M1	219	391	274	411	276	393	270	387
N4 M2	378	457	377	459	370	457	378	458
N8 M2	344	435	369	456	361	434	382	460
N16M2	315	381	327	398	357	414	356	409
N32M2	270	401	292	392	302	385	319	391
N16M4	291	399	329	409	349	403	337	388
N32M4	258	393	300	390	308	397	305	399

7.2.4 Qsys: Interconnect Pipelining

Here, we collect similar results for Qsys-generated LU systems. In addition to varying the system size, we use two settings for the amount of interconnect pipelining provided by Qsys: two stages (the tool's default) and four stages (the maximum allowed value). Doing so will allow us to choose a fair comparison point against GENIE and CONNECT.

Table 7.4 and Figure 7.7 provide the achieved clock frequencies for the Qsys results, and Table 7.5 and Figure 7.8 provide the area consumption. As expected, using the higher level of interconnect pipelining

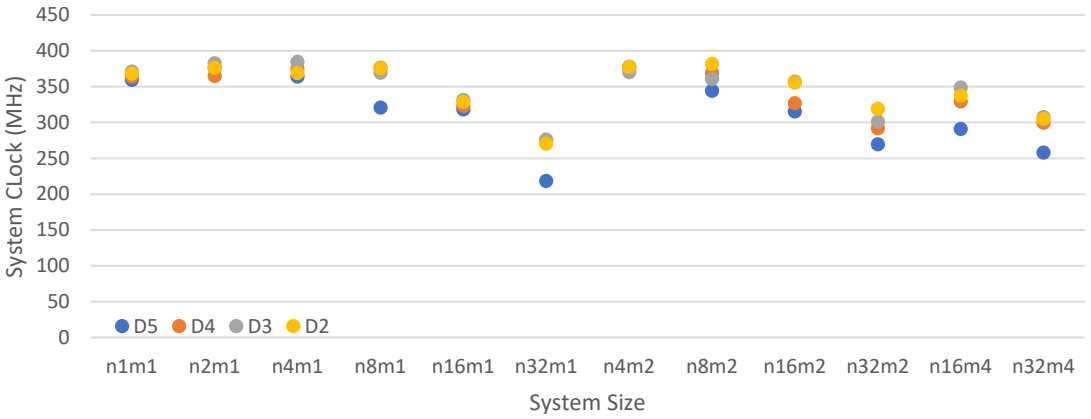

Figure 7.5: Achieved System Clock frequencies for GENIE-generated LU systems versus system size and maximum logic depth.

Table 7.3: Area usage (ALM, combinational LUTs, and registers) of GENIE-generated interconnect versus system size and maximum logic depth.

	D5			D4			D3			D2		
Size	**ALM**	**COMB**	**REG**	**ALM**	**COMB**	**REG**	**ALM**	**COMB**	**REG**	**ALM**	**COMB**	**REG**
N1 M1	229	29	875	233	29	873	428	31	1466	439	32	1504
N2 M1	356	338	922	390	347	1002	651	348	1941	1018	354	3051
N4 M1	606	412	1113	1729	445	4870	1786	440	5026	1952	467	5616
N8 M1	1152	1109	1484	3639	1185	10052	3883	1187	10677	3873	1189	10633
N16M1	3818	2054	9221	7033	2063	18813	7230	2118	19467	7309	2104	19519
N32M1	7365	4293	16749	14244	4515	37307	14434	4504	37876	14425	4517	37905
N4 M2	1397	1004	3327	1578	1006	3459	2066	1575	5619	3205	1602	9053
N8 M2	2015	1527	3734	3200	1557	6914	4589	2374	11405	5736	2383	15407
N16M2	3548	2422	5655	6860	3901	14258	8753	4379	22609	9516	4385	24899
N32M2	7006	5135	10567	13595	6458	31626	15151	6844	40753	16206	6901	43755
N16M4	7717	4846	15589	10676	4839	26623	12478	5689	33481	14927	5796	39962
N32M4	9896	6145	17732	15897	10371	41819	20418	10366	55001	21332	10445	58076

($R = 4$ versus $R = 2$) achieves higher clock frequencies at the expense of more area, specifically in terms of register usage.

7.2.5 CONNECT: Topology

The only design parameter we will examine for the CONNECT-based interconnect is the network topology. We are interested in whether a double ring or hub-based (Figure 7.2) topology would make the better candidate for comparison against GENIE.

Using a single system size of $N = 16$ and $M = 4$, we compare the achieved system clock frequencies and interconnect area utilization of systems built using these two topologies. The systems were also simulated (using fixed Compute/System clock frequencies of 300/200 MHz respectively) to obtain the time required to process a 17×17-block matrix). The results are presented in Table 7.6. We observe that the double ring consumes 8.4 % more packed area but yields a two-fold increase in achieved interconnect frequency. While the double ring employs many more routers, each router is of lower cardinality than the four routers used in the hub topology, which is likely responsible for the difference observed in clock frequency. The simulated application runtime of the double ring based system is less than a percent slower than the hub-based system – the application is compute-bound and not utilizing the full

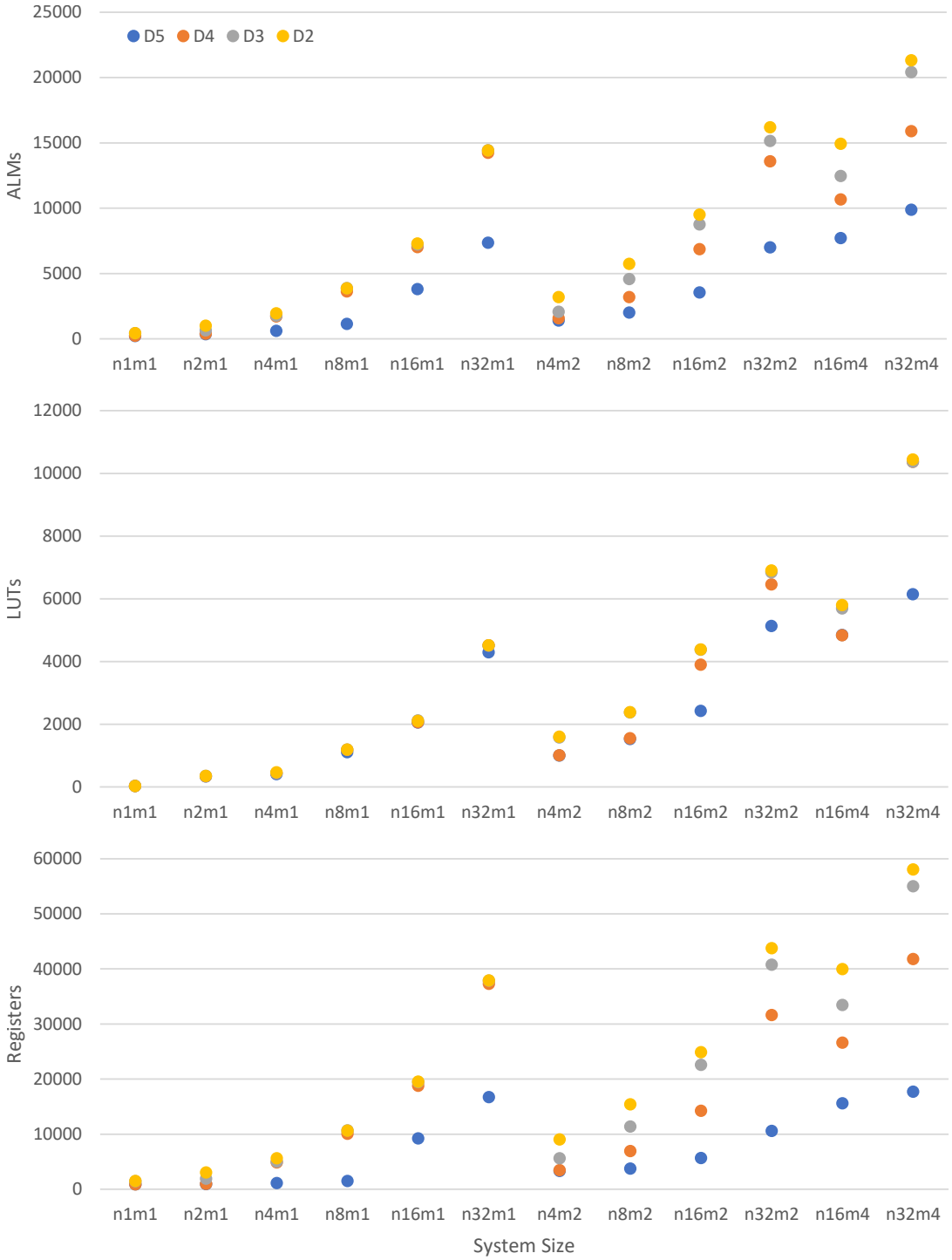

Figure 7.6: ALM, combinational LUT, and register usage for GENIE-generated interconnect versus system size and maximum logic depth.

Table 7.4: Achieved System and Compute Clock frequencies in MHz of Qsys-generated LU systems versus system size and number of interconnect pipeline stages(R).

	R2		R4	
Size	**System**	**Compute**	**System**	**Compute**
N1 M1	335	495	351	475
N2 M1	312	447	375	410
N4 M1	298	460	376	463
N8 M1	269	458	379	466
N16M1	254	417	328	408
N32M1	196	393	226	416
N4 M2	307	457	392	459
N8 M2	290	450	380	453
N16M2	251	410	355	384
N32M2	221	387	271	402
N16M4	286	412	344	402
N32M4	231	388	263	383

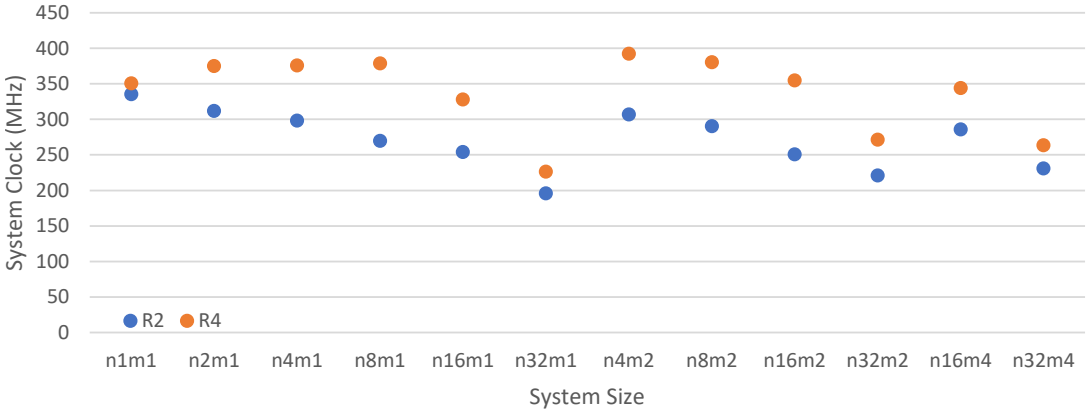

Figure 7.7: Achieved System Clock frequencies for Qsys-generated LU systems versus system size and number of interconnect pipeline stages(R).

throughput of the network. Meanwhile, the increased number of hops in the double ring network is likely responsible for the small observed runtime difference, as the increased latency would affect the serial portions of computation of the outer loop. Because the double ring topology has a significant clock frequency advantage over the hub topology for relatively little area cost, we will use it to represent CONNECT in the comparison against GENIE and Qsys.

7.2.6 GENIE vs. Qsys vs. CONNECT

Here, we select five versions of the LU system to perform a final comparison of clock frequency, area, and application execution time:

- **GENIE D3:** A high-performance variant of the GENIE interconnect that uses a maximum logic depth of 3.
- **GENIE D5:** An area-efficient variant of the GENIE interconnect that uses a maximum logic depth of 5.

Table 7.5: ALM, combinational LUT, and register usage of Qsys-generated interconnect versus system size and number of interconnect pipeline stages(R).

	R2			R4		
Size	**ALM**	**COMB**	**REG**	**ALM**	**COMB**	**REG**
N1 M1	37	62	53	36	60	49
N2 M1	346	685	274	1185	810	3189
N4 M1	746	1366	475	2125	1506	5425
N8 M1	1489	2403	767	4086	2772	9698
N16M1	2947	4791	1405	7806	5168	18290
N32M1	6341	10824	3079	15511	10203	35657
N4 M2	1486	2670	1921	3593	2838	9602
N8 M2	2951	5445	3609	6248	5582	16288
N16M2	5895	10355	7010	12026	10359	29971
N32M2	11806	20264	13767	23457	19955	57092
N16M4	13691	20040	7529	16840	16226	43323
N32M4	17510	30801	14555	32949	30397	81408

Table 7.6: Clock frequency, area, and application runtime results for two different topologies of the CONNECT variant of an $N = 16M = 4$ LU system

Topology	**System Clock**	**Compute Clock**	**ALM**	**LUT**	**REG**	**Mem. ALM**	**Runtime**
Double Ring	185 MHz	421 MHz	24084	24897	3957	9930	26.03 ms
Hub	92 MHz	395 MHz	22236	22155	2139	4230	25.99 ms
Double Ring vs. Hub	+ 101%	+ 6.6%	+ 8.3%	+ 12.4%	+ 85%	+ 135%	+ 0.2%

- **QSYS R4:** High-performance Qsys, 4 stages of interconnect pipelining.
- **QSYS R2:** Area-efficient Qsys, 2 stages of interconnect pipelining.
- **CONNECT:** Double ring topology CONNECT network.

The two versions of GENIE and Qsys represent performance-prioritized and area-prioritized corners of the design space. The performance-prioritized corners (GENIE D3 and QSYS R4) will compete for clock frequency, where the area-prioritized corners (GENIE D5 and QSYS R2) will compete for area utilization. The two Qsys variants (area-prioritized $R = 2$ and performance-prioritized $R = 4$) comprise a corresponding set of contenders for the Qsys interconnect.

Previously in Section 7.2.3, we determined that using a maximum logic depth of less than 3 when creating GENIE interconnect yields diminishing returns, hence our usage of $D = 3$ as the higher performance corner. Maximum logic depths of greater than 5 are possible, but as we will soon show, a setting of $D = 5$ for GENIE matches up with Qsys' corresponding area-prioritized corner in such a way that there exist both wins and losses in area in GENIE's favour.

Area, clock frequency, and application performance will be swept across eight different system sizes, ranging from $(N = 8, M = 1)$ to $(N = 32, M = 4)$. This is a slightly smaller range of sizes than have been used previously in this chapter, as system sizes of $N = 4$ and smaller are being ignored.

We will begin with the area comparison. Table 7.7 provides absolute area usages for the five variants in terms of packed area (ALMs) and pre-packed combinational LUTs (COMB) and registers (REG). Two additional tables re-interpret this data as relative comparisons. Table 7.8 compares the GENIE area-prioritized variant ($D = 5$) against the Qsys and CONNECT variant, showing how much smaller

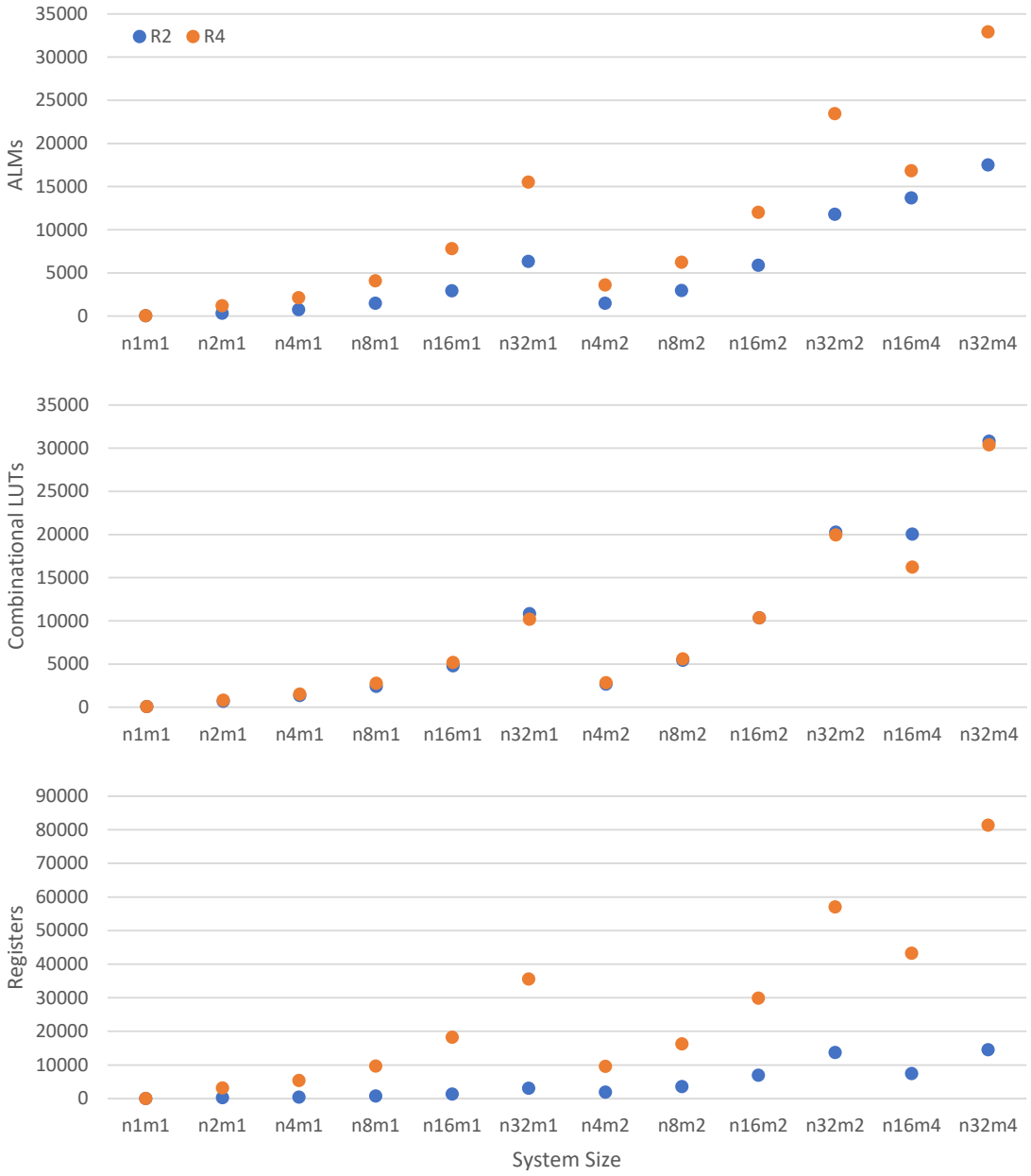

Figure 7.8: ALM, combinational LUT, and register usage for Qsys-generated interconnect versus system size and number of interconnect pipeline stages(R).

or larger, in percent, each GENIE system is relative to the corresponding Qsys or CONNECT system. Table 7.9 repeats this comparison for the GENIE performance-prioritized variant ($D = 3$). Figure 7.9 displays the absolute area comparison in bar chart form.

Comparing the area-prioritized interconnects, GENIE D5 consumes less packed area (ALMs) than Qsys R2 and CONNECT in most cases. The exceptions are the two systems in which there are a large number of compute elements ($N = 16$ and $N = 32$) and only one memory controller ($M = 1$). Here, GENIE consumes 16% and 30% more ALMs than Qsys, respectively, with the absolute differences being 900-1000 ALMs. With larger systems and more complex interconnect requirements due to additional memory controllers, GENIE provides an ever-increasing area savings, topping out at 43% at the largest

Table 7.7: Absolute ALM, combinational LUT, and register usage for GENIE, Qsys, and CONNECT-generated systems.

		N8M1	N16M1	N32M1	N8M2	N16M2	N32M2	N16M4	N32M4
GENIE D5	**ALM**	1152	3818	7365	2015	3548	7006	7717	9896
	COMB	1109	2054	4293	1527	2422	5135	4846	6145
	REG	1484	9221	16749	3734	5655	10567	15589	17732
GENIE D3	**ALM**	3883	7230	14434	4589	8753	15151	12478	20418
	COMB	1187	2118	4504	2374	4379	6844	5689	10366
	REG	10677	19467	37876	11405	22609	40753	33481	55001
QSYS R2	**ALM**	1489	2947	6341	2951	5895	11806	13691	17510
	COMB	2403	4791	10824	5445	10355	20264	20040	30801
	REG	767	1405	3079	3609	7010	13767	7529	14555
QSYS R4	**ALM**	4086	7806	15511	6248	12026	23457	16840	32949
	COMB	2772	5168	10203	5582	10359	19955	16226	30397
	REG	9698	18290	35657	16288	29971	57092	43323	81408
CONNECT	**ALM**	11164	20002	38662	12574	21705.2	39697	24084	42898
	COMB	11426	20626	39246	12901	22091	40672	24897	43485
	REG	1863	3311	6207	2073	3585	6609	3957	6981

system configuration ($N = 32, M = 4$). For the performance-prioritized corners, GENIE D3 consistently consumes fewer ALMs than Qsys R4, from 5% at the smallest system ($N = 1, M = 1$) to 38% at the largest.

CONNECT has the poorest area utilization out of all the five variants, and is already configured with a relatively area-efficient double ring topology. With the same number of network nodes, a topology with more bisection bandwidth such as a mesh would have more ports and links, further increasing the area. Even the high-performance GENIE D3 variant, which consumes more area than GENIE D5, yields a minimum 48% ALM savings versus CONNECT.

Several interesting observations arise when examining the usage of individual FPGA resource types by the different interconnects. In terms of combinational logic, GENIE consistently uses fewer look-up tables than the other two interconnect types, and CONNECT uses the most. However, GENIE is register-heavy, and CONNECT is the opposite. As we will see in the clock frequency comparison, this design strategy will hurt CONNECT's results there. Against Qsys, GENIE tends to use more registers when comparing the area-prioritized corners, and fewer registers when comparing the performance-optimized corners. In the former case, GENIE still wins in packed area more often than not. For example, for the $N = 16M = 4$ system, GENIE D5 uses over twice the number of registers as QSYS R2, yet still wins in packed area by 44%. If clock frequencies and (architectural) FPGA interconnect delays increase in the future, register-heavy designs may be a more favourable design strategy.

Next, we compare achieved clock frequencies. We are mainly concerned with the system clock frequency, as in the LU application, this clock domain drives the interconnect. However, the compute clock is also recorded, as it will be used to perform an accurate simulation during the runtime comparison. Table 7.10 contains the achieved clock frequencies for both clock domains, as well as a geometric mean, across ALL interconnect types, per system size. These will be used to drive simulations. Figure 7.10 graphs the system clock frequencies only. Relative comparisons, of the two GENIE variants versus the three others as a baseline, are provided in Table 7.11. Each entry corresponds to a percentage increase, or decrease, over a Qsys or CONNECT system clock frequency for a given system size.

When prioritizing clock frequency, the GENIE D3 and Qsys R4 variants are of interest, as they were designed to maximize performance. Excepting the smallest systems, GENIE consistently achieves

Table 7.8: Relative area usage of area-prioritized GENIE variant ($D = 5$) versus Qsys and CONNECT. Results in GENIE's favour are shown in green.

		N8M1	N16M1	N32M1	N8M2	N16M2	N32M2	N16M4	N32M4
QSYS R2	**ALM**	-23%	+30%	+16%	-32%	-40%	-41%	-44%	-43%
	COMB	-54%	-57%	-60%	-72%	-77%	-75%	-76%	-80%
	REG	+93%	+556%	+444%	+3%	-19%	-23%	+107%	+22%
QSYS R4	**ALM**	-72%	-51%	-53%	-68%	-70%	-70%	-54%	-70%
	COMB	-60%	-60%	-58%	-73%	-77%	-74%	-70%	-80%
	REG	-85%	-50%	-53%	-77%	-81%	-81%	-64%	-78%
CONNECT	**ALM**	-90%	-81%	-81%	-84%	-84%	-82%	-68%	-77%
	COMB	-90%	-90%	-89%	-88%	-89%	-87%	-81%	-86%
	REG	-20%	+178%	+170%	+80%	+58%	+60%	+294%	+154%

Table 7.9: Relative area usage of performance-prioritized GENIE variant ($D = 3$) versus Qsys and CONNECT. Results in GENIE's favour are shown in green.

		N8M1	N16M1	N32M1	N8M2	N16M2	N32M2	N16M4	N32M4
QSYS R2	**ALM**	+161%	+145%	+128%	+55%	+48%	+28%	-9%	+17%
	COMB	-51%	-56%	-58%	-56%	-58%	-66%	-72%	-66%
	REG	+1292%	+1286%	+1130%	+216%	+223%	+196%	+345%	+278%
QSYS R4	**ALM**	-5%	-7%	-7%	-27%	-27%	-35%	-26%	-38%
	COMB	-57%	-59%	-56%	-57%	-58%	-66%	-65%	-66%
	REG	+10%	+6%	+6%	-30%	-25%	-29%	-23%	-32%
CONNECT	**ALM**	-65%	-64%	-63%	-64%	-60%	-62%	-48%	-52%
	COMB	-90%	-90%	-89%	-82%	-80%	-83%	-77%	-76%
	REG	+473%	+488%	+510%	+450%	+531%	+517%	+746%	+688%

Table 7.10: System and Compute clock freqeuencies (in MHz) of the five final interconnect types versus system size. The geometric mean is calculated per-column for each clock domain.

		N8M1	N16M1	N32M1	N8M2	N16M2	N32M2	N16M4	N32M4
GENIE D5	**Sys**	321	318	219	344	315	270	291	258
	Compute	430	411	391	435	381	401	399	393
GENIE D3	**Sys**	369	331	276	361	357	302	349	308
	Compute	460	418	393	434	414	385	403	397
QSYS R2	**Sys**	269	254	196	290	251	221	286	231
	Compute	458	417	393	450	410	387	412	388
QSYS R4	**Sys**	379	328	226	380	355	271	344	263
	Compute	466	408	416	453	384	402	402	383
CONNECT	**Sys**	201	189	167	200	192	171	185	170
	Compute	452	419	398	440	406	389	421	394
Geomean	**Sys**	300	278	214	307	286	242	284	241
	Compute	453	415	398	442	399	393	407	391

a higher clock frequency than Qsys at this performance-focused corner. The largest observed gains are 17% and 22%, while the two losses to Qsys are by 2% and 5% at *N8M1* and *N8M2* respectively. CONNECT is the poorest performer, and struggles to achieve 200 MHz. Its low register-to-logic ratio corroborates these results. At the area-prioritized design corner, still consistently achieves higher clock frequencies than Qsys R2, by up to 26%. Even in the "unfair" comparison to performance-prioritized Qsys R4, area-prioritized GENIE D5 achieves a frequency within 2% for the largest-sized LU system.

Finally, we compare application execution times. These are measured in simulation, using a 33×33 block matrix as input, to ensure that the largest systems tested, which have 32 compute elements, are fully utilized during at least one iteration of the LU decomposition algorithm's outer loop. This will

Figure 7.9: ALM, combinational LUT, and register usage for GENIE, Qsys, and CONNECT-generated systems.

measure the impact of each interconnect variant on cycle count, rather than cycle time as reported by the previous clock frequency results. Since there are two clock domains in the application, a cycle count cannot be reported in a frequency-neutral way, as the ratio between the two clock frequencies affects the cycle count. Therefore, we must choose some System and Compute clock frequency to simulate at, and report the time taken. One possibility would be to simply use the achieved clock frequencies reported in Table 7.10, unique to each system size and interconnect variant. While this would give the most accurate estimate of application runtime, it would also obscure the effect of cycle count on performance between the interconnect variants. At the other extreme, choosing a single System and Compute clock frequency for all data points would be unrealistic, as the system clock frequencies drop significantly as system size increases, and a single choice would be either too optimistic for large systems or too

Table 7.11: Achieved system clock frequency of both GENIE variants ($D = 3$ and $D = 5$) relative to Qsys and CONNECT variants. Results in GENIE's favour are shown in green.

		N8M1	N16M1	N32M1	N8M2	N16M2	N32M2	N16M4	N32M4
GENIE D3 vs.	**QSYS R2**	+37%	+30%	+41%	+24%	+42%	+37%	+22%	+33%
	QSYS R4	-2%	+1%	+22%	-5%	+1%	+11%	+2%	+17%
	CONNECT	+84%	+75%	+66%	+80%	+86%	+77%	+89%	+81%
GENIE D5 vs.	**QSYS R2**	+19%	+25%	+12%	+19%	+26%	+22%	+2%	+12%
	QSYS R4	-15%	-3%	-3%	-9%	-11%	-1%	-15%	-2%
	CONNECT	+59%	+68%	+31%	+72%	+64%	+58%	+57%	+51%

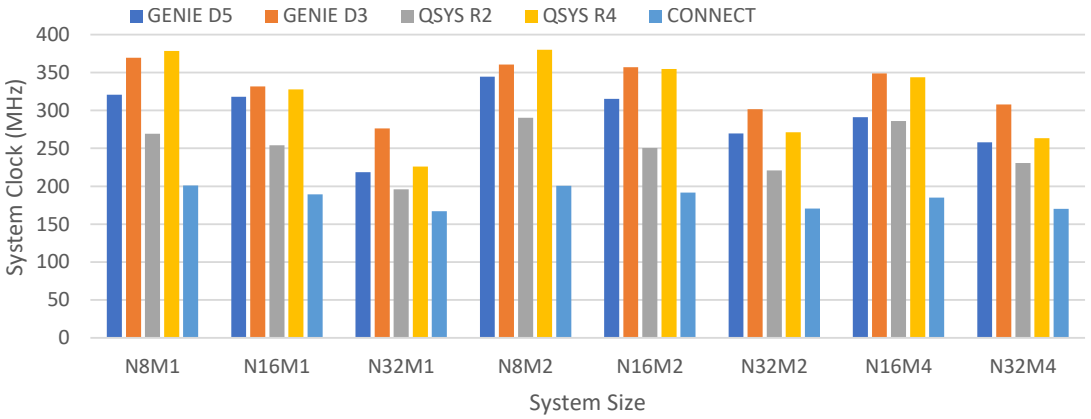

Figure 7.10: Achieved system clock frequency for GENIE, Qsys, and CONNECT-generated systems.

Table 7.12: Simulated execution times for processing a 33×33 block matrix, in milliseconds, for the three interconnect types.

	N8M1	N16M1	N32M1	N8M2	N16M2	N32M2	N16M4	N32M4
GENIE D3	146.49	101.33	89.47	149.39	104.06	72.83	102.20	72.69
QSYS R4	147.13	106.17	113.95	150.08	104.97	83.48	103.46	77.87
CONNECT	147.54	118.85	128.92	150.46	106.21	97.13	103.87	78.22
GENIE vs. Qsys	-0.43%	-4.56%	-21.48%	-0.46%	-0.87%	-12.76%	-1.23%	-6.66%
GENIE vs. CONNECT	-0.71%	-14.74%	-30.61%	-0.71%	-2.02%	-25.02%	-1.61%	-7.07%

pessimistic for small systems. As a compromise, we elected to choose a System and Clock frequency common to all interconnect types but different for each system size – for each of the eight sizes from $N8M1$ to $N32M4$, there will be a single Compute and System clock, calculated as the geometric mean of achieved frequencies from *all* five interconnect variants at that system size. These frequencies appear at the bottom of Table 7.10. In this comparison however, we will only look at three variants: the two performance-oriented variants, GENIE D3 and QSYS R4, and CONNECT.

Table 7.12 provides the resulting run times in milliseconds, and relative comparisons of GENIE versus the Qsys and CONNECT runtimes. Recall that GENIE is capable of supporting multicast, and the Left Block broadcast optimization is able to make more efficient use of interconnect resources. As we previously saw in Section 7.2.2, the effect of this optimization becomes more prominent as the bandwidth demand on extant memory controllers increases, which occurs at higher ratios of N to M. The greatest effect is at $N32M1$ with a 21% speedup over Qsys and 31% over CONNECT. CONNECT performs the worst, likely due to the poor bisection bandwidth offered by its double ring topology, which is shared

for all network traffic. This is in contrast to the GENIE and Qsys interconnect fabrics, which use separate networks for each traffic type. We could have used a different network topology, with similar area characteristics, for the CONNECT design point to alleviate this bandwidth shortage. However, as we saw in Section 7.2.5, topologies that increase the cardinality of each router will suffer a penalty in clock frequency, which may negate any cycle count reduction.

7.3 Conclusion

In this chapter, we built a hardware system composed of coarse-grained functional modules and connected with GENIE and two other types of interconnect synthesis tools/architectures: Qsys and CONNECT. Across a range of system sizes, the GENIE interconnect performed favourably in terms of area, clock frequency, and application runtime. Within the possible design space of interconnect, each tool had parameters that could prioritize the generated interconnect for performance or for area, and we performed comparisons at both corners. The most significant benefit of GENIE was application-specific, in that its support of multicast enabled the LU factorization engine to better utilize available bandwidth and often complete the computation in less time than when using non-multicast-capable interconnect to build the application.

In summary, GENIE has shown to be more than capable of targeting coarse-grained systems, even when compared against other tools that were also designed for the same purpose. We note again that the LU factorization application explored in this chapter contains main instances of the fine-grained Compute Element that was studied in Chapter 6, and that a single invocation of the GENIE tool was all that was required to build both the outer coarse-grained system as well as the fine-grained system submodules. This provides some opportunities for optimizations across hierarchy boundaries, which we leave as future work to explore.

Chapter 8

Automatic Pipelining and Synchronization

This chapter covers two related features of the GENIE interconnect synthesis flow:

- **Automatic Pipelining:** The insertion of registers into the interconnect in order to increase *performance* by limiting the maximum number of consecutive combinational logic stages.

- **Synchronization:** The intentional insertion of delays into the interconnect to allow exact cycle-accurate relationships between arrival times of transmissions, with the goal of enabling *correct and functional* globally-synchronized communication without the use of backpressure signals.

At first glance, these are not related – pipelining is a structural change used to increase clock frequency, and synchronization is related to cycle counts and correct circuit operation, and does not prescribe a specific physical implementation of how delays are achieved. However, GENIE implements the delays required to perform synchronization *also* by inserting registers. In both the case of pipelining and synchronization, there can exist many choices of where to insert registers into the interconnect, and some choices cost more total registers than others, yielding an optimization problem. GENIE solves both the pipelining and synchronization problems simultaneously using a unified register insertion engine that uses Integer Linear Programming (ILP) to find the area-optimal locations where registers need to be inserted to solve both problems. The automatic pipelining stage and the synchronization stage of the GENIE flow both emit ILP constraints to feed this unified solver. Note that we will refer to the 'insertion of registers' throughout this chapter to imply the insertion of GENIE Elastic Buffer primitives (described in Section 4.4), which comprise two registers when backpressure is used, and a single register when it is not.

The case study application for this chapter will be the convolutional neural network inference engine that was previously described in Section 5.2. The design of this application makes use of globally synchronized communications, as well as requiring pipelining to achieve high performance, and thus presents an ideal test case. However, we will first begin this chapter by describing the two separate features, synchronization and automatic pipelining, in isolation. This will be followed by results of using these features within GENIE to implement the convolutional neural network application.

8.1 Synchronization

A consequence of designing hardware at a high level of abstraction is the loss of knowledge or control over the detailed implementation. Specifically in the case of automated interconnect synthesis, the boundary between the designer and the tool is enshrined in the contract of the interface protocol. As we have mentioned before, existing signaling protocols such as Avalon[7] or AXI[9] employ the use of flow control and backpressure signals in order to allow maximum freedom in interconnect implementation. As a result, a functional module *cannot* assume, in general, that the interconnect will provide a specific end-to-end latency, or that the latency will even be constant during system operation. This presents a challenge when functional modules require synchronized data arrival from two or more sources. To guarantee synchronization even in the face of unknown interconnect latency, FIFOs or similar constructs can be inserted just before the functional module inputs, and dequeued when the module sees fit.

The alternative would be to manually create the interconnect with explicit, fixed, known latencies, such that the data arrives at each functional module input at the correct clock cycle by design. While this removes the area penalty incurred by FIFOs, it requires significantly more effort for the designer. They must either create the interconnect in HDL, giving up the productivity advantage of automated tooling, or (if their tools allow) manually specify the locations of registers in the interconnect. Not only must they add the correct *number* of registers, but they could potentially select among many equally-valid *locations* to insert them, with some yielding higher area usage than others.

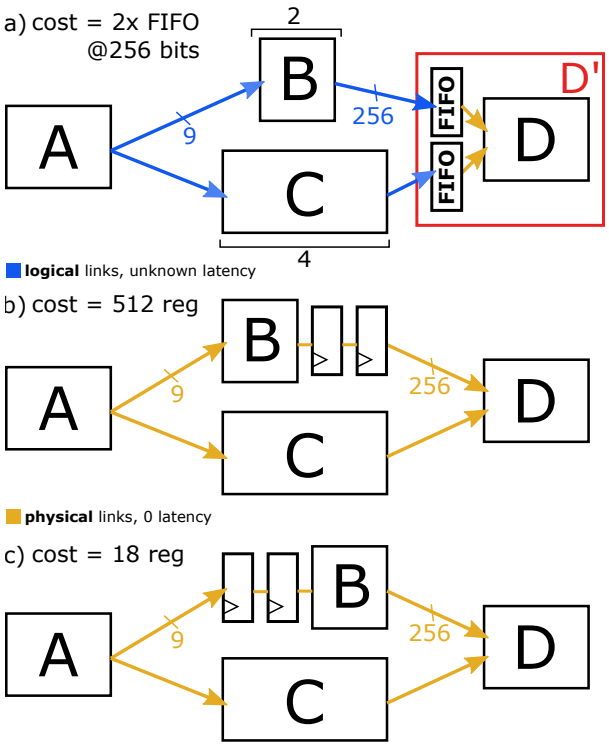

Figure 8.1: Three solutions for ensuring that inputs at module D arrive at the same time: **a)** Using an automated system-building tool and modifying module D (as D′) by adding FIFOs at its inputs, or **b)** building the interconnect manually and adding two pipeline stages after module B or **c)** before module B.

Figure 8.1 illustrates a motivating example, containing four functional modules, of such a synchro-

nization problem. Here, modules B and C are internally fully pipelined with fixed input-to-output latencies of 2 and 4 clock cycles, respectively. Each takes a 9-bit input and produces a 256-bit output, as a block RAM might commonly do, for example. Module D requires matching inputs to arrive during the same clock cycle. If a tool is used to build this system, the four modules would be connected with abstract logical links that are synthesized to an implementation with an unknown latency. The designer may employ the solution shown in Figure 8.1(a), where module D is wrapped inside a new module D′ that adds two 256 bit wide FIFOs to synchronize the data arrival at the inputs. However, if the designer had full control over the design of the interconnect, they may opt instead to use balancing registers to add the correct, fixed amount of extra latency to synchronize data arrival, and avoid the unnecessary hardware complexity of FIFOs. Two equally-valid solutions are shown in Figures 8.1(b) and 8.1(c), with the latter having the lower area usage of 18 (versus 512) registers. The choice of (c) over (b) may be trivial to see in this example, but a larger more complex system would present the designer with less-obvious choices.

In our original paper [64] titled *Synchronization Constraints for Interconnect Synthesis*, we proposed augmenting an interconnect synthesis tool (GENIE) with the ability to automatically create area-optimal, fixed-latency interconnect in response to the synchronization needs of the designer's application, effectively enabling solutions such as Figure 8.1(c) to be generated automatically. This is accomplished by accepting, from the designer, a set of *synchronization constraints*, which take the form of equations or inequalities that relate the end-to-end latencies of one or more logical links and a constant. GENIE then uses these constraints during interconnect creation by inserting the correct number of balancing registers, favouring solutions that use the minimal amount of total registers. The remainder of this section will be a description and formulation of these synchronization constraints, and how they can be used to help the user describe new types of interconnect requirements.

8.1.1 System Representation

Before we describe the synchronization constraint feature of GENIE, we will first revisit its baseline system and connection representation as seen from the point of view of the designer. Systems contain one or more functional modules containing interfaces. Some of these interfaces are Routed Streaming interfaces (see Section 3.2.3) that participate in complex automated interconnect synthesis. These are referred to as "logical links" in Figure 8.2. Ultimately, GENIE realizes these logical RS links as physical interconnect, which appear as "physical links" in the figure along with some of GENIE's interconnect primitives: a Split and Merge node (which also communicate with other modules not shown), and an elastic buffer. Synchronization constraints are processed later in the GENIE flow, after other interconnect primitives (Split, Merge, clock domain converters, address converters) have already been inserted and connected together.

8.1.2 Internal Links and Chains

In the existing representation, logical links originate and terminate at the interfaces of functional modules. In order to capture the type of global synchronization requirements depicted in the opening example shown in Figure 8.1, we must first extend the basic system representation with the ability to specify communication *through* modules. *Internal links* serve this purpose – they define a communication path from one of a component's receiving interfaces to one of its transmitting ones, following the flow of

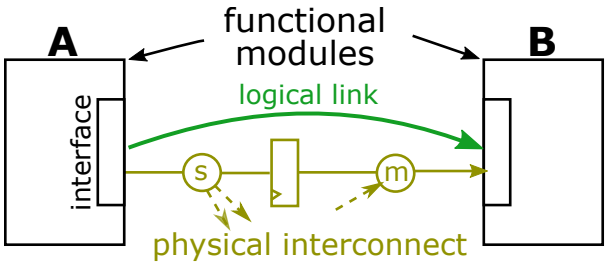

Figure 8.2: Baseline GENIE system representation

data. Each internal link has an associated fixed latency, in clock cycles, and is explicitly specified by the designer as part of a functional module's definition. It is also possible for an interface to participate in multiple internal links within a module, each with a different latency. Internal links are created with a distinct GENIE API call, available both in Lua and the C++ library.

After introducing internal links, we can now define a higher-level type of construct called a *chain*, which captures a transmission beginning at one module, through zero or more intermediate modules, and terminating at an ultimate destination. A chain defines a contiguous set of one or more logical links and internal links. Figure 8.3 illustrates an example of a chain spanning three modules - A, B, and C. The intermediate module B has an internal latency of 5 clock cycles, which is defined by its internal link.

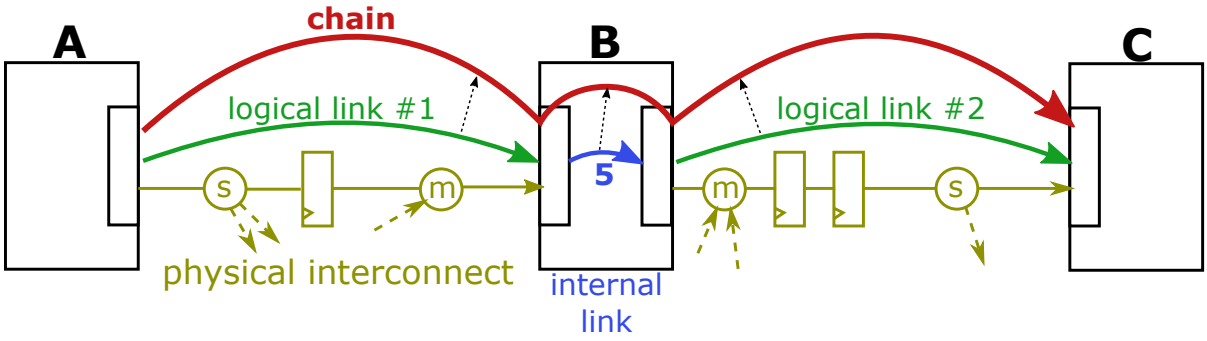

Figure 8.3: A chain spanning three modules A, B, and C, with its constituent two logical links and one internal link within B that has a latency of 5 clock cycles. Each logical link will be realized into the example interconnect shown.

8.1.3 Synchronization Constraints

Recall that the goal of this feature is to automatically generate interconnect that obeys user-specified synchronization constraints. Now, with the ability to capture transmissions spanning multiple modules using chains, we are ready to introduce the formulation of the constraints proper. Given a set of $N \geq 1$ chains $h_1, h_2, \ldots, h_N$, a synchronization constraint takes the form:

$$h_1 \pm h_2 \pm \cdots \pm h_N \ \mathbf{op} \ K \tag{8.1}$$

where **op** is a comparison operator (one of $<, \leq, =, \geq, >$), and K is an integer. Each term h_i represents the end-to-end latency, in clock cycles, of that chain. This general form allows the designer to specify

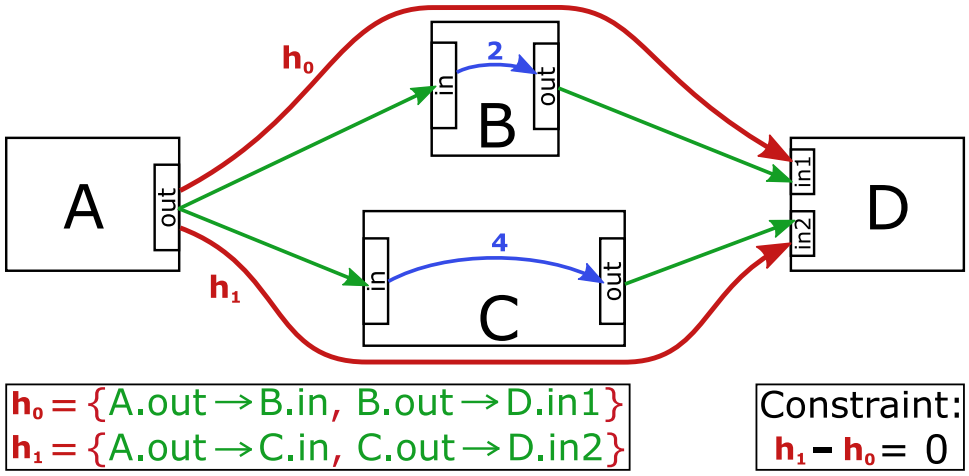

Figure 8.4: The example in Figure 8.1 restated using a synchronization constraint. The two chains from A to D are constrained by the user to have equal latency.

arbitrary latency relationships between chains, or to bound the latency of an individual chain. A chain (and its constituent logical links) can participate in multiple constraints.

Figure 8.4 restates the example system in Figure 8.1 as an input to GENIE using chains, logical links, and synchronization constraints. The explicitly-specified physical interconnect in the original example has been replaced with logical links between components A, B, C, and D, whose interfaces have been named 'in' and 'out' (with D having two inputs 'in1' and 'in2'). The latencies of B and C are captured with internal links. The requirement for D's inputs to arrive simultaneously has been captured as a synchronization constraint between two chains $h_0 = \{A.out \rightarrow B.in, B.out \rightarrow D.in1\}$ and $h_1 = \{A.out \rightarrow C.in, C.out \rightarrow D.in2\}$, with the constraint being that $h_0 = h_1$. The GENIE API introduces several new functions that allow the user to define chains of arbitrary length, and define synchronization constraints on these chains in the general form of Equation 8.1. Refer to Appendix A.3 for a description of these functions.

8.1.4 Optimization Problem Formulation

In general, there may be *many* legal solutions that satisfy a set of synchronization constraints, differing in the number of total inserted registers; ideally, we would like to find the solution that yields the fewest. Here, we will formulate this goal as an Integer Linear Programming optimization problem. While the problem will be presented in isolation for clarity, in reality it will be solved simultaneously with the automatic pipelining constraints described in the next section of this chapter. This combined problem will be restated in Section8.3.

Let $\mathbf{C}$ be the set of all user-provided constraints, each taking the form of Equation 8.1. For a constraint $c \in \mathbf{C}$, let $\mathbf{H}_c$ represent the set of chains that appear on the left hand side. A chain $h \in \mathbf{H_c}$ has an associated set of logical links, $\mathbf{G}_h$, which is a subset of all logical links $\mathbf{G}$. Chains also traverse internal links, that are represented by the set $\mathbf{T}$. Let $\mathbf{P}$ be the set of all physical links between existing GENIE interconnect primitives. By splicing registers into physical links, cycles of delay can be added in appropriate places to satisfy the overall set of synchronization constraints. If we define $L(p)$ as the number of registers to insert into physical link p, then the goal of the overall optimization problem is

to solve $L(p)$ for all $p \in \mathbf{P}$. We also wish to satisfy the constraints using the minimum total amount of registers. If $W(p)$ represents each physical link's width in bits, then this objective can be codified as the minimization of the following cost function:

$$\#\ of\ registers = \sum_{p \in \mathbf{P}} W(p)L(p) \tag{8.2}$$

Figure 8.5 illustrates the relationship between an example chain h_0 and its constituent logical, internal, and physical links, as well as the properties W and L of physical links. The width W of a physical link includes the widths of all constituent signals of the Routed Streaming logical links that travel over the physical link. These widths can vary throughout the physical network, as does the presence or absence of RS protocol signals. The latency L of a physical link is a numerical annotation that is only later realized as extra registers.

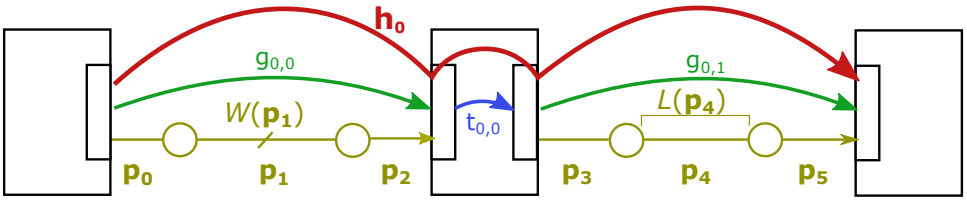

Figure 8.5: A single chain consisting of one internal link and two logical links, which are synthesized into interconnect containing a total of four primitives and six physical links p_0 through p_5. $W(p_1)$ is the width in bits of link p_1 and $L(p_4)$ is the necessary extra latency, in cycles, of p_4.

To solve the set of constraints $\mathbf{C}$, each constraint $c \in \mathbf{C}$ is first converted from the form of Equation 8.1, as provided by the user, into that of Equation 8.3 by expanding each chain term h_i into its constituent physical links p_i and internal links t_i:

$$h_1 \pm h_2 \pm \cdots \pm h_N\ \mathbf{op}\ K_c \tag{8.1}$$

$$L(p_0) \pm \cdots \pm L(p_N)\ \mathbf{op}\ K_c \pm L(t_0) \pm ... \pm L(t_M) \tag{8.3}$$

The left-hand side consists of unknowns (the latency of physical links to solve for), and the right-hand side contains constants (the user's constraint constant K_c together with the fixed latencies of internal links denoted by $L(t_i)$). The GENIE interconnect primitives that exist prior to register insertion currently have zero latency, but for generality's sake, any future primitives that *do* have non-zero latency should have their latencies included on the right-hand side. The resulting system of inequalities is in a canonical form suitable for solving using integer programming: the (nonnegative, integer) unknown variables $L(p)$ are on the left-hand side, and constants are on the right-hand side. A solution to the synchronization problem yields the values of $L(p)$ for all for all p, subject to the optimization criterion of minimizing the cost function of Equation 8.2, which is linear with respect to the unknown variables $L(p)$.

8.1.5 Variable Latency and Backpressure

The presented use cases for automatic synchronization are for fixed-latency modules without backpressure. There are currently no restrictions on using variable-latency modules (which use a valid signal) or those with backpressure (which additionally use a ready signal) from participating in automated synchronization, but GENIE still expects a single latency value to be annotated on each internal link. These additional use cases do not require synchronization for correctness, which is accomplished explicitly with flow control signals, hence making the presented use case for synchronization constraints redundant.

Could there still be a use for synchronization constraints for performance, rather than correctness purposes, in systems that ensure correctness via valid/ready signals? If a system has reconverging paths, it may be the case that a multi-input module representing a reconvergence point stalls all upstream inputs until all inputs have a valid token. In this situation, the branch (or chain, in GENIE's parlance) with the longest latency determines the latency of the final output. If the reconverging path is part of a loop, this latency directly translates into throughput, even if every functional module in isolation would be capable of accepting and producing one output per cycle in a fully-pipelined fashion. To increase throughput, additional capacity (in the form of registers or other storage elements) would need to be added on every other chain to match the latency of the longest chain. In principle, the existing synchronization constraint mechanism (with the constraints specifying equality for all chains) could insert the necessary capacity. If a module's latency is variable, the user would annotate its worst-case latency. This is similar to existing *buffer insertion* problems [29] targeting performance.

8.2 Automatic Interconnect Pipelining

This section describes GENIE's automatic interconnect pipelining functionality. The outcome of the process is similar to that of synchronization, in that registers will be inserted into existing locations in the interconnect. However, the goal is different: rather than to insert cycles of latency, the purpose of registers will be to break long combinational paths to increase clock frequency. For the synchronization problem, we were concerned with determining the value of $L(p)$ for all p, representing the latency in cycles to insert into a physical interconnect link, which is a count of the number of registers that will be inserted. For pipelining a link p, we re-use the same variable $L(p)$, but are only interested whether it is 0 or greater than 0, with the exact count not being important.

We have decided to pipeline based on combinational logic depth, which is a count of the number of technology-dependent FPGA LUTs traversed by signals through the interconnect. To pipeline the interconnect thus requires inserting registers such that the number of LUTs between any two registers is less than some fixed value. The other option would have been to pipeline based on absolute delays (for example, in nanoseconds). While theoretically more accurate, this approach would have required even more device specificity than logic depth, namely the estimation of delays across multiple process, voltage, and temperature corners, which can vary across different sizes of FPGAs within a single architecture. Our work also ignores FPGA interconnect delay, with 'interconnect' here referring to the low-level wiring, switching, and signal driver circuitry within the device that is controlled by configuration bits. Knowledge of this component of signal delay would require placement and routing information, which is beyond the scope of a tool that generates HDL.

1. Annotate each existing interconnect primitive with estimated logic depth values.

2. Create a timing graph from the set of interconnect primitives and their physical connectivity.

3. Deploy the on the timing graph to emit a series of ILP constraints on $L(p)$ to ensure register presence.

4. Traverse the timing graph and emit a series of ILP constraints on $L(p)$ to ensure register presence. This step is performed by our novel *Snake Algorithm*.

5. Solve for $L(p)$ subject to the constraints, with the goal of minimizing register usage. This solution can include any additional constraints on $L(p)$ specified by the user for synchronization purposes, solving both problems simultaneously.

8.2.1 Annotation of Logic Depth

GENIE maintains a database, for every interconnect primitive, containing area usage and timing (logic depth) information. This was previously described in Section 4.1. For automatic pipelining, we require use of the timing information. For each primitive type (Split node, Merge node, etc), and for various parameterizations of it (eg. "backpressure used" vs. "backpressure not used", number of input ports, etc), we store the combinational logic depth between:

- Each input port and output port.
- Each input port and an internal buried register.
- Internal buried registers to each output port.

where the granularity of representation is that of individual HDL ports. The logic depths are obtained by synthesizing each primitive, in each parameterization, with the back-end FPGA CAD software in a test harness, and storing the results in a database that GENIE loads when it is executed. This logic depth information is necessary for building a timing graph that will be used to perform automatic pipelining.

When the synchronization/pipelining stage of the GENIE flow is run, the parameterizations of thus-far-existing interconnect primitives are known, and the timing information for them can be obtained from the database. However, the granularity of GENIE's interconnect synthesis at this time in the flow is that of "physical links", which employ the Routed Streaming protocol. At this level of abstraction, a port or connection is not an HDL port or HDL wire, but a bundle of related signals of different roles (eg. data, valid, ready, address, etc.). Since the individual HDL ports of an interconnect primitive module are treated together as a single interface (and HDL wires to/from them treated as a single physical link) the detailed logic depths obtained from the database are simplified and worst-cased (using the longest possible path) to obtain representative values.

Figure 8.6 shows an example. Here an interconnect primitive has a single RS input port called `in_iface` and a single output port called `out_iface`. Only the data and valid signals are being used on each, and the appropriate parameterization has been obtained from GENIE's database. The obtained logic depths from both of the input port's constituent HDL ports, `I_DATA` and `I_VALID` have arcs to internal registers (collectively labeled "INT") and to one or both of the output HDL ports. The ones that terminate in internal registers, having depths of 1 and 3, are worst-cased to a single value of 3 and annotated as `in_iface`'s logic depth value. Only a single internal-to-HDL-output arc exists with a depth of 1, so this is kept as the annotated value of `out_iface`'s logic depth. The worst-case HDL input to

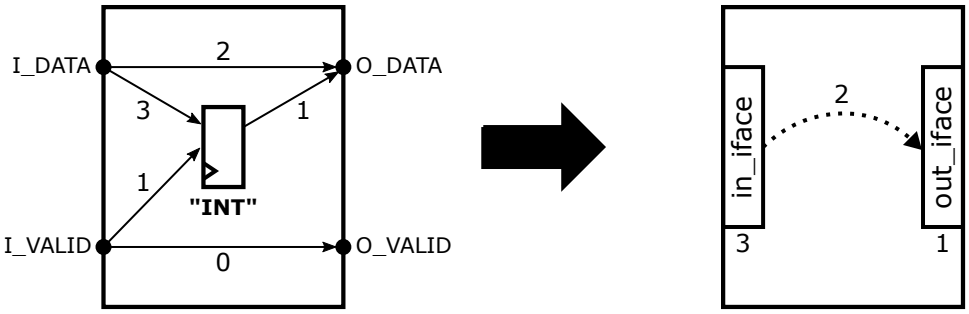

Figure 8.6: **Left:** Timing database representation of logic depths for an example interconnect primitive, at the level of HDL ports. **Right:** GENIE view of the same primitive at the level of RS interfaces, with timing database depths worst-cased to single representative values.

HDL output logic depth is 2, so this is used as the single representative logic depth from `in_iface` to `out_iface`.

As a consequence of this level of abstraction, when physical links between interconnect modules have register insertion performed, it is performed uniformly on *all* constituent HDL signals that are part of that bundle. The reason for the detailed explanation of this annotation is mainly to highlight its limitation and the possibility of future work in which the constituent signals are treated independently and can be pipelined by potentially differing amounts, as long as global end-to-end correctness and synchronization are maintained.

Thus far, the discussion has been about annotating logic depth on GENIE interconnect primitives only. However, we have also provided the ability for the designer to annotate the RS interfaces of their own functional modules with an optional logic depth attribute. This is a single integer, defaulting to zero, which represents the number of combinational logic stages from any member HDL port of the RS interface, to some internal register deep within the module. For combinational paths through a user module, which do not terminate at an internal register, the user can define internal links between an input port and an output port, internally within a module. These are the same types of links previously introduced for the synchronization functionality in the previous section. However, instead of specifying a latency (which would imply the number of that many registers), the user can (mutually exclusively, of course) specify a logic depth instead, defining a combinational path through the module. Both the port-based and internal-link-based logic depth annotations must be manually measured or estimated by the user, and become part of the input to GENIE. With this feature, GENIE is able to pipeline the interconnect taking the logic depth within (and through) users' modules into account.

8.2.2 Timing Graph Creation

After all the interconnect primitive modules have been annotated with their logic depths, the interconnect modules, user modules, and physical connectivity between them are transformed into a timing graph. This is a directed, and possibly cyclic, graph that preserves the data flow direction in the original hardware netlist, and is annotated with logic depth values. It will be used to lay down ILP constraints for automatic pipelining The vertices in the timing graph represent one of two possible things: *possible* locations of pipeline registers on physical links between modules, and *known* locations of registers within the cores of the modules. If vertices represent (fixed and possible) registers, then the edges represent combinational logic paths between them, annotated with logic depth values as weights.

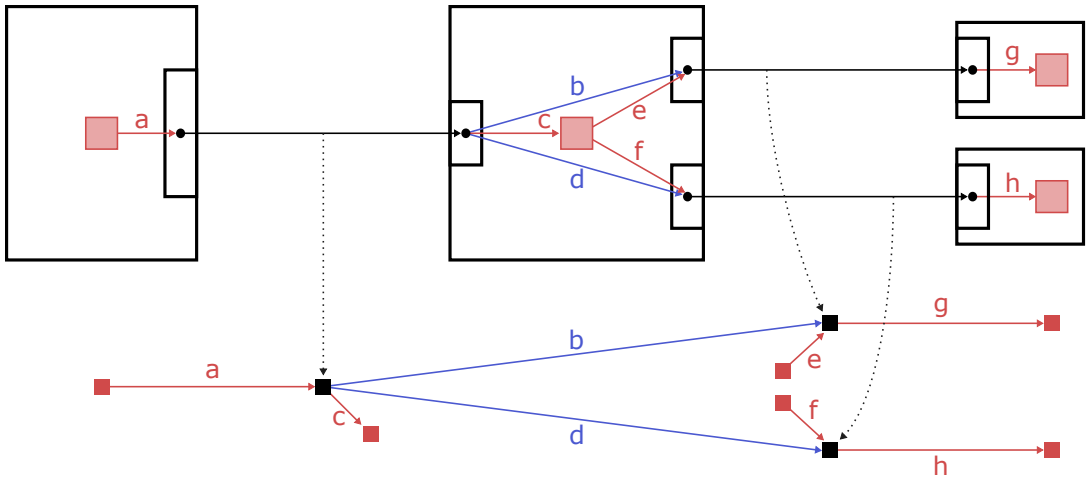

Figure 8.7: **Top:** Example domain with four modules (user and interconnect) with internal registered cores (red), inter-module physical links (black), and purely combinational internal links (blue). Logic depth values are annotated with letters a-h. **Bottom:** Timing graph extracted from the above. Red vertices represent terminal, internal registered cores. Black vertices represent pipeline-able locations on inter-module links. The edge weights are logic depths.

Figure 8.7 illustrates the process using an example interconnect domain with four modules: a user module with one interface acting as a source, a middle module which could be a 2-output Split node, and two more user modules acting as sinks. Logic depth values of both the interface-to-internal-register variety (red) and interface-to-interface variety (blue) are labeled with the letters a through h. Each of the three external physical links becomes a black vertex in the extracted timing graph. The red vertices in the timing graph represent the cores of the four modules. These are locations of inflexible registers that ultimately initiate or terminate combinational logic paths. Because of this, directed graph connectivity is intentionally cut through red vertices. For example, the core of the center module, which is fed by one input interface and itself feeds two output interfaces, is not represented as a single vertex in the timing graph, but instead as three separate vertices.

As a final step, the timing graph is post-processed to remove edges with zero weights. The two vertices bordering a zero-weight edge are combined into one vertex that arbitrarily takes the identity of one of the original vertices, such that it continues to represent a single pipeline-able physical link. For the case of zero-weight edges that originate or terminate at an internal vertex (one that does not represent a physical link, but a connection to an internal module register), the internal vertex and the edge are culled from the graph completely. These steps are not strictly necessary, but they reduce the size of the timing graph and simplify both the operation and understanding of the traversal algorithm. In the following sections, it is assumed that all weights are nonzero integer values.

8.2.3 Timing Graph Traversal

Here we provide a high-level overview of how the timing graph is traversed and give intuitive reasoning behind the more formal description of the Snake Algorithm presented in the next sections. The goal of timing graph traversal is to generate a set of ILP constraints on the variables $L(p)$ which, when satisfied, guarantee a minimum number of registers to be inserted within contiguous segments of physical links. The criterion for pipelining is a user-specified maximum logic depth, specifying the longest desired

combinational path between two registers. In GENIE, this value defaults to five stages and can be specified on a per-system basis. A trivial solution would of course be to pipeline every link, but this would, in the general case, require more area than necessary. Additionally, just as with synchronization constraints, there can be many legal solutions that yield different area results. Thus, there exists an optimization problem here, similar to the one with synchronization constraints. In fact, the two problems share the same ILP optimization goal function, and this will be described later in the appropriate section.

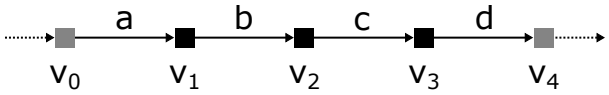

Figure 8.8: A path within a larger timing graph containing five vertices v_0-v_4 and edge weights a-d.

Before attempting to solve the pipelining problem on an entire timing graph, we will first examine the simpler problem of pipelining a smaller section of the graph. Figure 8.8 shows such a section. Recall that vertices represent the locations of registers (either fixed existing ones prior to pipelining, or pipeline-able locations located on inter-module physical links). Edges represent combinational paths between registers, and are annotated with logic depths. Figure 8.8 is a path containing five vertices, and could be located anywhere within the timing graph. Let it also be a path where pipelining is definitely required: that is, the sum of the edge weights a through d, representing logic depths, exceeds the user's desired maximum logic depth specification D_{max}. We say that such a linear subgraph is *overweight*. Then, as long as each of the individual weights is less than or equal to D_{max}, we can successfully pipeline this section of the graph by setting *some* combination of $L(v_1)$, $L(v_2)$, and $L(v_3)$ to at least 1, including the trivial solution of setting all three to nonzero values, which would pipeline every link. Note that we are not concerned here with the extreme ends of this section at v_0 or v_4, as inserting registers there does not affect the D_{max}-satisfiability of the section under consideration.

Let us impose two additional conditions on this hypothetical subgraph. In addition to having a total weight exceeding D_{max}, let the values of the weights be such that by removing *either* a or d from consideration, the respective remaining sums $(b + c + d)$ or $(a + b + c)$ would be less than or equal to D_{max}. If both of these conditions are satisfied, then the requirement for successfully pipelining the subgraph becomes much simpler: it is now merely sufficient that *at least one* of $L(v_1)$, $L(v_2)$, or $L(v_3)$ is nonzero. Expressed as an inequality, this requirement can be written as $L(v_1)+L(v_2)+L(v_3) > 0$, which is directly representable as an input constraint to an ILP problem, which we call a *pipeline constraint*.

More generally, we say that an overweight linear subgraph with vertices labeled v_0 through v_N is *minimally overweight* if placing a register at locations v_1 or v_{N-1} partitions the subgraph into two linear subgraphs whose weights are less than or equal to D_{max}. It follows that in a minimally-overweight subgraph, placing a register at *any* location from v_1 through v_{N-1} yields a similar result – moving the partition point between the two extremes can not yield a total partition weight greater than observed at the extremes, as the weights are nonnegative. A pipeline constraint, specifying that at least one vertex from v_1 through v_{N-1} should contain a register, is enough to successfully constrain the subgraph for successful pipelining.

Our approach to pipelining the entire timing graph is therefore to visit all of its minimally-overweight subgraphs, and for each, emit a single pipeline constraint. Each subgraph must have at least three vertices: a head, a tail, and at least one interior vertex representing a pipeline-able location. By simultaneously solving all pipeline constraints, yielding a solution to the variables $L(p)$ and then inserting

registers according to $L(p)$, there will exist no locations in the timing graph where more than D_{max} consecutive logic stages are found. This is the essence of our Snake Algorithm, whose main focus is to perform an orderly traversal of the entire timing graph to visit all the necessary minimally-overweight subgraphs.

8.2.4 Snake Algorithm: Linear Case

The Snake Algorithm is a novel method that we developed to construct a pipelining problem using ILP. First we will introduce it in a simplified way, by operating on a timing graph that consists of a linear chain of vertices, starting and ending on internal vertices belonging to modules. The full algorithm simply deconstructs a more general timing graph into many such linear subgraphs and operates much in the same way, with additional steps added to handle branching, loops, and reconvergence in the graph. Our example graph in Figure 8.9 has ten vertices, $v_0 - v_9$, with the logic depths indicated on the edges. Let D_{nm} be the *sum* of edge weights starting at vertex v_n and ending at vertex v_m. For example, let us use a maximum logic depth D_{max} of 5.

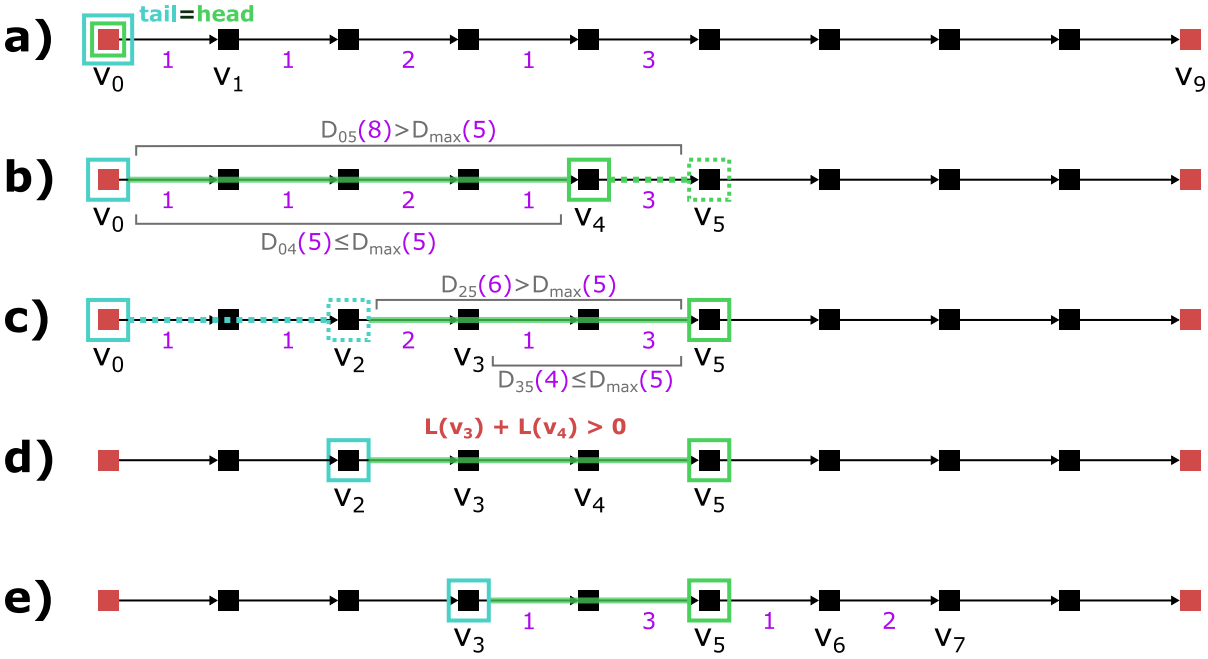

Figure 8.9: Snake Algorithm in operation on a linear timing graph.

The current state of the algorithm is represented by the *snake* – a set of adjacent vertices within the graph, defining a path from a tail vertex to a head vertex. The weight of the snake D_{snake} is equal to $D_{(tail)(head)}$. Initially, the snake contains just the source terminal vertex of the graph. In Figure 8.9a, this is v_0, and it is both the tail and the head of the snake. Next, the head of the snake advances forwards until the snake is *overweight*, meaning $D_{snake} > D_{max}$. If this never occurs, then pipelining is not required at all and we are done. However, in Figure 8.9b, this occurs when the head reaches v_5, as the addition of weight $D_{45} = 3$ brings the snake over the weight limit of 5. At this point, the section of the graph contained within the snake definitely requires pipelining, but we can not guarantee that this can be accomplished with just a single register, which would be the ideal case and allow us to create a single ILP pipeline constraint.

The next step is to remedy this by advancing the tail forward as far as possible until the snake becomes minimally-overweight. In Figure 8.9c, this occurs when the tail reaches v_2, as the next weight $D_{23} = 2$, if subtracted from D_{snake}, would cause the snake to stop being overweight. After advancing the tail, Figure 8.9d is the result. The snake is now minimally-overweight and we can emit a pipeline constraint, requiring that at least one register be placed within the snake, excluding the tail and head vertices. In Figure 8.9d, we can see that these are the two vertices v_3 and v_4, and the emitted constraint is also displayed. The variable $L(p)$ is the number of registers to insert into physical link p, and each vertex represents a possible pipeline register location for an associated physical link.

This constraint will ensure that a pipeline register will be inserted for the physical link represented by v_3, v_4, or both. Note that by successfully pipelining the range v_2-v_5, we also guarantee that this successfully pipelines v_0-v_5, which was the original range covered by the snake before its tail was advanced. This is important, as at first glance it appears that the section of the graph to the left of the snake, v_0-v_2 is neglected by the pipeline constraint and that more than D_{max} consecutive logic stages may remain in the overall graph even after registers are inserted. Our argument against this is as follows: assume the worst case, in which the solution to the pipeline constraint emitted at d) results in only a single register inserted at v_4. It is the worst case because this would give the range v_0-v_4 the greatest chance of being overweight. However, we know that the cumulative weight for this range, W_{04}, is less than or equal to D_{max}, because in step b), we advanced the head from v_4 to v_5 for that exact same reason. Therefore, the satisfaction of the emitted pipeline constraint at step d) also guarantees that the entire graph up to the head vertex will be successfully pipelined.

Finally, in step e), we move the tail forward one vertex, and since the snake was minimally-overweight at step d), we now have a non-overweight snake with a total weight of 4. The algorithm now repeats from step b), advancing the head forwards until the snake is overweight again, which will happen when the head reaches v_7 and the snake weight becomes 7. Looking forward, repeating the next step, step c), will move the tail forward to v_4, where the snake will be minimally-overweight again. Note that due to the pipeline constraint emitted during step d), at least one register is guaranteed to be placed at either v_3 or v_4 meaning that at worst, if it is placed at v_3, it would act as a terminal red vertex and reduce the remainder of the pipelining problem to the situation shown in Figure 8.9a. If the register is placed even further forwards, then it is an even more optimistic scenario, as not only does the problem resemble a), but there is already at least one register placed in the remainder of the path ahead. At this point, steps b) through e) repeat and the algorithm terminates when the head attempts to move past the end of the graph. The output is a set of pipeline constraints, the number of which depends on the weights in the graph.

8.2.5 Snake Algorithm: General Case

Finally, we can extend the idea behind the Snake Algorithm presented in 8.2.4 to a general timing graph. The approach is to traverse every possible source vertex to sink vertex path in the graph, and treat each like the linear case previously presented. A source vertex is one with only outgoing directed edges, and a sink vertex only has incoming edges. Additional bookkeeping is employed to avoid visiting the same parts of the graph more than once, and to avoid having the number of extracted linear paths be a quadratic function of the number of source and sink vertices. The state of the algorithm now contains *many* snakes, rather than one snake, stored within a stack Q. The snakes in the stack are completely processed one at a time, as per Section 8.2.4, but will spawn additional child snakes when encountering

divergence points in the graph. These snakes are pushed onto the stack for later continuation. The global algorithm state also includes a set of visited vertices, which is initially empty. Each snake has the following properties:

- The list of vertices that are currently in its body, including the head vertex at the front and the tail vertex at the end.
- The number of vertices within its body that have not been previously visited by any other snake.
- The weight of all edges currently spanned by the snake.

The full Snake Algorithm is presented as Algorithm 4. The queue is populated with initial snakes on Line 3. In order to process each snake, three phases are performed, corresponding roughly to Section 8.2.4: advancing the snake's head (Line 11), advancing its tail (Line 23), and emitting an ILP constraint (Line 31). Advancing the head now requires taking into account *all* the possible vertices that the head can advance into. Each such possible vertex, stored in the set N, yields a new extended snake, and these are pushed onto the stack. The (arbitrarily) topmost of these pushed snakes completely replaces the initial un-extended snake, assuming its identity, and processing resumes from there. The other pushed snakes will end up being resumed later.

There is also a change in how the tail advancement works, compared to the introduction given in Section 8.2.4. Previously, the snake's tail was advanced as far as possible while keeping the snake overweight, and the vertices in the interior of the snake on the open set $(head, tail)$ would participate in ILP constraints. For algorithmic simplicity, the semantics have changed somewhat: the tail is now advanced one step further, until the snake *stops* being overweight. Then, the tail vertex becomes part of the ILP constraint, on the half-open set $[head, tail)$. The result is the same, but it avoids the need to backtrack the tail once discovering loss of overweight status, or to look ahead to individual edge weights rather than considering the total weight of the snake alone at any given time.

Vertices are only marked visited once they leave a snake through its tail. If at any time a snake contains only previously-visited vertices, then processing of it can be terminated early. This is how cycles and reconvergence in the graph are handled. The output of the algorithm is a set of ILP constraints. Traversal of the graph is bounded by the number of edges, which also bounds the maximum number of ILP constraints emitted. However, an ILP constraint containing only one vertex (one physical link) is trivial, as it guarantees a solved value of 1.

8.3 Solution of Synchronization and Pipeline Constraints

In this section, we describe the simultaneous solution of the synchronization constraints specified by the user from Section 8.1 and the pipeline constraints emitted by the Snake Algorithm in Section 8.2. Both synchronization and pipeline constraints involve the integer variables $L(p)$ that correspond to the number of registers that should be inserted on a physical link p.

The optimization goal remains the same as it did in Section 8.1.4: to minimize the total number of 1-bit register elements inserted into the system. For any given physical link p, this is a product of its width in bits $W(p)$ and the number of inserted register stages $L(p)$. To summarize, the inputs to the final ILP problem are shown in Equations 8.4. From top to bottom, it contains synchronization constraints, each of which constrains the L variables as a function of a user-chosen relational operator **op**, a user-chosen

ALGORITHM 4: Snake Algorithm for Generating ILP Pipeline Constraints

inputs : timing graph $G = (V, E)$ with edge weights D, maximum logic depth D_{max}
output: zero or more ILP constraints on $L(p)$

```
Q = {}
visited = {}
// Initialize Q with snakes, one for each source vertex in G
foreach {v ∈ V | [u, v] ∉ E ∀u ∈ V} do
    s = snake{.verts = {v}, .weight = 0, .unvisited = 1}
    Q.push(s)
end
while !Q.empty do
    // Grab a snake off the stack and process it completely
    s = Q.pop()
    snake_done = false
    while !snake_done do
        // Advance head of snake until it is overweight
        while s.weight ≤ D_max do
            // Examine all possible next vertices the snake head can move forward into
            N = {n ∈ V | [s.head, n] ∈ E} foreach n ∈ N do
                newsnake = s // Construct a new snake based on s, with new head vertex
                newsnake.verts.push_head(n)
                newsnake.weight += D([s.head, n]) // Add new edge weight to snake
                if n ∉ visited then
                    newsnake.unvisited++ // Update unvisited vertex count
                end
                Q.push(newsnake)
            end
            // s resumes as one of the extended snakes just pushed, if any
            if !Q.empty then s = Q.top else snake_done = true, break
        end
        // Advance tail of snake until it just becomes underweight, or too short
        while s.weight > D_max and |s.verts| > 2 do
            old_tail = s.verts.pop_tail()
            visited = visited ∪{old_tail} // Mark outgoing tail as visited
            s.weight -= D([old_tail, s.tail]) // Reduce snake weight
        end
        if s.unvisited == 0 then // Snake only contains visited vertices, can stop early
            snake_done = true
            break
        end
        // Include all vertices in the half-open interval [tail, head) in the constraint
        emit ILP constraint: L(s.verts[tail]) + L(s.verts[tail + 1]) + ··· + L(s.verts[head − 1]) > 0
    end
end
```

constant K_c, and the fixed latencies of modules' internal links L. Next come the pipeline constraints emitted by the Snake Algorithm, each constraining the sum of one or more L variables to be nonzero. Finally comes the optimization goal which is to minimize the total number of inserted registers. GENIE internally uses the *lpsolve* [1] package to solve the ILP problem.

After solving for L, we only need to insert the corresponding number of registers on each physical link. However, before inserting the actual registers, there exist two additional optimization steps that transform the interconnect based on the solved values of L, which we describe in the next two sections.

$$\begin{gathered} L(p_0) \pm \cdots \pm L(p_N) \textbf{ op } K_c \pm L(t_0) \pm ... \pm L(t_M) \\ \cdots \\ L(p_0) + L(p_1) + \cdots + L(p_J) > 0 \\ \cdots \\ \text{minimize} : \sum_{\forall p} W(p)L(p) \end{gathered} \tag{8.4}$$

8.3.1 Systolic Retiming Transform

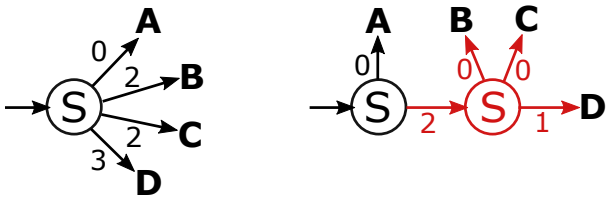

Figure 8.10: Left: a Split node feeding four physical links with differing latencies. Right: applying systolic retiming, the number of required register stages is reduced from 7 down to 3.

After solving for L for every physical link, situations may arise similar to the one on the left side of Figure 8.10. Here, a Split node fans out to four physical links that were each assigned different values of L. Each link feeds some other interconnect or user module, labeled A through D. Realizing this assignment would require 7 register stages – the sum of the 2, 2, and 3 register stages for destinations B, C, and D. However, if we allow for the freedom to change the topology of interconnect primitives *after* initial latency assignment, a cheaper solution can be found, as shown in the right side of the figure where an extra Split node is used to reduce the cost to three register stages. We call this optimization the *systolic retiming transform*. The overall process is:

1. Sort the Split node's fanout physical links into bins by their latency assignments $L(p)$.
2. Remove the initial Split node.
3. Create a Split node for each bin from step 1.
4. Connect each Split node from step 3 to the original destinations in the respective latency bins, but reset the latency on these new physical links to 0.
5. Connect the Split nodes together with physical links whose latencies are the differences between successive bins.

6. The last Split node can be removed and its sole fanout reassigned to the second-last Split node.

After performing the transform, new values of L exist on the updated interconnect. Note that the timing as observed by A, B, C, and D is not affected. Since the original Split node is replaced with one or more Split nodes with lesser cardinality than the original, the number of logic stages for each Split node should also be less than the original, which ensures that the existing pipelining provided by the new register placements is sufficient. As future work, a similar transformation could be performed on Merge nodes.

8.3.2 Long Register Chain Optimization

The second kind of post-ILP optimization that we perform involves choosing the type of interconnect primitives used to realize the values of $L(p)$. Nominally, this would simply be GENIE elastic buffer primitives (Section 4.4). However, if a the value of L is sufficiently large for a particular physical link, inserting that many consecutive elastic buffers into that link may not be the most area-efficient option.

Instead, we may opt to replace that entire chain of elastic buffers with a *single* Delay Buffer primitive (Section 4.7) that implements the required number of delay cycles using FPGA distributed memory. For each physical link, the estimated area costs of the two approaches (chain of elastic buffers vs. single delay buffer primitive) are compared, and the cheapest option is chosen. The estimated cost is a function of both the width of the link in bits as well as the number of cycles of delay requested by the L value for that link. Lookups in GENIE's primitive area database (Section 4.1.1) are used, and take into account the stepwise area consumption characteristics (with respect to width and depth) of distributed memory blocks.

8.3.3 Summary

The combined optimization problem of satisfying synchronization constraints and automatic pipelining is solved with ILP, yielding a value of $L(p)$ for every physical link p in an interconnect domain. Then, the systolic retiming transform is performed on Split nodes with multiple fanout and differing L values on outgoing physical links. Finally, the (possibly transformed) domain's L values are realized physically as either elastic buffers or delay buffer primitives, completing the entire register insertion portion of the GENIE flow.

8.4 Results

In this final section, we present results that demonstrate both the effectiveness and limitations of the synchronization and automatic pipelining flows that have been described in this chapter. First, we will focus on automatic pipelining in isolation, in both fine- and coarse-grained systems, using the LU Decomposition application. The goal here will be to demonstrate the effects on clock frequency and area as a result of changing the amount of automatic pipelining. Some of these results have been collected in previous chapters, and will simply be referred to.

Then, we will shift our focus to studying the use of synchronization constraints on the Convolutional Neural Network application introduced in Section 5.2. Synchronization is an issue of correctness rather than quality of area/frequency results, so the focus here will be a more detailed study to show that the GENIE synchronization flow is indeed inserting registers in intelligent and correct locations to enable

a functional design. Finally, we will return to automatic pipelining, but on the CNN design, where we will examine some of the limitations of our approach, which relies on the bounding of logic depth alone and ignores the real distances that signals need to travel as a result of FPGA placement.

Clock frequency and area measurements are obtained through full compilation and timing analysis in Quartus Prime Pro 17.0, targeting an Arria 10 device – `10AX115N2F45E1SG` for LU engine designs, and `10AX115S2F45I2SGES` for the CNN. Clock frequencies were geometrically averaged over six compilation seeds.

8.4.1 Automatic Pipelining: LU Decomposition Engine

Our first set of results demonstrates the effectiveness of automatic pipelining by observing how varying the maximum logic depth parameter, D_{max}, affects interconnect area usage and total system clock frequency. The application being tested is the LU Decomposition Engine from Section 5.1, specifically the outer (coarse-grained) level of design hierarchy. Automatic pipelining is enabled for the inner fine-grained CE design as well, but D_{max} is kept constant at the default of 5 stages, while we vary the D_{max} for the outer level. This experiment was previously performed in Section 7.2.3 as part of a design space parameter sweep for comparing GENIE against other tools. Data and conclusions may be found there.

8.4.2 Automatic Pipelining: Compute Element

Up to this point, the effects of automatic pipelining have only been observed as measurements of area and clock frequency. It would also be instructive to observe the detailed effects on individual register insertion on a smaller, but still realistic, design example. The LU and CNN design examples are relatively large, such that excepting trivially small parameterization of those applications, there are too many functional and interconnect modules to effectively visualize at the level of detail of individual registers. Here, we will study the effects of automatic pipelining on the much smaller fine-grained Compute Element subsystem of the LU engine, which can be illustrated in its entirety at our desired level of detail.

Figure 8.11 shows netlists containing the CE's functional modules (in green), GENIE-inserted interconnect primitives (in white), automatically-inserted registers (in red), and the connections between them. Each connection is a Routed Streaming physical link representing a bundle of different signal types. The three illustrated systems differ in the values of D_{max} supplied to GENIE's automatic pipelining process. Using a sufficiently large value of 10, no extra registers are inserted, yielding the unpipelined version in Subfigure a). The subsequent Subfigures b) and c) use smaller values of D_{max}, 5 and 3 respectively, which increase the amount of inserted registers, shown as red rectangles. Note that the two register stages seen in Subfigure b), where $D_{max} = 5$, do not reappear at a more aggressive setting of $D_{max} = 3$ in Subfigure c). They are instead replaced by two stages of pipelining, which further subdivide the combinational logic in that part of the CE, which happens to correspond to the cache write paths. This is the most complex interconnect in the CE, with two sources and five sinks, and it is not surprising that the longest combinational paths exist there, and are therefore pipelined first.

We also swept D_{max} across a wider range of values to perform the same clock frequency and area analysis for the CE as we previously did with the coarse-grained outer LU system. These results are in Table 8.1. The critical path in both clock domains lies within the functional modules, explaining why the clock frequencies (notably the compute clock) do not change much with additional interconnect pipelining. The interconnect register count, however, does change, confirming our previous detailed

observations.

Table 8.1: Achieved system and compute clock frequencies and interconnect register count for generated CE versus maximum logic depth.

$\mathbf{D_{max}}$	**1**	**2**	**3**	**4**	**5**	**6**	**10**
Compute Clk (MHz)	487	497	488	487	471	485	465
System Clk (MHz)	630	629	613	633	622	630	610
Registers	9429	4017	3024	2987	1484	1317	845

8.4.3 Synchronization Constraints: CNN

The remainder of the results in this chapter will focus on the Convolutional Neural Network design, which requires precise cycle-level synchronization of transmissions to ensure correct behavior. Recall that the design has a two dimensional array of Dot Product Units (DPUs) containing N columns and typically 16 rows. Each DPU has two inputs: image data, and kernel data, and these two streams of data must arrive simultaneously. This poses an issue due to the differing latencies of intervening functional modules, as well as due to registers inserted by GENIE within the interconnect itself. Figure 8.12 illustrates an example of such a situation for the inputs of a single DPU. The image delivery paths are generally longer, requiring registers on the kernel path to compensate. Here, four extra registers would be needed to achieve this.

We specify several sets of synchronization constraints to guide GENIE into inserting the necessary delay registers. These are shown in Figure 8.13. A naive approach would lead to the creation of a significant number of synchronization constraints to achieve our goal: these would be used to equalize the delays between the single kernel data chain feeding each DPU and the $N \times N$ total possible image data chains to that DPU. Since there are $N \times 16$ DPUs, many total constraints would be required. However, we can reduce the required number of constraints by decomposing the synchronization problem into two phases. First, as illustrated in Figure 8.13a, we first create a *single* set of N^2 constraints between chains h_R, which start at the launch signal and terminate only as far as the image buffer inputs. These will ensure that the image iterators and buffers process through the image data in perfect lock-step by synchronizing the arrival of the "launch" control signal that begins iteration, as well as the iterated addresses that are fed into the image buffers. To finalize the decomposition, in Figure 8.13b we return to the original problem of synchronizing kernel and image data. However, rather than touching all N^2 image data chains, we only need N of them: the ones that go through a single (arbitrary) iterator and each image buffer. These are constrained to match each other, as well as the single kernel data chain h_{K0} which represents the single kernel buffer that feeds the DPU at location $(0, 0)$ of the array. The transitive nature of equality causes the constraints from subfigures a) and b) together to implicitly recover all the necessary paths.

Finally, we present the results of the application of synchronization constraints to the system. The goal is to demonstrate that registers have been correctly inserted, into intelligent area-minimizing locations, to compensate for delay differences. The CNN system was configured as follows:

- **Synchronization constraints:** Enabled, and specified as described above. Functional modules are also annotated with the correct internal links, specifying input-to-output pipeline latencies.
- **Topology:** Manually-specified, containing two $N \times N$ full crossbars using split and merge nodes:

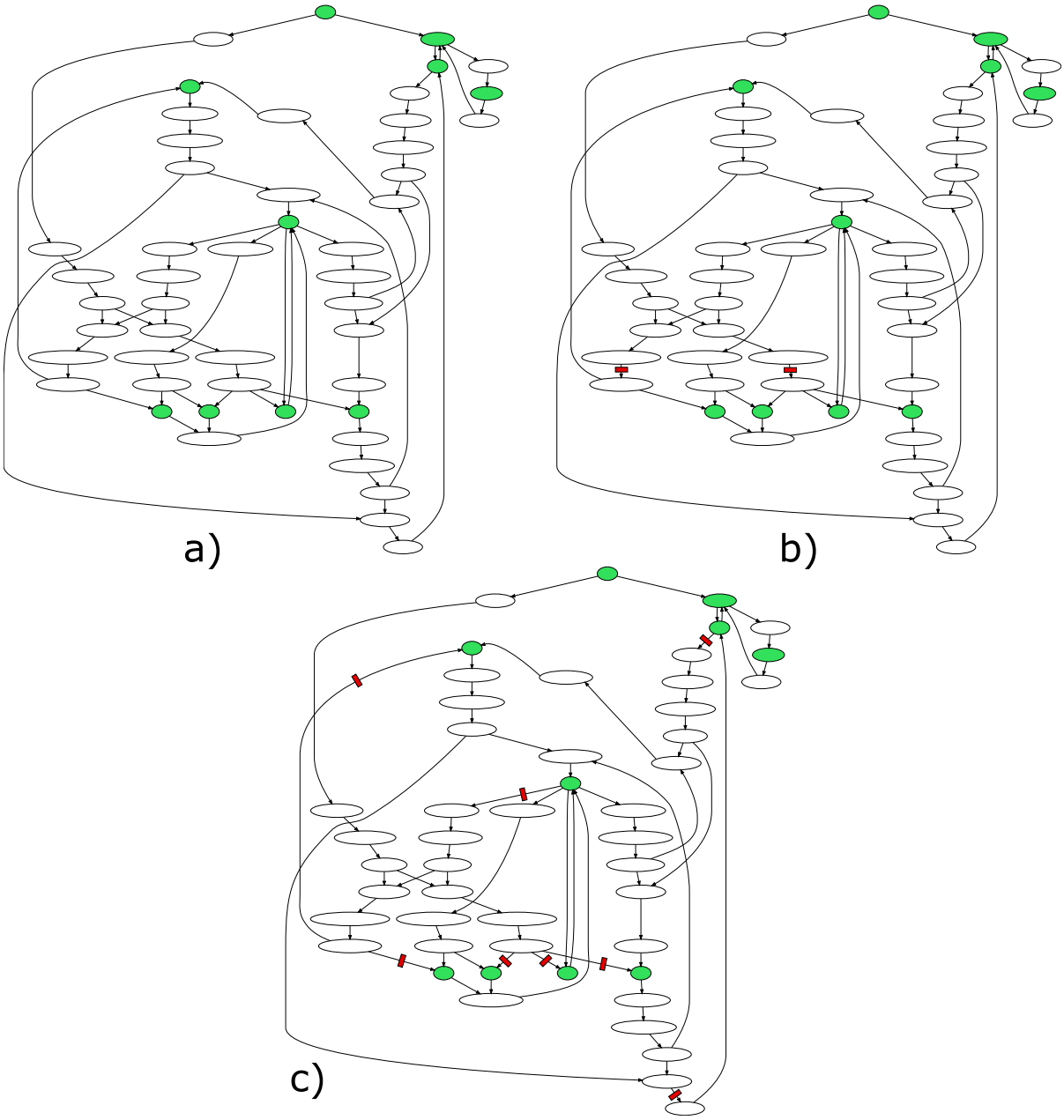

Figure 8.11: Compute Element system netlist with three levels of automatic pipelining: **a)** $D_{max} = 10$, **b)** $D_{max} = 5$, **c)** $D_{max} = 3$. User functional modules are shown in green, and inserted register stages in red.

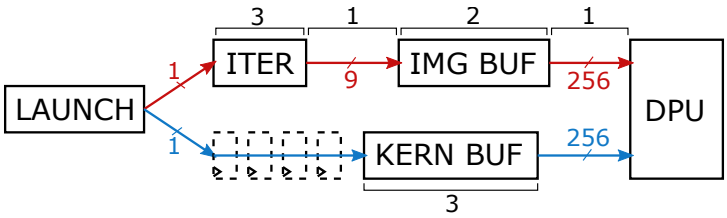

Figure 8.12: The paths taken by image data (top, red), and kernel data (bottom, blue) to reach the inputs of a single DPU. Also shown are bit widths of each logical link, and the fixed latencies of design modules and interconnect. Four extra registers are required on the kernel path to achieve balance.

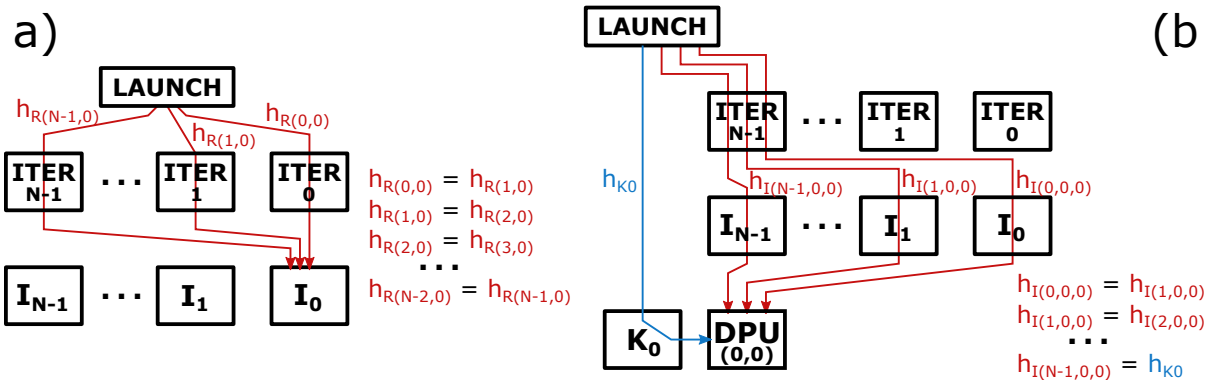

Figure 8.13: Chains and synchronization constraints used to ensure simultaneous input and kernel data delivery to DPUs. **a)** Chains h_R and related constraints used to equalize delays from all image iterators to a single image buffer I_0. There are N similar sets of these for each of the remaining image buffers. **b)** Chains h_I represent complete image data paths to a DPU, but through a limited subset of all $N \times N$ possible such paths to each DPU. Constraints force equal delays among them and also with kernel data chain h_{K0} to ensure the overall synchronization goal.

one between the iterators and image buffers, and one between the image buffers and DPU columns. At the top of each DPU column is a split node that broadcasts to all 16 DPUs within that column. The launch signal is similarly broadcast, via a split node, to all kernel buffers and all iterators.

- **Pipelining:** Also manually-specified for this demonstration. We place one pipeline stage within each of the two full crossbars, matching the implied interconnect delays in Figure 8.12. Additionally, to make the synchronization problem more interesting and challenging, we also manually systolically pipeline the row- and column- wide data broadcasts such that each row/column of DPUs receives its vertically/horizontally-broadcasted image/kernel data one cycle later than the previous row/column.

Figure 8.14 displays the resulting array. The automatically-inserted synchronization registers (in yellow) exist to balance the delays caused by the manually-specified pipeline registers (in white) as well as the pipeline latency inherent to each functional module. Several interesting phenomena can be observed here. First, the common latency difference between all kernel and all image paths (4 cycles) has been realized as a chain of four consecutive synchronization registers inserted at the very start of the kernel data paths, where it is cheapest to do so, as the launch control signal is a single bit wide. The one cycle systolic difference between each row's image data broadcasts is compensated for with additional, cheap, 1-bit-wide registers that offset every subsequent kernel buffer. Note that the matching systolic

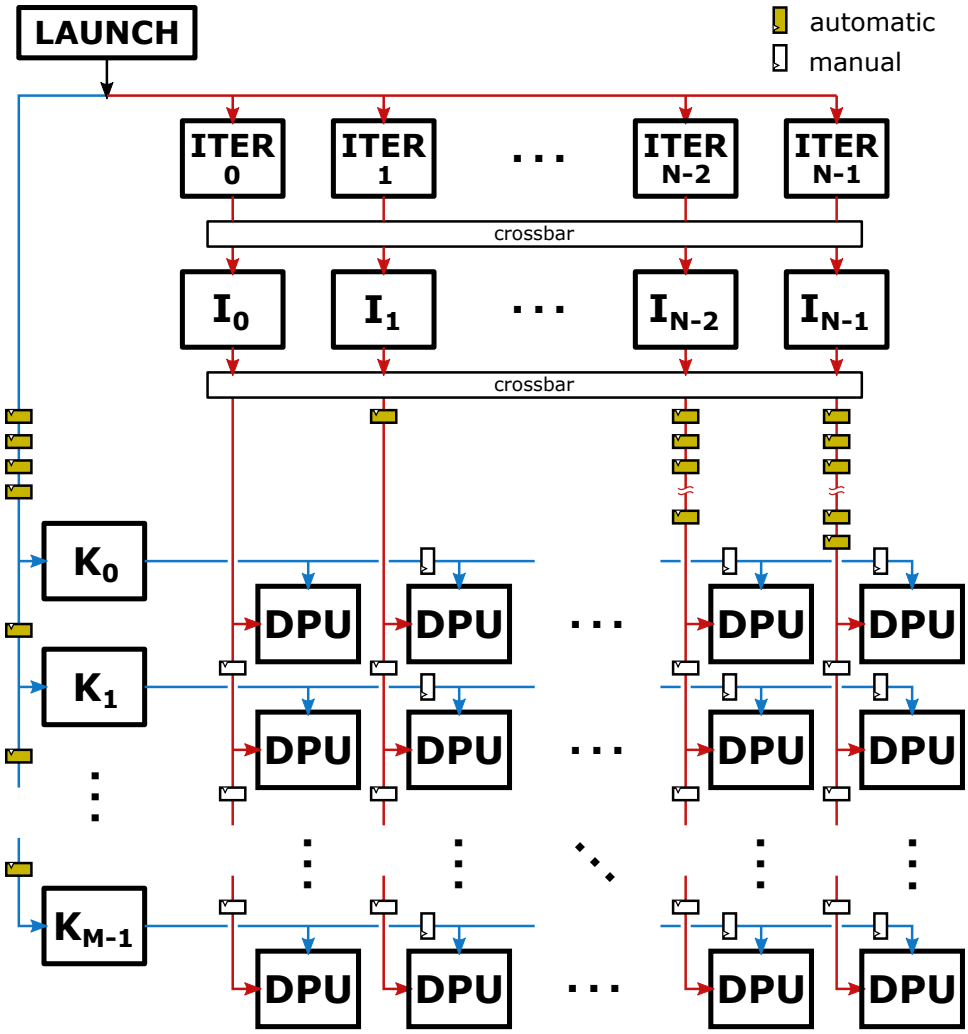

Figure 8.14: DPU array resulting from the application of the synchronization constraints. Automatically-inserted balancing registers are shown in yellow, and manually-specified pipeline registers are in white. Each crossbar also contains a single pipeline register.

appearance of these registers was automatically created due to the Systolic Retiming Transform described in Section 8.3.1. Without it, the i^{th} kernel buffer would have had an independent i-register chain between it and the launch signal, with $O(M^2)$ total such registers, where M is the number of kernel buffers.

Due to the crossbars present in the image data path, the cheap compensation method that was performed for the kernel data is not available for the image data paths, as every register that delays the launch signal would affect *all* columns. Instead, the ILP solution inserts synchronization registers later in the image data paths, where every physical link is 256 bits wide, after the second full crossbar. Every column receives an additional cycle of delay compared to the previous column. After the first column, it becomes cheaper to implement the delay using a Memory Delay primitive rather than a chain of actual registers, so here we see the invocation of the other post-processing optimization, described in Section 8.3.2. This optimization alleviates some of the otherwise-egregious area cost that would exist for a chain of N 256-bit-wide synchronization registers for large N.

Finally, it is important to ensure that this result produced by GENIE is actually functionally correct.

Table 8.2: Clock frequency and interconnect area usage for CNN design generated with all-automatic pipelining subject to D_{max} values of 5 and 3.

	N=4		N=6		N=8		N=11	
	D5	**D3**	**D5**	**D3**	**D5**	**D3**	**D5**	**D3**
FMAX	313	319	261	260	220	211	167	174
ALM	2805	2371	4156	4991	7909	10165	15789	34915
COMB	3961	2934	4101	4307	7702	9922	19107	17366
REG	2710	6925	4061	7317	5407	30546	7124	74695
MEM	10	10	10	10	10	10	10	10

We ensured correctness through RTL simulation in ModelSim 10.4c, by validating the simulated CNN output against a golden output generated by a software model.

8.4.4 Automatic Pipelining: CNN

We now return to automatic pipelining, but applied to the CNN design. With the LU design, we successfully demonstrated the ability to increase achieved clock frequency by increasing the amount of automatic pipelining, at the natural cost of increased area for the additional registers (Section 7.2.3). With the CNN design, we will also test the limitations of our logic-depth-based automatic pipelining approach. Specifically, we will compare CNN systems that are completely automatically pipelined against those that start to include manually-specified pipelining in more and more sections of the design. The regions chosen to manually pipeline are those that designer intuition would indicate have a good chance of relaxing the pressure on the FPGA CAD placement and routing engine and compensate for long signal propagation distances, therefore increasing achieved clock frequency beyond what an approach solely based on logic depth can offer.

In all experiments, the synchronization constraints described previously are also enabled, as they are required to end up with a functionally-correct design. Some of these automatically-inserted registers will compensate for the delays introduced by the manual and/or automatic pipeline registers that are the focus of the experiments, further demonstrating the interplay and dependence of the two GENIE flows described in this chapter. To limit the complexity of our analysis, a consistent manually-specified interconnect topology is used throughout, whereas later, in Chapter 9, it too will be allowed to vary and be automatically generated by the tool.

Fully Automatic Pipelining

First, we begin by observing what happens when we allow GENIE to automatically-perform pipelining based on logic depth alone. We utilize two different values of the maximum logic depth, D_{max}: the default of 5, and a more aggressive value of 3. The results can be observed in Figure 8.15, which shows where the automatic pipeline registers are inserted. Here we see part of the CNN's computation array corresponding to the image data paths, containing the N image address iterators and N image data buffers, each of which feeds a column of 16 DPUs. Only this part of the design is shown, as these are the only locations where pipeline registers were required to be inserted to satisfy maximum logic depth constraints. Additional compensation registers were also automatically added within the kernel data paths, due to satisfaction of synchronization constraints, and are not shown. The two Split/Merge layers form two full crossbars, which are simplified for illustration.

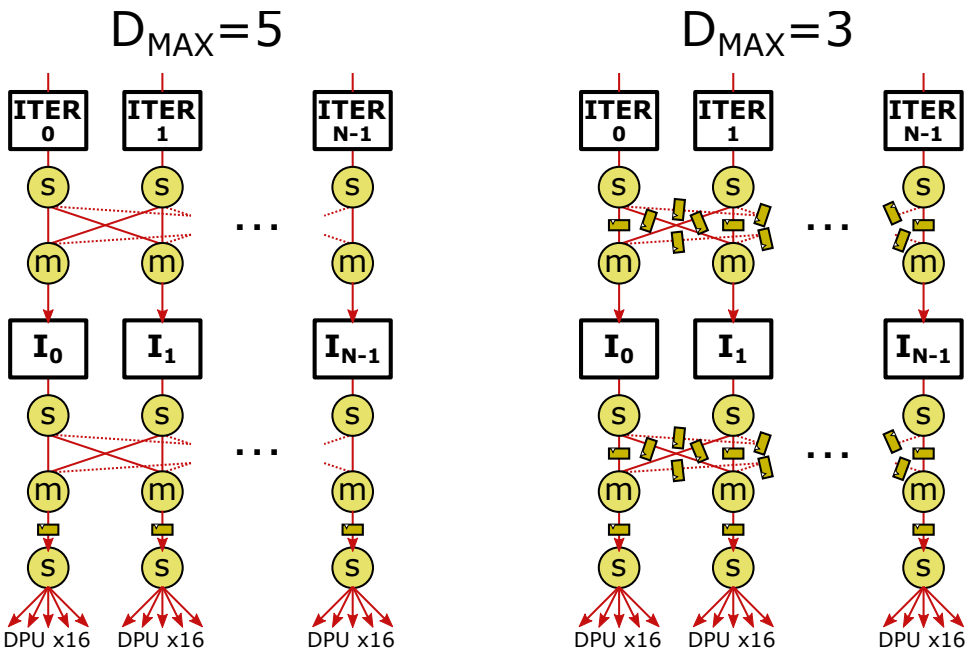

Figure 8.15: Locations of automatically-inserted pipeline registers for the CNN design under all-automatic pipelining, subject to D_{max} values of 5 and 3.

We can see that under $D_{max} = 5$, a single stage of N 256-bit registers were added between the second full crossbar and the final Split node. Under $D_{max} = 3$, these registers remain, but an additional two sets of N^2 registers are inserted in between the Split and Merge nodes of each full crossbar. These are costly locations to insert pipeline stages, but this is the only available solution for achieving a D_{max} requirement of 3 LUT stages. The associated effects on clock area and frequency are given in Table 8.2, which sweeps four different CNN array sizes, parameterized by $N = 4, 6, 8$ and 11. The results provide the achieved clock frequency (FMAX), and the number of ALMs, combinational LUTs, registers, and distributed memory ALMs used in clock domain conversion or delay elements. The area numbers are for all GENIE-generated interconnect in the design, not just the image data path section shown in Figure 8.15. Notably, the number of registers increases significantly under $D_{max} = 3$ versus $D_{max} = 5$, in the super-linear fashion expected from the pipelining of the N^2 crossbars. However, the clock frequency is not significantly affected by the extra pipelining: it changes by at most 4% across the four system sizes.

Semi-Manual Pipelining

A designer may be satisfied with the achieved clock frequency under fully automatic pipelining, however, we would like to know if even higher performance is possible by utilizing pipelining techniques that would not be available when considering logic depth alone. The CNN design can be challenging to place and route on an FPGA and achieve a high clock frequency. It contains many functional modules: image and kernel buffers, iterators, and the DPU array. Some of these require full crossbar connectivity with 256-bit data widths, and the image/kernel buffers utilize block RAM, restricting their possible placement to certain columns on the FPGA. A tool such as GENIE, which ultimately only outputs RTL, is unaware of such physical design considerations. However, a human designer may intuitively realize the inherent

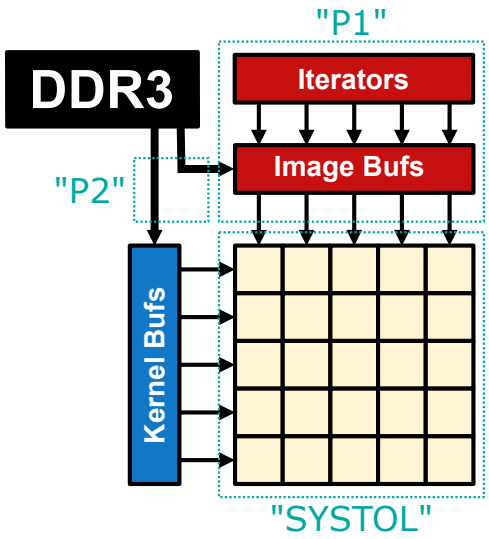

Figure 8.16: Three regions of the top-level CNN design whose interconnect will be manually-pipelined.

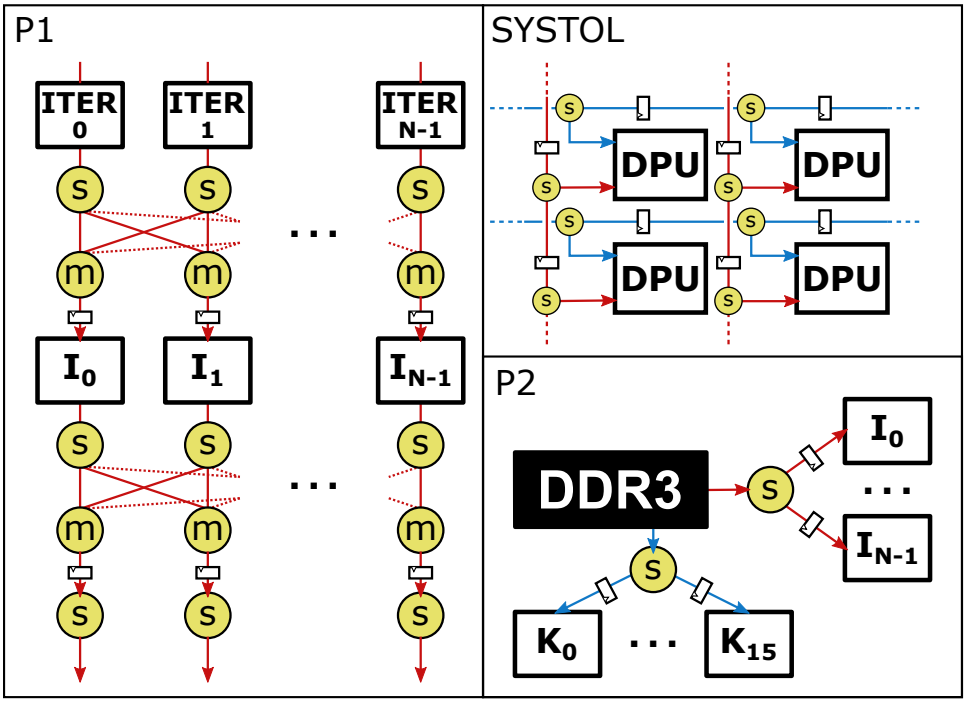

Figure 8.17: Detailed manual register placement in the three manually-pipelined CNN regions.

timing closure challenges and add pipelining for the sole reason of making it simpler for signals to cross large distances on the chip. In the next three iterations of the CNN design, we will incrementally relax GENIE's automatic control over pipelining and manually specify some pipeline register locations. This is achieved during manual topology specification, in which links between modules, split nodes, and merge nodes may have an explicitly-specified number of register stages (which further constrain the ILP solution at the granularity of physical links).

Figure 8.16 illustrates the regions we will manually pipeline, and Figure 8.17 shows exactly where the

Table 8.3: Achieved clock frequency and area for all automatic and manual pipeline schemes across four sizes of CNN system.

N=4	Fmax	ALM	COMB	REG	MEM
D5	313	2805	3961	2710	10
D3	319	2371	2934	6925	10
SYSTOL	331	9258	3850	30879	400
SYSTOL + P1	339	9289	3851	30948	400
SYSTOL + P1 + P2	375	10541	3832	36309	400

N=6	Fmax	ALM	COMB	REG	MEM
D5	261	4156	4101	4061	10
D3	260	4991	4307	7317	10
SYSTOL	283	15054	4022	47749	660
SYSTOL + P1	287	14727	4025	48049	660
SYSTOL + P1 + P2	320	16851	5035	53861	660

N=8	Fmax	ALM	COMB	REG	MEM
D5	220	7909	7702	5407	10
D3	211	10165	9922	30546	10
SYSTOL	261	23104	9286	64999	920
SYSTOL + P1	267	23200	9256	64959	920
SYSTOL + P1 + P2	279	24465	9236	71663	920

N=11	Fmax	ALM	COMB	REG	MEM
D5	167	15789	19107	7124	10
D3	174	34915	17366	74695	10
SYSTOL	185	36386	19306	90190	1310
SYSTOL + P1	195	35887	18847	90606	1310
SYSTOL + P1 + P2	200	36941	19407	97700	1310

registers will be placed in each region. Region P1 is the image data delivery section that was previously the only area that was automatically pipelined under both $D_{max} = 5$ and $D_{max} = 3$. The very regular two-stage pipelining proposed for region P1 is more aggressive than $D_{max} = 5$ was (which only had one stage), but avoids pipelining the N^2 connections within each full crossbar like $D_{max} = 3$ did. It is also possible that with too many registers, the design actually becomes harder to place and route, so in our manual scheme for P1 we only place pipeline registers after the output of each merge node, yielding $2N$ total registers versus $2N^2 + N$.

The "SYSTOL" region takes place within the compute array, where we manually pipeline the delivery of kernel and image data in a systolic fashion. This arrangement was previously used to stress-test the synchronization constraint functionality in Section 8.4.3. The intuition behind placing pipeline registers in the array is to provide more slack to distribute the long and wide broadcast data to the 2D array of DPUs.

Finally, the P2 region exists between the off-chip memory and the kernel/image buffers. These are the links that periodically fill the on-chip buffers as their contents are processed. This is significant, as these links are on a separate clock domain from the one that we have been measuring and the one on which we hope to increase clock frequency. Intuitively, without the proposed manually-placed registers, there would be a fully-combinational high-fanout broadcast, which would restrict the placement of the kernel and image buffers.

The results are displayed in Table 8.3. Each quadrant is a different system size, parameterized by a different value of N. Within each quadrant, all pipelining configurations (including the first two fully-automatic pipeline schemes) are included, giving clock frequency and area measurements. The three manual configurations add an ever-increasing amount of manual pipelining, starting with just the "SYSTOL" region, and then incrementally adding "P1" and then "P2". Automatic pipelining, under a maximum logic depth of 5, is still applied under the SYSTOL-only configuration, before it is completely overridden after the introduction of the P1 manual pipelining.

All three installments of manual pipelining achieve a higher clock frequency than either of the automatic-only D5 and D3 configurations. When combining SYSTOL, P1, and P2, this yields a 15-25% improvement over the automatically-pipelined D3 configuration. SYSTOL alone adds a significant

amount of registers to achieve the first clock frequency bump, so it is tempting to conclude that this is simply a case of area/performance tradeoff – throwing more registers at the problem than automatic pipelining would normally perform on its own. However, observing the gain seen when adding the P2 manual pipeline region, we can see that there is more happening: adding extra registers in an *unrelated clock domain* gives us up to 10% better clock frequency in the compute domain.

What we can conclude from this is that there is much room for improvement in automatic pipelining in terms of considering physical design characteristics beyond simple combinational logic depth. The ideal place for an automatic pipelining algorithm would be *during* FPGA place and route, where such information is known to an exact degree. Intel's Stratix 10 [50] FPGAs and accompanying Quartus Prime software are an example of such an approach, where pipelining is performed during placement and routing, with the aid of special registers embedded within FPGA interconnect links.

Chapter 9

Automatic Topology Optimization

This chapter describes a method of automatic topology generation and optimization within the GENIE interconnect synthesis flow. It is based on the short paper *Automatic Topology Optimization for FPGA Interconnect Synthesis* [65].

Recall that GENIE's interconnect microarchitecture uses two primitives to perform all routing: the Split node and the Merge node. Different arrangements of Split and Merge nodes can be composed to create different interconnect topologies. This set of possibilities enables tradeoffs between area usage and performance of the resulting system. GENIE partitions the user's logical connectivity into connected components called domains, and each domain may either have a manual topology (in which Split and Merge nodes are explicitly instantiated and connected by the user), or an automatically-generated and optimized topology. The automatic topology generation process is guided by information, provided by the user, about the transmissions represented by the logical links in a domain, with the goal being to create a topology that satisfies the user's communication requirements while minimizing interconnect area usage. We will describe the nature of these transmission communication specifications as well as the topology creation and optimization process. Results will be provided that demonstrate successful and unsuccessful use cases of automatic topology optimization within real designs. First, we will begin by motivating our need for automatic topology creation and optimization.

9.1 Motivation

Routed Streaming links, defined by the user, represent logical transmissions that must traverse GENIE-generated physical interconnect to arrive at their destinations. The topology of a GENIE interconnect domain constrains the available paths that transmissions can take, and defines the maximum available throughput. Specifically, Merge nodes are the locations within a network topology where one physical link (the Merge node's output) must be shared among all of the Merge node's inputs.

Figure 9.1 illustrates three different topologies that connect two source interfaces X and Y to four sink interfaces A, B, C, and D. The leftmost topology is the most complex, using four merge nodes and two split nodes, but it provides the maximum possible throughput. In contrast, the rightmost topology uses the fewest routing primitives, but forces all transmissions to share the central link connecting the Merge node to the Split node. Going from left to right, there is a trend of reduced resource usage at the cost of reduced maximum throughput.

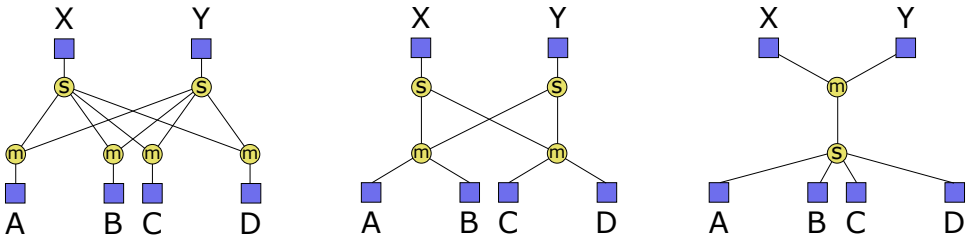

Figure 9.1: Three different topologies that permit routing of transmissions from sources X and Y to sinks A, B, C, and D.

However, the effect of topology on performance is highly dependent on the transmissions themselves. If, for example, sources X and Y sent short infrequent bursts of data to one out of the four sinks in Figure 9.1, then all three topologies would perform equally, and it would be advantageous to simply choose the one with the smallest area. Similarly, if X rarely communicated with C or D, and Y rarely with A and B, then the center topology would provide nearly as good performance as the fully-connected case, but with less area. Nevertheless, with a sufficient number of simultaneous transmissions, any simplification of the fully-connected topology on the left would result in reduced performance. Provided with knowledge about an application's transmissions and their demands on the interconnect, it should be possible to make intelligent topological decisions.

Our goal will be to allow GENIE to automatically construct topologies that minimize area usage while simultaneously respecting performance constraints provided by the user. This will require additional inputs from the user to specify these constraints, as well as modifications to the overall GENIE synthesis flow.

9.2 Transmission Specifications

There are three types of topology-related optional specifications that the user may annotate Routed Streaming logical links with. These provide GENIE with information about transmissions' performance requirements and are used to guide the topology synthesis and optimization process.

The first type of specification is *transmission mutual exclusivity*. This is a guarantee from the user that the transmissions represented by two or more RS links will never occur at the same time, or overlap temporally in any way. The user knows this because they are responsible for the design of their functional modules, and therefore also for the initiation of transmissions into the interconnect. If GENIE is also provided with this knowledge, it will know that such non-conflicting transmissions will not incur any penalty or throughput degradation if they are made to share a Merge node. Additionally, if all transmissions through a Merge node are non-conflicting, a simpler and less-costly Merge node can be instantiated that lacks complex arbitration circuitry (see Section 4.2.3).

The second specification is *importance*, which is an optional value that can be attached to each transmission by the user. A transmission's importance is a dimensionless value between 0 and 1 indicating how much the user desires that that transmission avoid contention with other transmissions. A value of 1 indicates that this transmission should encounter as little contention as possible, and a value of 0 means that the user does not care how much contention it encounters. Since contention results in throughput loss, importance becomes a way of specifying throughput requirements, relative to a "best case" that we will described in Section 9.3. By specifying importance, the user gives GENIE

permission to reduce performance (up to a point) in exchange for potential area savings via the sharing of interconnect resources. The default value of a transmission's importance is 1, meaning that when generating topologies, GENIE will not sacrifice any performance.

The third specification is the transmission's *packet length* in clock cycles. This is the total size of a transmission in bits, divided by the interface's data width, and rounded up. If left unspecified, the default is 1. RS interfaces use the EOP (End-of-Packet) signal to dynamically specify the length of a transmission, but GENIE cannot know that information at system generation time, hence the need for this new specification, which would then represent a worst (longest) case. By knowing a transmission's packet size, the topology optimization process can be more aware of the performance impact resulting from transmissions being forced to share interconnect resources. The details of how contention between transmissions is modeled will be described in Section 9.4.

9.3 Topology Generation and Optimization Flow

Figure 9.2 is a high-level overview of the entire GENIE flow, as first presented in Section 3.4. It illustrates where topology generation and optimization fits in – as an iterative loop performed only for interconnect domains that do not have a user-overridden explicit manual topology. The input to the automatic topology flow is the functional modules and logical RS connectivity for just one domain, and this includes all annotations and constraints (such as importance) associated with the logical RS links. A 'topology' in the specific context of this flow is then just the same interconnect domain but with Split and Merge nodes added in, along with physical links connecting the Split, Merge and user functional modules. The box labeled 'Inner Flow' is responsible for adding additional interconnect primitives in between the Split and Merge nodes and is where the bulk of GENIE's automating and optimization capabilities perform their work. This transforms a topology into a fully functional netlist for that domain, suitable for final HDL output. However, *many* possible topologies will be explored and iterated upon to attempt to find the one with lowest cost while still satisfying the user's performance requirements. A key part of this process is estimating the area cost of each topological choice, and belongs in the "Estimate Area" step that is performed after a topological candidate is run through the Inner Flow. The remainder of this section details the different phases of the topology generation and optimization flow.

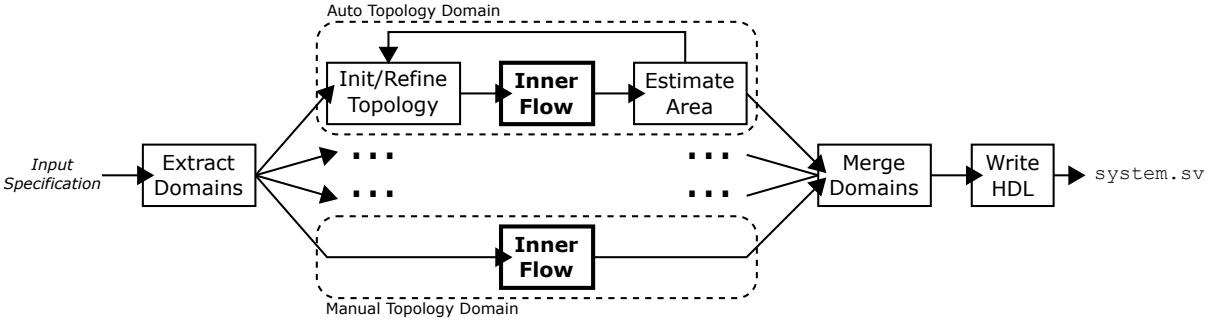

Figure 9.2: GENIE Outer Flow

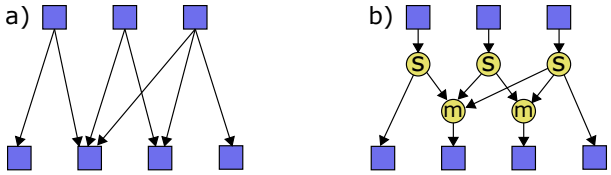

Figure 9.3: **a)** A set of logical connections between the interfaces of functional modules within an interconnect domain. **b)** The logical connectivity realized as a sparse crossbar topology using Split nodes, Merge nodes, and physical connections.

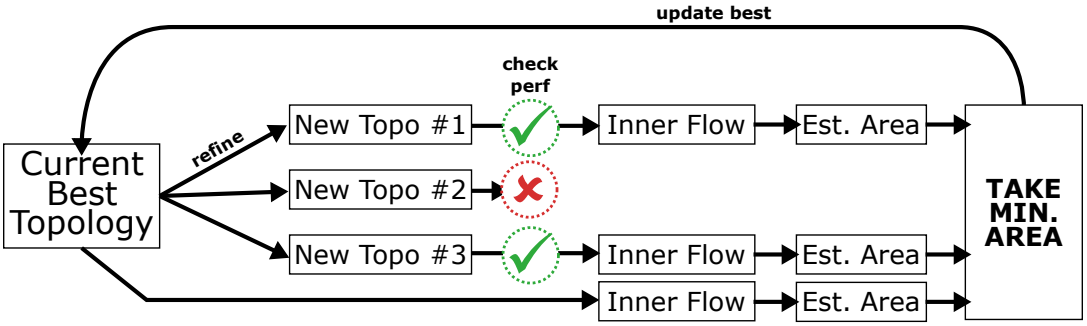

Figure 9.4: The topology optimization loop

9.3.1 Initial Crossbar Topology

The logical connectivity of the domain is initially mapped to a sparse crossbar implemented using Split and Merge nodes, an example of which is shown in Figure 9.3. A sparse crossbar is used as a starting point because it offers the least possible contention and greatest possible throughput, should that be the user's desire. The initial crossbar topology is used as a baseline for area and performance, and all further topologies will be created by incrementally simplifying this crossbar.

9.3.2 Topology Optimization Loop

After the initial crossbar topology is created, it becomes the "current best" topology and enters the optimization loop, which is illustrated in Figure 9.4. This is a more detailed view of the loop than in Figure 9.2. From the current best topology, new topologies are generated by performing a single *refinement step*. This combines two existing Merge nodes into a single Merge node followed by a Split node (Figure 9.5). By reducing the total number of Merge nodes by one, area consumption may decrease, but contention between transmissions may increase, reducing performance.

One refinement step only combines two Merge nodes, and every existing pair of Merge nodes is considered, independently. This yields many possible new candidate topologies, all derived from the current best topology. Candidates that fail to meet performance requirements are rejected, but those that pass are processed by the GENIE Inner Flow to become complete interconnect implementations. The area of each candidate topology's full implementation is estimated, and the one with the lowest estimated area is chosen to become the new "current best" topology. This loop continues until either: no further refined candidate topologies can be found that meet performance requirements, or, no candidate topologies have an estimated area lower than the current best. This current best topology then becomes the output of the loop.

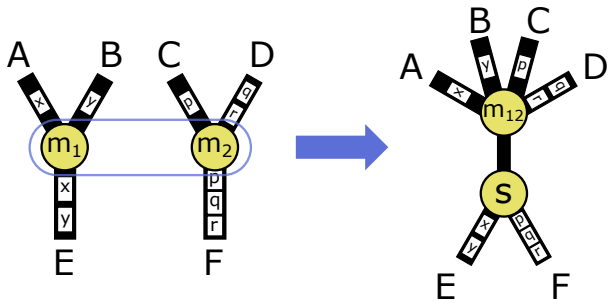

Figure 9.5: A topology refinement step, which combines two Merge nodes and adds a Split node. The transmissions (x,y,p,q,r) routed over physical links (A-F) now face greater contention.

The initial crossbar topology and its subsequent refinements yield a limited space of possible topologies achievable with this method. For example, rings and meshes are types of topologies can not arise from GENIE's automatic topology flow, although they are still possible to create via *manual* topology specification. The method is also inherently greedy, and may arrive at a local rather than global minimum. An exhaustive traversal of the search space would yield all the possible partitions of the set of merge nodes in a domain, where each partition corresponds to a set of merge nodes that would ultimately be combined into a single merge node via a refinement step similar to Figure 9.5. Given m merge nodes, there are $B(m)$ such partitions, where $B(m)$ is the m^{th} *Bell Number*, for which the tightest known asymptotic bound is currently $\mathcal{O}\left(\left(\frac{0.792m}{ln(m+1)}\right)^m\right)$ [10]. Our greedy approach, in contrast, has complexity $\mathcal{O}(m^3)$. The runtime performance will be measured in Section 9.7.5.

9.4 Contention Model

In order for a candidate topology to be deemed acceptable and continue through the optimization loop, it must meet the user's performance requirements. To formalize these requirements, we must first discuss how the performance impact of topological decisions is quantified. This is done by modeling *contention* between transmissions as they enter a shared Merge node. Contention is a worst-case upper bound on the number of clock cycles that a transmission must wait at the input of a Merge node before its round-robin arbiter circuit grants it permission to proceed. The transmission is then allowed to proceed in its entirety, which is indicated by its end-of-packet (EOP) signal. In the worst case, a transmission will be forced to wait for all other contending transmissions to finish passing through the Merge node, being delayed for a number of clock cycles equal to the sums of the contending transmissions' packet lengths. The packet lengths are annotated explicitly by the user, and should ideally match the actual runtime lengths of the transmissions as demarcated by their EOP signals.

To define contention more formally, consider two transmissions t_x and t_y. They *contend* if all the following are true:

- There exists a Merge node M with two physical interconnect links P_x and P_y as inputs, such that P_x carries transmission t_x and P_y carries t_y.
- Any physical link P_z that carries *both* t_x and t_y is downstream of M.
- t_x and t_y are not explicitly marked as mutually-exclusive by the user.

The second condition is equivalent to saying that if t_x and t_y have already contended before reaching M, they can not contend at M. Next, we define the *incremental contention* of transmission t_x *due to* transmission t_y as:

$$R(t_x, t_y) = \begin{cases} L(t_y) & \text{if } t_x, t_y \text{ contend} \\ 0 & \text{otherwise} \end{cases} \tag{9.1}$$

where $L(t_y)$ is the user-specified packet length of t_y. Finally, we obtain the total worst-case contention experienced by a transmission t_x by summing its incremental contention due to every other transmission:

$$C(t_x) = \sum_{t_y}^{t_y \neq t_x} R(t_x, t_y) \tag{9.2}$$

9.5 Criteria for Acceptable Performance

Having defined contention, we can resume explanation of the criteria by which the topology optimization flow accepts or rejects a candidate topology produced by a refinement step. This will depend on the contention experienced by transmissions in the new topology, which in turn depend on importance, packet lengths, and explicit mutual exclusivity of transmissions as annotated by the user. Recall that the importance of a transmission t, $I(t)$, is a value between 0 and 1 with 1 reflecting the user's desire for maximum performance for t. Since a sparse crossbar ensures minimal contention, it is used as the baseline for defining the throughput experienced by a transmission when its importance is set to 1.

Let $C_0(t)$ be the total contention experienced by transmission t when sent through a sparse crossbar topology, from its source to its destination. A fresh topology x produced by a refinement step will potentially have a different contention, $C_x(t)$, for t. That topology x is deemed to have acceptable performance if the following is true:

$$\frac{C_0(t)}{C_x(t)} \geq I(t) \quad \forall\ t \tag{9.3}$$

The condition in Equation 9.3 requires that every transmission have a total contention in the new topology that is no worse than in the sparse crossbar topology, up to a factor equal to the transmission's importance. There is elegance to this construction: in one extreme, if all transmission importances are set to 1, then only topologies that are as good as the crossbar will be acceptable. At the other extreme, with importances set to 0, all possible topologies are acceptable, up to and including ones consisting of a single Merge node. Levels of importance moving down from 1 will give rise to more contention in exchange for lower cost. The results given in Section 9.7.2 will illustrate the trade-off enabled by various values of importance.

9.6 Area Modeling

The final key component of the topology optimization flow is the ability to rapidly estimate the area impact of topological decisions. This can only be done after a topology has been elaborated into a full interconnect implementation by the Inner Flow, as it is not enough to simply count the number of Split or Merge nodes in a topology. This is because a topological change, such as the addition/removal of Split or Merge nodes, or even changing the number of inputs/outputs on existing Split/Merge nodes, can have cascading downstream effects on area consumption as a result of the various processing stages and optimizations performed within the Inner Flow. As an example, Split and Merge nodes will change in combinational logic depth as their internal complexity changes with the addition/removal of inputs or outputs. This will lead to a different numbers of registers to be inserted by the automatic pipelining stage of the Inner Flow. Counter-intuitive area effects can also occur: two 2-input Merge nodes may require less area than a combined 4-input Merge node *if* the inputs to each 2-to-1 Merge node have registers preceding them, causing the combinational logic to vanish and be implemented with SLOAD/SDATA signals (Figure 4.7). The area models described in Chapter 4 take into account many such parameterizations of interconnect primitives, which depend on work performed in the Inner Flow that is ultimately dependent on the topology.

GENIE uses a single score to estimate its generated interconnect area, and this is equal to the sum of combinational LUT and register usage, summed over every primitive. As future work, the user could elect to prioritize LUT or register usage, which would introduce weighting factors to the area score calculation.

9.7 Results

In this section, we will evaluate our topology optimization approach using the design examples from Chapter 5. There exist two types of useful features that are made available through the use of topology optimization:

- The user specifies mutual temporal exclusivity between transmissions, resulting in a simplified topology with lower area usage and **no performance loss** relative to the default crossbar topology.
- The user specifies a reduced importance (less than 1) on transmissions that *do* conflict, yielding a topology that **loses some performance in exchange for reduced area usage**.

Both situations will be investigated as two separate experiments, using the LU factorization application described in Section 5.1. Specifically, we examine the coarse-grained outer design rather than the internal fine-grained design of each Compute Element.

The application will be parameterized to have $N = 16$ compute elements and $M = 4$ memory controllers. Additionally, there is the central Control Node that coordinates the CEs and aggregates Left Block Broadcast read requests, as previously described in Section 5.1.2. In that same section, we described the three types of memory traffic that occurs between the CEs, memory controllers, and Control Node:

- Read requests from CEs to MEMs, CEs to CTRL, and CTRL to MEMs: one flit containing address information (20 bits)

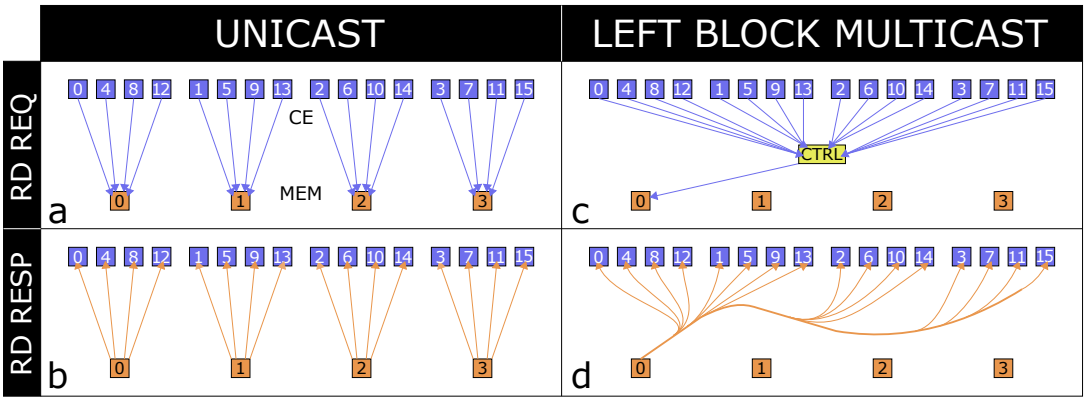

Figure 9.6: **a/b)**: Unicast read requests and read responses. **c)** Read requests for left blocks aggregated by the Control Node and forwarded to a single memory controller. **d)** Left block contents are multicasted from a memory controller to all CEs.

- Read replies from controllers to CEs: 512 flits of read data (272 bits each);
- Write requests from CEs to controllers: 512 flits of write data (272 bits each)

Of these three, the first two present the most interesting use cases for topology optimization and area/performance trade-offs, so we will focus on them for the evaluation of our flow. Recall that the read request and read reply transmissions occur in both unicast and multicast variants, with the multicasts used for delivering Left block data in an efficient manner. These two types of memory reads are summarized in Figure 9.6, which is a copy of Figure 5.4.

For unicast traffic, the 16 CEs are divided into groups of four and each group is associated with one of the four memory controllers. The CEs issue read requests to the controller associated with their group (Figure 9.6a) and receive the corresponding replies (9.6b). For Left block reads, the 16 CEs issue their requests to the control node, which then forwards a single copy of the request to ONE of the four controllers (9.6c). Then, a *single* reply transmission is broadcast back to all of the CEs (9.6d). Controller #0 is used as an example in the figure, but three other similar possibilities (for each of the remaining controllers) also exist.

If one were to overlay all possible read request (or read reply) transmissions together as a single picture, it would result in all-to-all connectivity: every memory controller is accessed by each CE at some point in time. However, there are properties concerning the timing and importance of the transmissions, which if captured and specified by the user, could potentially yield optimizations to the interconnect design, saving area over the default full crossbar needed for all-to-all connectivity while not impacting overall performance in a significant way.

The first observation we make is that read requests are small in cycle length (one flit) compared to their associated read replies (512 flits). Delaying the arrival of read requests at memory controllers should minimally impact application performance if the controllers are throughput-bound by the read replies. Secondly, the four possible multicast replies (such as the one shown in Figure 9.6) can never occur simultaneously. That is, only one of the four memory controllers is ever broadcasting a reply at any given time.

We will use these two observations to create importance, packet length, and mutual exclusion specifications for this application, and create two separate experiments to test our topology optimization

flow's ability to take advantage of these application-specific communication requirements.

9.7.1 Experimental Description and Methodology

Our first experiment will test the algorithm's ability to trade off performance and area by varying the importance of transmissions. Recall that in this application, the arrival of read requests at memory controllers is not time-sensitive, and a delay of several cycles should not adversely impact total application performance. This will be tested by assigning all read request transmissions the same importance, and varying that value from 1 down to 0, while measuring the simulated application runtime (in clock cycles) and interconnect area after a full run of FPGA synthesis, placement, and routing tools.

In the second experiment, we will exercise the ability to specify mutual exclusivity, specifically for the read response network. We wish to see if the designer's extra knowledge about the mutual exclusivity between the four read response transmissions can be leveraged to reduce interconnect area with our automated flow. To test this, we will specify, or omit, the necessary mutual exclusivity annotations from the system input specification.

For both experiments, packet length specifications $L(t)$ are provided for all transmissions t. GENIE is used to generate the full system and interconnect, and this requires approximately 2 seconds on an Intel Xeon E5-2643. The resulting SystemVerilog output is simulated to obtain application runtime (in cycles), and synthesized with Quartus Prime Pro 17.0 for an Arria 10 FPGA (`10AX115N2F45E1SG`) to obtain area utilization and clock frequency, which was geometrically averaged across six compilation seeds. In addition to measuring the actual synthesized area of each result, we will provide the area estimated by the algorithm, to assess the quality of the area modeling.

9.7.2 Experiment 1: Effects of Varying Importance

In our first experiment, we focus on the read request subnetwork within the LU application. All read request transmissions are annotated with the same importance value, for which we use six different values: 1, 0.3, 0.25, 0.2, 0.15, and 0. Aside from the extreme values of 1 and 0, the others four were discovered manually through an initial broader sweep of values between 0 and 1. We chose the values for which some change in the generated result (topology or area) was observed. We only wish to illustrate effects on the generated system as a result of varying importance values – the ability to automatically discover inflection points, such as the ones that we found manually, is left to future work.

The differences between the resulting automatically generated read request network topologies are depicted in Figure 9.7. CEs are omitted in the figure for clarity – only the sinks of read request transmissions (the memory controllers and the control node), and split and merge nodes, are shown. As importance decreases, the five initial merge nodes (one for each sink) are combined more aggressively until only one remains at importance 0. Although the network being optimized is relatively simple, it illustrates the ability to generate new topologies in a sensible manner through the varying of importance.

Table 9.1 and Figure 9.8 provide the achieved system clock frequency, area (both estimated by GENIE and actual post-place and route), and the latency in clock cycles of the read request network, versus the chosen importance value of the read request transmissions. The area score is used by the topology optimizer is the total number of LUTs and registers, and it decreases monotonically as importance is reduced to 0. An importance of 0 yields interconnect that is 60% smaller than using an importance of 1. Note that this represents a 6.6% reduction of total interconnect area, when considering the entire

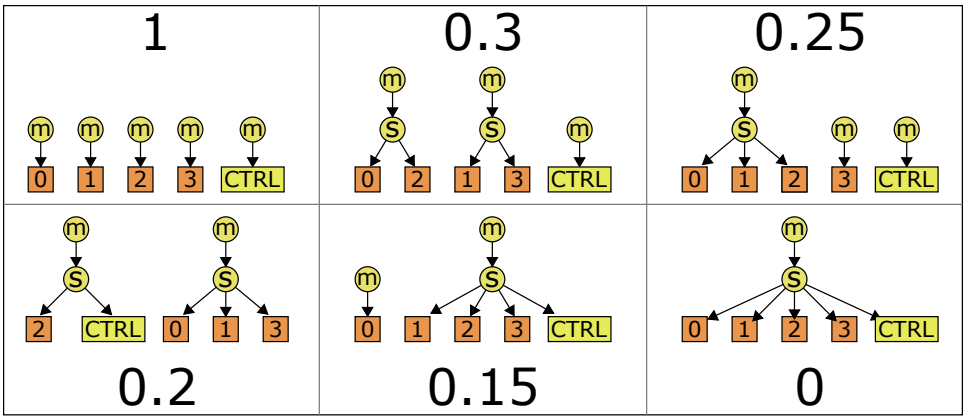

Figure 9.7: The topology of the LU read request network for six different values of transmission importance.

Table 9.1: Raw Results for Experiment 1

Importance		**1**	**0.3**	**0.25**	**0.2**	**0.15**	**0**
Fmax (MHz)		279	269	271	234	278	275
Area (Est.)	**LUT**	929	793	788	739	583	452
	REG	2111	2077	2052	1200	737	650
	TOTAL	3040	2870	2840	1939	1320	1102
Area (Act.)	**LUT**	683	595	566	622	472	402
	REG	1567	1551	1531	896	580	506
	TOTAL	2250	2146	2097	1518	1052	908
Latency (cycles)		7	11	15	15	15	19

system and not just the read request domain.

The price of this area reduction is an increase in worst-case latency, which was measured by simulating a scenario in which all 16 CEs send unicast read requests simultaneously to their preferred memory controllers, as shown in Figure 9.6a. We then recorded the clock cycle at which the last request arrived at a memory controller. An importance value of 0 caused a 2.7x increase in this worst-case latency, caused by the sharing of that topology's single Merge node. However, when the full application was simulated with real input data, and CEs were allowed to issue read requests at their natural rate, the total application runtime remained constant at 5.46 million clock cycles, deviating by no more than $\pm$ 0.02% across the six chosen values of importance (and their associated network topologies). This result illustrates that the read request network is not the critical communication path in this application, and that it is a wise design decision to locally reduce the performance of the read request network in exchange for area savings. By varying the importance values of transmissions, the user is able to automatically take advantage of such application-specific opportunities in an automated manner.

We also illustrate the correspondence of GENIE's area modeling and ground truth results obtained after placement and routing. While GENIE over-estimates the area, likely due to not accounting for synthesis optimizations, the estimates are sufficient to be able to compare the fitness of topologies in a relative manner. The over-estimation is due to our area modeling being unable to fully predict all the cross-module optimizations that Quartus is able to achieve during logic synthesis and placement/routing.

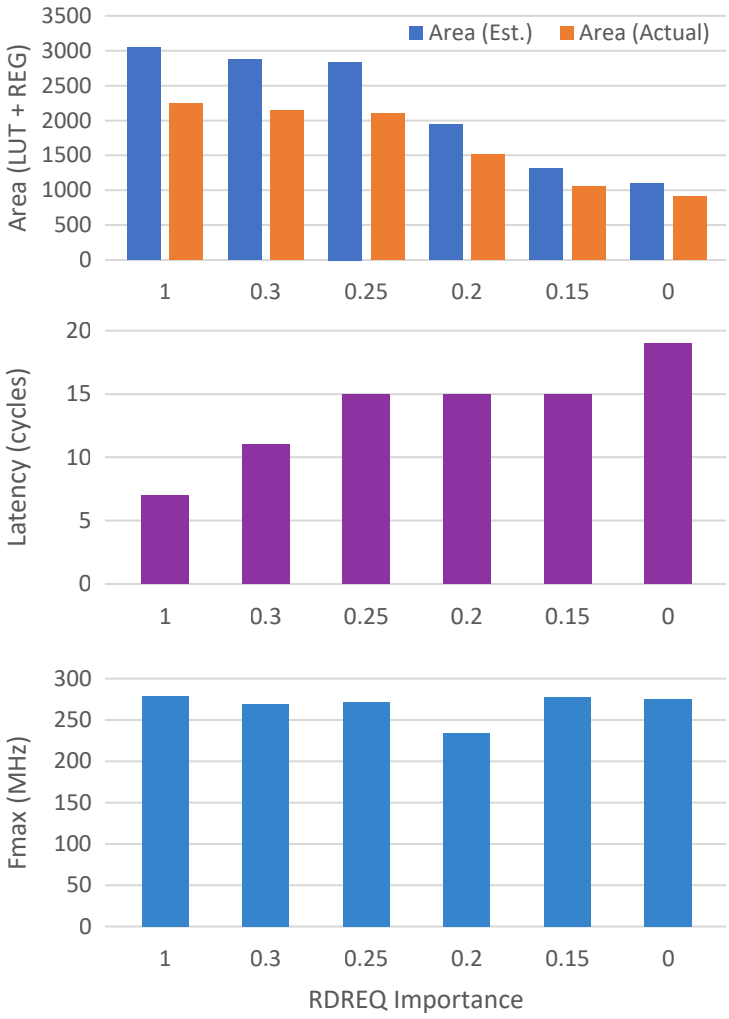

Figure 9.8: Top to bottom: area, latency, and clock frequency of the read request network, versus decreasing transmission importance.

9.7.3 Experiment 2: Effects of Mutual Exclusivity

In our second experiment, we examine the application's read response (rather than read request) network. These communication paths are relatively wide (256 bits) and expensive. As previously described, there exist two kinds of read replies: unicast (Figure 9.6b) and multicast (Figure 9.6d). If only unicast replies existed, the read reply network would naturally be segregated into four networks, one for each memory controller. However, because of the existence of the multicast replies, all-to-all connectivity is required in the general case. The four possible multicast reply transmissions can never occur simultaneously, which is application-specific behavior that, if GENIE is made aware of, could result in a simpler topology than a full crossbar.

Thus, our experiment tests two scenarios: one in which GENIE remains unaware of the mutual exclusivity of the multicast reply transmissions ('NO MUTEX') and one in which mutual exclusivity is specified ('MUTEX'). Figure 9.9 demonstrates the two resulting topologies. Rather than showing all 16 CEs, only a single cluster of four remains for clarity – the remaining 12 have a similar, symmetric connectivity as these four. In the NO MUTEX case, a full crossbar results, as expected: every CE must

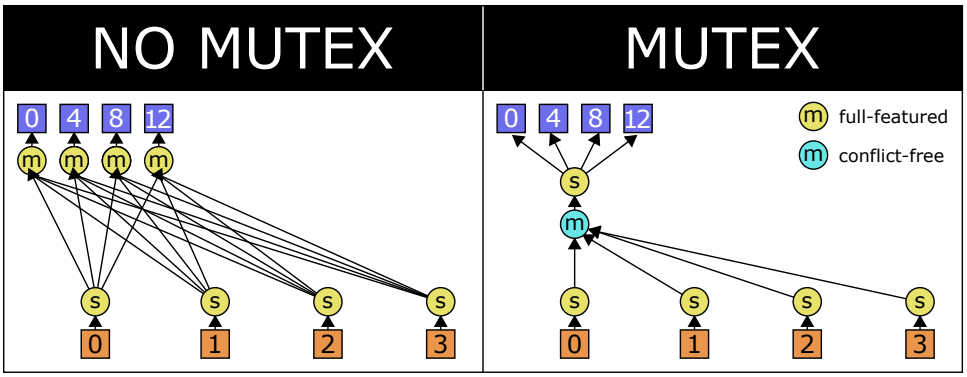

Figure 9.9: Auto-generated topology for the read response network of a 4-CE cluster, showing functional modules, split nodes, and merge nodes. Left: unoptimized (no mutual exclusivity information). Right: optimized (mutual exclusivity specified)

Table 9.2: Raw Results for Experiment 2

		NO MUTEX	MUTEX	Change
Fmax (MHz)		273	279	+2.2%
Area (Est.)	**LUT**	4988	1340	-73%
	REG	38524	9212	-76%
	TOTAL	43512	10552	-76%
Area (Act.)	**LUT**	4694	1240	-74%
	REG	37773	8835	-77%
	TOTAL	42467	10075	-76%

accept transmissions from every memory controller, yielding a 4-to-1 merge node per CE. Additionally, without knowledge of the transmissions' mutual exclusivity, GENIE instantiates standard full-featured Merge nodes that include round-robin arbiters. In the MUTEX case, the flow is made aware of mutual exclusivity between the Phase 2 read response transmissions, and the algorithm deems that the four merge nodes are allowed to be combined into a single node for the entire cluster. With the transmissions' mutual exclusivity known to GENIE, it is able to implement the lone remaining Merge node as a conflict-free variant (Section 4.2.3), further simplifying the interconnect.

The resulting area reduction within the read response network is significant – averaging 75% between LUT, register, and total area score (LUT+REG). This represents a 61% reduction of total interconnect area for the design, as the read response network is a significant component. The full clock frequency and area measurements (estimated and actual) are provided in Table 9.2. There was no measured impact on the total application cycle count (5.46 million) as a result of this optimization, which in essence, becomes an entirely free one.

9.7.4 Fine-Grained Topology Optimization

In Chapter 6, we studied the fine-grained system within the LU application's Compute Element. There, a manually-specified custom topology was employed to reduce the area of the already very small interconnect fabric. The specification of mutual transmission exclusivity was used in a limited manner, which was only to infer the generation of conflict-free Merge nodes. However, if we enable the automatic topology optimization flow, the same mutual exclusivity specifications yield an optimized topology that

Table 9.3: Total GENIE Runtime for Various Design Parameterizations

Design	Merge Nodes	Topologies Iterated	Topologies Elaborated	Runtime (s)
LU N16M4	16	676	148	1.73
LU N32M4	32	5452	1512	42
CNN N4	84	3486	480	18
CNN N5	101	5050	600	40
CNN N6	118	6903	720	78
CNN N7	135	375820	34995	5030
CNN N8	152	549286	47592	14619

is equivalent to the manually-specified one (as was illustrated in Figure 6.5). The results, compared to a sparse crossbar (with and without conflict-free Merge nodes) are therefore also equivalent to those previously shown in Table 6.4, with the "ALL" case representing what an automatic topology optimization flow would yield, if given mutual exclusivity specifications.

9.7.5 Performance and Quality of Results

The topology optimization results so far have only tested the LU factorization engine, whose interconnect sub-networks have a comparatively small number of merge nodes. However, our other example application – the convolutional neural network – demands more complex interconnect, and yields a larger search space for our method to navigate. Table 9.3 provides the total GENIE executable run times in seconds, as measured with the UNIX `time` command, for several different design parameterizations. The intent is to illustrate the performance overhead due to the topology optimization loop.

The first column indicates the application (LU factorization engine or Convolutional Neural Network) and its parameterized size (N/M for LU, and N for CNN). The second column specifies the number of merge nodes contained within the most complex interconnect domain, whose topological optimization is responsible for the majority of GENIE runtime. For LU, this was the read response network, which can be optimized due to mutual exclusivity constraints. The importance of the read request transmissions remained at 1, so that interconnect domain remained untouched. For the CNN design, the bottleneck interconnect domain was the one which carried the synchronized transmissions between the functional modules in Figure 5.8b. It too has optimization potential due to the mutual exclusivity of transmissions, which in this case, are the non-overlapping image data that are broadcast to each column of DPUs.

The next two columns in the table are: the total number of topological candidates that were iterated over, and the number of those candidates that satisfied contention requirements and went on to be elaborated by the Inner Flow. The latter category is a subset of the former, and topologies that are elaborate contribute significantly more to runtime than those that got rejected due to failing to meet performance requirements. Finally, the last column is the GENIE runtime. With the first two LU systems, we can see a large difference in run time between $N = 32$, which is the largest number of compute units we have tested, and $N = 16$[1]. The optimized read response topologies of both systems matched that of Figure 9.9, which is the expected successful area-minimizing result.

However, for the CNN, there appears to be a dramatic discontinuity in measured runtime between the $N = 4, 5, 6$ and $N = 7, 8$ designs. This arises from the topology refinement flow exiting early at a

[1] changing M has little effect on these times

local minimum for the $N = 4, 5, 6$ cases. The topology, in these cases, never progresses past the initial crossbar, and every single possible candidate refinement yielded a higher estimated area, causing the known global minimum topology (the manually-specified one used in Chapter 8 to never be reached. A deeper investigation reveals the cause of this to be the interaction of topology refinement with other automatic processes within the GENIE flow, specifically pipelining and synchronization.

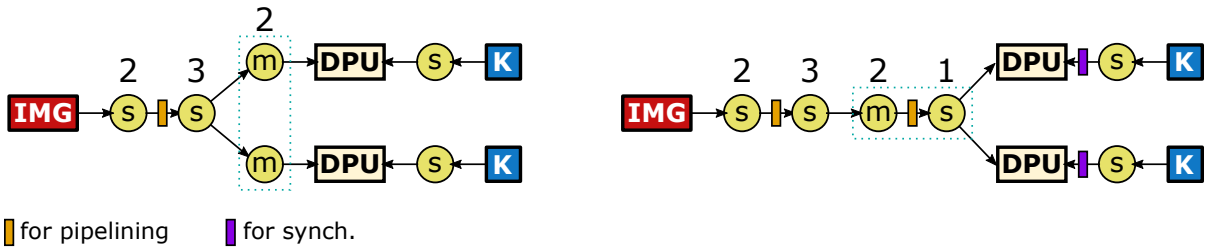

Figure 9.10: **Left:** Section of the initial crossbar topology for an $N = 4$ CNN design. **Right:** Candidate topology derived immediately from crossbar, differing by a single refinement step. Logic depths and pipeline/synchronization registers are indicated.

Figure 9.10 illustrates what occurs during the topological exploration of the CNN network topology. The Figure shows two DPUs that belong to the same column of the array, which in total contains 16 DPUs. Shown also are one of N image buffers, and the two kernel buffers that feed the DPUs. The left side of the Figure is the state of the interconnect for the initial crossbar topology, which is the beginning of the topological search space. This is just a slice of the overall topology relevant to the two DPUs, as the Split and Merge nodes would also contain additional sinks/sources. Two levels of Split nodes are used as to avoid a single $N \times 16$ fanout Split node. During the elaboration of this topology, the interconnect is automatically pipelined using the default maximum logic depth of 5, resulting in a single inserted pipeline register in the location shown. This is inserted in response to the logic depths of 2, 3, and 2 that are shown.

In the initial crossbar topology, every DPU is fed by a final no-conflict Merge node that collects image data from one of the N image buffers. Since image data is broadcast within a column, any two Merge nodes (including the two that are shown) are eligible for a refinement step that would lead to no loss of application performance. The right side of the Figure indicates the resulting candidate topology corresponding to the combination of the two Merge nodes shown. The refinement step combines the two Merge nodes, but introduces a new Split node, which causes a chain of events that increases total area. First, this Split node increases the total combinational logic depth, causing the automatic pipelining process to insert an additional register, shown in orange. The addition of this extra register, in the image data path, would cause a one cycle mismatch in arrival times between the kernel and image data at the DPUs. Due to synchronization constraints being present in this design to avoid such a scenario, two additional registers are inserted in response (shown in purple) to maintain kernel data synchronization. In total, each candidate topology has one fewer Merge node, but three extra 256 bit wide elastic buffers, compared to the crossbar, and thus the optimization loop exits early giving the results shown.

For the larger $N = 7, 8$ CNN designs, exploration went much deeper into the search space, yielding the longer runtimes and increased number of visited and elaborated topologies. In these cases, the Merge nodes were of large enough size such that combining two of them generated sufficient area savings to avoid a local minimum. The $N = 8$ CNN design reached the known area-minimal topology corresponding

to the manually-specified topology used in Chapter 8. The $N = 7$ design stopped short of the global minimum.

9.8 Conclusion

GENIE's automatic topology optimization flow extends GENIE's ability to generate optimized interconnect in response to high-level performance requirements and communication characteristics provided by the user. It is currently limited to creating crossbar-like topologies, and can be selectively applied to some parts of a user's design on a per-interconnect-domain basis. In many of the cases we have studied, the automatic flow is able to replicate the structure of known-best manually-specified topologies in terms of area usage. However, the greedy approach is susceptible to being trapped in local minima of the search space, and future work should look at non-greedy optimization strategies. In the current approach, the time required to explore the search space grows with the cube of the number of Merge nodes in an interconnect domain, which yield significant runtime cost for the larger designs that we tested.

As future work, there exists a possibility to reduce this cost to $O(M^2)$ from $O(M^3)$ by re-using the estimated area savings of previously-computed refinement steps. This will require care to avoid re-use of stale results that become invalidated by the complex interplay of processes within the GENIE Inner Flow that insert registers and other auxiliary interconnect primitives. Two Merge nodes considered for combination early during the exploration of the search space might not be based on the same initial topological starting conditions if reconsidered later in the search space. This same non-ideal interaction of interconnect generation processes is a significant challenge to creating a viable topological exploration method for an interconnect synthesis tool.

Chapter 10

Conclusion

In this work, we have presented and evaluated a new interconnect synthesis and system generation flow, embodied in a tool called GENIE. Its core methodology is to create interconnect by combining together many small and simple parameterizable primitives – an idea present in pre-existing work that we build upon. This approach has unlocked many opportunities for new automation and optimization capabilities, several of which we have studied and implemented in GENIE, forming several core contributions:

- The generation of interconnect for *fine-grained* systems, with lower area, performance, and interface semantic overheads than pre-existing tools and approaches.
- Automatic cycle-level synchronization of applications containing fixed-latency modules, including a flexible input specification method ("synchronization constraints") for the user.
- An automatic logic-depth-based interconnect pipelining algorithm which also cooperates seamlessly with synchronization constraints.
- A topology optimization flow that generates crossbar-like topologies while attempting to minimize total interconnect area.

Notably, we have integrated these isolated features together into a complete system building tool that navigates the engineering challenges of having all its numerous capabilities working together simultaneously, harmoniously, and sometimes in imperfect ways. We evaluated GENIE and its capabilities using two complete design examples: an LU matrix factorization engine and a convolutional neural network. These are fully functional designs which use multiple clock domains and off-chip memory, and were either designed from the ground-up with GENIE or re-written from earlier incarnations based on previous work. Using these two complete examples allowed us to perform a deeper analysis of real-world use cases which would have been time-prohibitive had we used a more extensive benchmark set.

We performed a head-to-head comparison of GENIE's output against that of two other existing tools/methods, using the coarse-grained LU factorization design, which falls into the traditional use case of such tools. In performance-centric use cases, GENIE attained an 11-22% clock frequency advantage over Qsys for the largest tested parameterizations of the LU system, utilizing 7-38% fewer ALM resources. When prioritizing area over performance, GENIE still maintained a 12-25% clock frequency advantage over Qsys, while using up to 43% fewer ALMs. Against the CONNECT FPGA network-on-chip version

of LU, the GENIE-generated systems outperformed it by a wide margin, yielding 31-89% higher clock frequencies and 48-90% fewer ALMs.

We believe that we have demonstrated design effort reduction through an improved interconnect synthesis and system building flow. Chapter 7 demonstrated improved quality-of-results over existing tools for the same workloads, which can translate into achieving better results with the same effort, or perhaps equivalent results with less effort. The combined register insertion flows of Chapter 8 give the user the capability to rapidly explore the design space of their application. For example, automatic synchronization ensures a correct output in the face of changing numbers, configurations, and arrangements of functional modules, as well as across different manual customizations of interconnect pipelining. The automatic pipelining functionality gives the user a knob to trade off area for clock frequency, adding another dimension of exploring the design space and more rapidly converging on a final design.

For our novel use case of fine-grained interconnect synthesis in Chapter 6, we attempted to measure design effort improvement quantitatively (through counting source code lines) as well as qualitatively (by recounting the challenging experiences and processes required to use an existing system building tool for this fine-grained use case). The line count numbers alone were not sufficient to quantify the actual effort required to write each extra line, especially when attempting to match GENIE's performance using Qsys. We speculate that the only way to truly quantify total design effort would be through a controlled study that measures the time required by engineers to complete the design task.

The creation of GENIE was motivated by the shortcomings of existing tools when creating certain kinds of systems. The initial motivating use case was the LU Compute Element, which was originally designed by hand for an unrelated project. Although a small system, the interconnect design was sufficiently difficult and error-prone for us to desire some form of automation for future projects. However, the CE's communication patterns did not fit well with the memory-mapped idioms of existing tools like Qsys, and the small system was sensitive to any introduced area and performance overheads. And so, this initiated the push to ideologically deconstruct interconnect and interconnect synthesis into a small set of conceptual functions and hardware primitives.

Along the way, it was noted that the automatically-generated hardware was, in some places, losing in area or performance compared to hand-designed interconnect. In each instance, we asked "what knowledge would an expert human designer need to make the decision to build the more efficient structure?". The first such examples were merge nodes which were being slowed down by correct-in-the-general-case arbitration circuitry. By knowing that transmissions were mutually exclusive, the arbitration circuitry would not be necessary. Each such analysis yielded a new piece of high-level specification that could optionally be specified to GENIE to help it perform the same kinds of optimizations that would be done by an expert designer. The end result is a tool that not only succeeds at its original motivating goal of reducing design effort in fine-grained systems, but also offers automation and optimization capabilities to more general use cases as well.

10.1 Future Work

There are numerous areas of potential future work on both the GENIE tool/flow itself, as well as analyses and experiments that could be performed with its existing incarnation. Perhaps the most important such work would be to continue to evaluate GENIE using more design examples beyond the two presented in this thesis, which is a matter of expending more engineer-hours of effort.

In such additional design examples, it would be beneficial to better-exercise the existing features of GENIE, such as building a wider variety of network topologies (using manual specification) to build analogues of existing NoC topologies like tori, rings, meshes, and fat trees using combinations of split/merge primitives. Extending our automatic topology exploration to include these other classes of topologies (beyond the crossbar sub-space that we currently explore) would also improve the capabilities of the tool. Our current automatic topology exploration algorithm is prone to becoming stuck in local minima, and its runtime does not scale well with input size. Alternate techniques such as simulated annealing would be worthwhile to explore.

The current crossbar-like topologies generated automatically by GENIE are free from deadlocks due to topology alone. If GENIE were to support more types of automatically-generated topologies, it would be necessary to have more robust handling of potential deadlocks. The future work in this area could be phased in through two tiers of increasing complexity: deadlock detection, and deadlock avoidance. Detection on topology alone would be the simplest, but overly conservative: a message informing the user that their (manually or automatically generated) topology could potentially (but not for certain) cause a deadlock. Information about the actual transmissions would make detection more accurate, and could potentially detect application-level deadlocks (ones that are a function of the actual traffic). GENIE already has transmission information in the form of RS links, but aside from mutual exclusivity, there is no sense of scheduling or causal/temporal relationships between them that would be necessary for a detailed automated deadlock analysis. More complex techniques in which GENIE actually avoids or prevent deadlocks could take place through routing changes, topological modifications, or microarchitectural adjustments such as configuring virtual channels. All techniques would benefit from more detailed knowledge about the user's transmissions than is currently provided.

GENIE currently supports an arbitrary number of clock domains, but our design examples only use two, and it may be instructive to stress-test the automatic clock domain crossing insertion by finding (realistic) use cases that utilize three or more clock domains, since that is the regime in which the underlying optimization problem enters NP-complete complexity.

We have shown the limitations of a purely logic-depth-based approach to automatic pipelining. Certainly, an avenue for future work would be finding methods of including the approximate effect of signal propagation delay within pipelining criteria, as this is a significant portion of combinational delay on modern FPGAs and large designs. As the majority of this thesis was completed before Intel introduced their own version of automatic pipelining in Stratix 10 FPGAs, it would also be very instructive to perform a new comparison using these newer FPGAs.

One type of optimization left unexplored in this work is that of automatically designing the binary encodings for internal address representations of transmissions in an intelligent manner. Address conversion, briefly touched upon in Section 3.4.3 is currently fairly primitive in GENIE, as the internal representation of addresses (that enable Split nodes to route transmission to their destinations) are monotonically-increasing unique integers. However, it may conceivably be possible to devise a scheme that generates encodings that, based on network topology, simultaneity of transmissions, and the encodings of addresses produced and consumed by user modules, would use fewer bits and/or fewer address conversion primitives and thus create lower-cost interconnect.

On the micro-architecture front, the six interconnect primitives that GENIE currently uses can be made more parameterizable. For example, the fully-functional Merge nodes currently has a round-robin arbiter that gives equal weight to all incoming transmissions. A method to (first manually, then automat-

ically) give *different* weights or time-shares to different transmissions could benefit certain applications. GENIE currently lacks support for virtual channels, which may lead to deadlocked transmissions under certain topologies. Some of this can be mitigated (and has been, in our design examples) through using separate physical networks, but nevertheless the introduction of virtual channel capability would further generalize the types of interconnect that GENIE is able to build.

10.1.1 GENIE and Other Design Tools

GENIE currently exists as a stand-alone application. A future goal would be to instead link the C++ library portion with another CAD tool, such as an HLS tool or graphical front-end, to serve as an internal interconnect synthesis engine. One specific use case for existing HLS tools would be the ability to integrate non-HLS RTL modules within the overall system. For the simpler use case of fixed-latency modules, GENIE is presently capable of accepting user-defined latency annotations for every possible input-to-output pair. If the modules generated by the HLS tool were similarly (and automatically) annotated with latency information generated from a partial schedule, GENIE could stitch together the HLS portion of the system with the user-provided RTL modules by using synchronization constraints.

It is reasonable to expect complex systems built with HLS to also interact with external memory or I/O that introduces variable latency and the need for backpressure into the system. The discussion on performance-oriented variable latency use cases for synchronization constraints in Section 8.1.5 applies here.

10.1.2 GENIE and Hard NoCs

A case has been made for including hard Networks-on-Chip within the FPGA fabric [5], and they have recently started to appear commercially in Xilinx's Versal architecture [67] as part of its Adaptable Compute Acceleration Platform (ACAP) devices, which also include a traditional FPGA fabric. In addition to providing high-bandwidth communication for user logic, it also provides dedicated connections to already-hardened on-chip external memory and I/O interfaces such as DDR4 and PCI Express. A future version of GENIE may wish to integrate hard NoCs as part of an overall interconnect synthesis strategy, and we will briefly speculate on some possibilities.

One important consideration is that a NoC is not suitable for all types of communication. Since it is a shared resource, bandwidth and latency are not necessarily guaranteed or predictable. Therefore, latency-insensitive protocols are used at fabric egress ports (AXI4 memory mapped and streaming in Versal's case), which rules out mapping tightly-coupled fine-granularity communications, such as those explored in Chapter 6, to the NoC. However, communications to and from off-chip memory would be ideal for mapping. The fixed protocol also restricts the available data widths at the NoC interface.

Although the NoC protocol is fixed, as long as a streaming interface exists, it should be able to connect with GENIE-generated interconnect, by using the subset of GENIE's Routed Streaming protocol used for latency-insensitive traffic – the ability to send data, have flow control, and select a destination are sufficient. Some soft-logic address conversion would need to take place, further generalizing the existing address representation problem.

One role that a future version of GENIE could play is automatically partitioning a system's communications between a hard NoC and traditional soft logic. The most latency-insensitive communications (those without tight latency bounds) would be ideal for the NoC, as long as bandwidth is available. To

allocate bandwidth, Xilinx includes a CAD flow called the "network compiler" that performs a similar function to GENIE's topology optimization – given a set of high-level communication requirements (bandwidth and traffic class), it determines the optimal mapping to the NoC resources. Rather than handling bandwidth allocation manually, GENIE could utilize such tools for the portions allocated to the NoC (although a feedback/query mechanism would be ideal, to know the point at which further mapping should be relegated to soft logic).

Before automatic partitioning, a good first step would be to allow the user to explicitly map certain logical links to the NoC. Later, the type of communication links (whether they use backpressure or not), combined with communication requirements (importance, packet size, and perhaps some constraint on latency) could be used to automatically qualify links for NoC-mapping.

10.1.3 GENIE and Pipelined FPGA Interconnect

Another relatively new feature of FPGA fabrics is embedded interconnect registers, such as those found in Intel's Stratix 10 HyperFlex architecture [50]. These registers do not consume regular FF resources found in logic blocks, and can be enabled selectively during placement and routing without otherwise disturbing the netlist. This provides an ideal capability to automatically pipeline long links to improve clock frequency while having complete physical knowledge, which is something that a tool like GENIE lacks. However, this capability is hamstrung without an appropriate way to tell the FPGA toolchain which nets are allowed to be pipelined.

Intel adds additional Verilog attributes that offer a basic form of GENIE's synchronization constraints – they state that all the wires of an HDL bus are allowed to be automatically pipelined (up to a maximum amount), and that they all must have the same final latency. This allows automatic pipelining of feed-forward paths without backpressure. In the future, GENIE could leverage these latency-insensitive net attributes and replace its logic depth based automatic pipelining flow when targeting compatible FPGAs, under certain conditions (for example, lack of simultaneously-present synchronization constraints).

Currently, GENIE's elastic buffer primitives rely on register enable control signals, making them not directly mappable to Stratix 10 embedded interconnect registers. When there exist many consecutive pipeline stages (the exact number of which might not even be known at GENIE system generation time), one could implement an alternate style of latency-insensitive channels in which a FIFO exists at the destination and deasserts readiness when a certain threshold is reached. Both the forward path to the FIFO, and ready signal backwards from the FIFO, could be pipelined without enable signals, and therefore be mappable to embedded interconnect registers. This style has been shown [3] to have area and frequency advantages on Stratix 10 when there exist multiple consecutive pipeline stages.

10.1.4 GENIE and ASIC Design

Although GENIE targets FPGAs, it may also be possible to generalize the tool to target ASIC design as well. One of the challenges of FPGA design is the lack of control and visibility into the physical implementation – placement and routing are automatic, and can not guarantee predictable area usage or delays resulting from a given change to the logical design. For example, our choice of using combinational logic depth instead of absolute delay for automatic pipelining was influenced by this.

If given a blank slate of silicon and a standard cell library, rather than an existing architecture to map to, GENIE's interconnect primitives could be stored in fully-implemented form and would have

predictable area (now measured in physical units) and timing characteristics. It is not clear whether GENIE would need to supplant some of the functions of a traditional ASIC design flow (floorplanning, placement, routing) to fully take advantage of its knowledge of latency-insensitive links. In this context, automatic pipelining would ideally be placement-aware and would run simultaneous to this flow. Even if it does not, and simply hands off a netlist to an existing commercial flow, the accuracy of automatic pipelining and topology area estimation could possibly improve.

As an aside, it is worthwhile to note that locked-down and predictable physical placement and routing are also possible in the FPGA space without the need for ASIC-level of control over the substrate. The Hoplite NoC [40] has managed to create a physical interconnect implementation by manually placing logic and routing resources on the FPGA device with user constraints, yielding predictable area and performance characteristics. Although highly architecture-specific, such techniques may be worth investigating in an FPGA-targeted tool like GENIE.

10.2 Software Release

GENIE is an open source project, and its home page is located at `http://www.eecg.toronto.edu/~jayar/software/GENIE/`. Additionally, the source code for GENIE, our example applications, and raw data for experiments will be included as supplemental attachments to the digital submission of this thesis.

Appendix A

Lua Specification Examples

This appendix provides additional Lua code snippets that demonstrate the use of the multitude of GENIE features described in the main document. It assumes familiarity with the basic example first provided in Section 3.2.6. First, a small primer on Lua table syntax is given that may help in understanding some of the code.

The Lua language only has one complex data type, the *table*, which is an associative map of key/value pairs in which all keys must be unique. The types of each key and value, even within the same table, can differ. Here is an example table, declared as a literal and assigned to a variable, containing five key/value pairs with a mix of type combinations, with the last value being a table itself:

```
1  mytab = {'stringkey1': 42, 'Alice': 'Bob', 53:  'Charlie',
2          12: 34, 'innertable': {'innerkey': 'innervalue'} }
3
4  -- modify table entries using two different syntax variations
5  mytab.stringkey1 = 43       -- dot, works only for string keys
6  mytab['Alice'] = 'Robert'   -- [] also works for all types of keys
7  mytab.alice = 'Roberto'
8  mytab[53] = 100             -- [] must be used for integer keys
9
10 -- same syntax creates new entries
11 mytab.newkey = 6
12 mytab['anotherkey'] = true
13
14 -- reads entries
15 x = mytab[53]         -- 100
16 y = mytab['Alice']    -- Roberto
```

Arrays are simply tables with integer keys (starting at 1, not 0). The following code creates an array where the values Alpha, Beta, 42, and Delta have implied keys 1, 2, 3, and 4 respectively:

```
17 arr = {'Alpha', 'Beta', 42, 'Delta'}
18
19 x = arr[3] -- x contains 42
```

Because entries in an array have unique implicit integer keys, duplicate values are allowed. For some applications, it may be more convenient to work with *sets*, in which each member can only appear once. These are simply tables where the desired members of the set appear as the keys instead of the values. The associated value of each key is some dummy value, such as the boolean constant `true`.

Many GENIE API functions accept a collection of items as one or more of their arguments. A collection argument can be provided in one of three ways:

- A single scalar value (a collection of one thing)
- A table representing an array
- A table representing a set

A.1 Mutual Exclusivity

Routed Streaming links may be marked "mutually-exclusive", which is a guarantee by the user that they will never be driven simultaneously, nor temporally overlap in any way. GENIE can leverage this user-provided hint in two ways:

- Instantiate a conflict-free Merge node within the interconnect if all its incoming links are mutually exclusive.
- More aggressively combine Merge nodes during the automatic topology optimization loop.

GENIE provides two methods, exposed via the Builder interface, to mark exclusivity. `make_exclusive` accepts a single collection of links, marking them all mutually exclusive with respect to one another. `make_exclusive_multi` accepts a variable number of collections of links, and marks each member of a collection exclusive with all members of all other collections, but not with the collection it is part of.

```
1  -- Assume b is a Builder object
2  -- Create three RS links and store a reference to each in a separate variable
3  link1 = b:rs_link(src1, sink1)
4  link2 = b:rs_link(src2, sink2, srcAddr2)
5  link3 = b:rs_link(src3, sink3, nil, sinkAddr3) -- no src address, only sink
6
7  -- Mark all three as exclusive
8  b:make_exclusive({link1, link2, link3})
9
10 -- Mark link1,link2 exclusive with link3 but not with each other
11 b:make_exclusive_multi({link1, link2}, link3)
```

Note that all RS links that begin at the same physical RS interface (and are bound to different source addresses) are automatically mutually exclusive by definition.

A.2 Manual Topology Specification

GENIE allows the user to explicitly specify a topology through manual creation and connection of Split and Merge nodes. These explicitly-instantiated Split and Merge nodes are connected to each other, and to the RS source/sink RS interfaces of functional modules, via special *topological links*. When connecting a topological link to a Split/Merge node, the entire node is used as the source/sink of the connection. Note that logical RS links must still be specified as usual, and will be automatically routed over the explicitly-specified topological links using the shortest number of hops. Manual routing specification is left as future work.

The following code instantiates four modules. Two of them have RS source interfaces, and the other two have RS sink interfaces. The four logical RS links between them form a full crossbar (both sources to both sinks). Without manual topology specification, GENIE would automatically create two Split and two Merge nodes. However, here we explicitly create a single Merge and single Split node and force all traffic over a single shared connection:

```
-- Assumes:
--   b is a Builder object.
--   SrcMod is a module with RS source interface 'send'
--   SinkMod is a module with RS sink interface 'recv'

b:system('SysDef')
  -- Create two sender modules and two receiver modules
  b:instance('SrcMod', 'src1')
  b:instance('SrcMod', 'src2')
  b:instance('SinkMod', 'sink1')
  b:instance('SinkMod', 'sink2')

  -- Create logical RS links from both sources to both sinks
  b:rs_link('src1.send', 'sink1.recv')
  b:rs_link('src1.send', 'sink2.recv')
  b:rs_link('src2.send', 'sink1.recv')
  b:rs_link('src2.send', 'sink2.recv')

  -- Create a single split node and single merge node
  b:split('the_split')
  b:merge('the_merge')

  -- Create topological links from both sources to the_merge
  b:topo_link('src1.send', 'the_merge')
  b:topo_link('src2.send', 'the_merge')

  -- Create topological link from the_merge to the_split
  b:topo_link('the_merge', 'the_split')

  -- Connect the_split to the two sinks
  b:topo_link('the_split', 'sink1.recv')
  b:topo_link('the_split', 'sink2.recv')
```

A.3 Synchronization Constraints

The following complete Lua listing creates the example system from Figure 8.4. It assumes all data widths to be 14 bits.

```
require 'builder'
local b = genie.Builder.new()

b:component('A')
  b:clock_sink('clk')
  b:reset_sink('reset')
  b:rs_src('out', 'clk')
    b:signal('valid', 'o_valid')
    b:signal('data', 'o_data', 14)
```

```
10
11 b:component('B')
12   b:clock_sink('clk')
13   b:reset_sink('reset')
14   b:rs_sink('in', 'clk')
15     b:signal('valid', 'i_valid')
16     b:signal('data', 'i_data', 14)
17   b:rs_src('out', 'clk')
18     b:signal('valid', 'o_valid')
19     b:signal('data', 'o_data', 14)
20   b:internal_link('in', 'out', 2) -- latency 2
21
22 b:component('C')
23   b:clock_sink('clk')
24   b:reset_sink('reset')
25   b:rs_sink('in', 'clk')
26     b:signal('valid', 'i_valid')
27     b:signal('data', 'i_data', 14)
28   b:rs_src('out', 'clk')
29     b:signal('valid', 'o_valid')
30     b:signal('data', 'o_data', 14)
31   b:internal_link('in', 'out', 4) -- latency 4
32
33 b:component('D')
34   b:clock_sink('clk')
35   b:reset_sink('reset')
36   b:rs_sink('in1', 'clk')
37     b:signal('valid', 'i_valid1')
38     b:signal('data', 'i_data1', 14)
39   b:rs_sink('in2', 'clk')
40   b:signal('valid', 'i_valid2')
41   b:signal('data', 'i_data2', 14)
42
43 b:system('sync_cnst_test')
44   b:clock_sink('clk')
45   b:reset_sink('reset')
46
47   -- Instantiate A,B,C,D and connect clocks and resets
48   for name in Set.mkvalues{'A', 'B', 'C', 'D'} do
49     b:instance(name, name)
50     b:clock_link('clk', name .. '.' .. 'clk')
51     b:reset_link('reset', name .. '.' .. 'reset')
52   end
53
54   local link_ab = b:rs_link('A.out', 'B.in')
55   local link_ac = b:rs_link('A.out', 'C.in')
56   local link_bd = b:rs_link('B.out', 'D.in1')
57   local link_cd = b:rs_link('C.out', 'D.in2')
58
59   -- define sync constraint
60   b:sync_constraint({link_ab, link_bd}, '-', {link_ac, link_cd}, '=', 0)
```

Internal links are specified with `internal_link`, which accepts the name of two interfaces in the same module and the latency between them. Synchronization constraints are defined with `sync_constraint`,

which expects one or more chain terms (each an ordered array of RS links) separated by '+' or '-' tokens, followed by a relational operator and an integer constant.

A.4 Transmission Importance and Packet Size

The importance and packet size of a transmission are used during automatic topology optimization. By default, all RS links have an importance of 1 and packet size of 1. Importance can be changed via the `set_importance` method of an RS link object, or be set on a source RS port with its `importance` method to change the default importance of all outgoing transmissions. Similarly, the packet size of an individual RS link is set with `set_packet_size`, and a port-wide default is set with `packet_size`.

```
-- All RS links connected to instances of this interface will
-- by default have an importance of 0.6 and packet size of 128
b:rs_src('ifacename', 'ifaceclk')
  b:signal(...)
  ...
  b:importance(0.6)
  b:packet_size(128)

-- This specific RS link will have importance 0.5 and packet size of 192
x = b:link(src1, sink1)
x:importance(0.5)
x:packet_size(192)
```

A.5 Automatic Pipelining Control

GENIE's automatic pipelining inserts registers to enforce a maximum combinational logic depth. As well as being a global command-line setting, it is possible to override the maximum logic depth on a per-system basis:

```
b:system('sysname')
  b:max_logic_depth(4)
```

It is also possible to inform GENIE of the combinational logic depth "looking in" to a source or sink interface of a user-defined functional module. GENIE will take these user logic depths into account when pipelining. Without specifying this, GENIE assumes that the interface is fully registered and has a logic depth of 0.

```
-- This source RS interface has a worst-case logic depth of 2 LUTs before hitting
-- an internal register within the containing module.
b:rs_src('ifacename', 'ifaceclk')
  b:signal(...)
  ...
  b:logic_depth(2)
```

Bibliography

[1] lpsolve – Mixed Integer Linear Programming (MILP) solver. `https://sourceforge.net/projects/lpsolve/`.

[2] CLOC – Count Lines of Code. `http://cloc.sourceforge.net`, 2015.

[3] Mustafa Abbas and Vaughn Betz. Latency Insensitive Design Styles for FPGAs. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, pages 360–367, 2018.

[4] Ossama Abdel-Hamid, Abdel-Rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 22(10):1533–1545, October 2014.

[5] Mohamed S. Abdelfattah, Andrew Bitar, and Vaughn Betz. Take the Highway: Design for Embedded NoCs on FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 98–107, New York, NY, USA, 2015. ACM.

[6] Altera Corporation. FPGA Architecture White Paper. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf`.

[7] Altera Corporation. QSys - Altera's System Integration Tool. `http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html`.

[8] ARM Ltd. AMBA AXI4-Stream Protocol Specification. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html`.

[9] ARM Ltd. AMBA Open Specifications. `http://www.arm.com/products/system-ip/amba/amba-open-specifications.php`.

[10] D. Berend and T. Tassa. Improved Bounds on Bell Numbers and on Moments of Sums of Random Variables. *Probability and Mathematical Statistics*, 30(2):185—205, 2010.

[11] Endre Boros, Peter L. Hammer, and Ron Shamir. A Polynomial Algorithm for Balancing Acyclic Data Flow Graphs. *IEEE Trans. Comput.*, 41(11):1380–1385, November 1992.

[12] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC Scheduling with Recurrence. Minimization in High-Level Synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[13] Luca P Carloni, Kenneth L McMillan, Alexander Saldanha, and Alberto L Sangiovanni-Vincentelli. A Methodology for Correct-by-Construction Latency Insensitive Design. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 309–315. IEEE Press, 1999.

[14] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. Elastic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1437–1455, 2009.

[15] Po Rong Chang and C. S. G. Lee. A Decomposition Approach for Balancing Large-Scale Acyclic Data Flow Graphs. *IEEE Trans. Comput.*, 39(1):34–46, January 1990.

[16] Seonil Choi and Viktor K Prasanna. Time and Energy Efficient Matrix Factorization Using FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 507–519. Springer, 2003.

[17] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. In *FPGA*, pages 97–106, 2011.

[18] Eric S Chung and Michael K Papamichael. ShrinkWrap: Compiler-Enabled Optimization and Customization of Soft Memory Interconnects. In *FCCM*, pages 113–116. IEEE, 2013.

[19] J. Cong, Y. Huang, and B. Yuan. ATree-Based Topology Synthesis for On-Chip Network. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 651–658, 2011.

[20] Jason Cong and Bingjun Xiao. Minimizing Computation in Convolutional Neural Networks. In *ICANN*, 2014.

[21] Jason Cong and Zhiru Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 433–438, New York, NY, USA, 2006. ACM.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[23] J. Cortadella, M. Galceran-Oms, M. Kishinevsky, and S. S. Sapatnekar. RTL Synthesis: From Logic Synthesis to Automatic Pipelining. *Proceedings of the IEEE*, 103(11):2061–2075, Nov 2015.

[24] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiway Cuts (Extended Abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 241–251, New York, NY, USA, 1992. ACM.

[25] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis. The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

[26] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[27] William J Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed systems*, 3(2):194–205, 1992.

[28] William J Dally and Charles L Seitz. Deadlock-free Message Routing in Multiprocessor Interconnection Networks. 1988.

[29] Youxin Gao and Martin D. F. Wong. A Graph Based Algorithm for Optimal Buffer Insertion Under Accurate Delay Models. In *DATE*, 2001.

[30] Wai Hong Ho and T. M. Pinkston. A Methodology for Designing Efficient On-Chip Interconnects on Well-Behaved Communication Patterns. In *The Ninth International Symposium on High-Performance Computer Architecture*, pages 377–388, 2003.

[31] Xiaobo Hu, S. C. Bass, and R. G. Harber. Minimizing the Number of Delay Buffers in the Synchronization of Pipelined Systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 13(12):1441–1449, November 2006.

[32] Yutian Huan and A DeHon. FPGA Optimized Packet-Switched NoC using Split and Merge Primitives. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 47–52, Dec 2012.

[33] Intel. Arria 10 Core Fabric and General Purpose I/O Handbook. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_handbook.pdf`.

[34] Intel Corporation. Avalon Interface Specifications. `https://www.altera.com/documentation/nik1412467993397.html`.

[35] JEDEC. FBDIMM: ADVANCED MEMORY BUFFER (AMB). https://www.jedec.org/standards-documents/docs/jesd-82-20a.

[36] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. On-Chip Networks, Second Edition. *Synthesis Lectures on Computer Architecture*, 12(3):1–210, 2017.

[37] Timothy Kam, Michael Kishinevsky, Jordi Cortadella, and Marc Galceran-Oms. Correct-by-construction Microarchitectural Pipelining. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 434–441. IEEE, 2008.

[38] 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)*, pages 739–751, June 2018.

[39] Nachiket Kapre. Deflection-Routed Butterfly Fat Trees on FPGAs. In *FPL*, pages 1–8. IEEE, 2017.

[40] Nachiket Kapre and Jan Gray. Hoplite: A Deflection-Routed Directional Torus NoC for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 10(2):14:1–14:24, March 2017.

[41] Nachiket Kapre, Nikil Mehta, Raphael Rubin, Henry Barnor, Michael J Wilson, Michael Wrighton, Andre DeHon, et al. Packet Switched vs. Time Multiplexed FPGA Overlay Networks. In *FCCM*, pages 205–216. IEEE, 2006.

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[43] Daniel Kroening and Wolfgang J Paul. Automated Pipeline Design. In *Design Automation Conference, 2001. Proceedings*, pages 810–815. IEEE, 2001.

[44] P. Kulkarni, P. Gupta, and R. Beraha. Minimizing Clock Domain Crossing in Network on Chip Interconnect. In *Fifteenth International Symposium on Quality Electronic Design*, pages 292–299, March 2014.

[45] P. Kulkarni, P. Gupta, and R. Beraha. Minimizing Clock Domain Crossing in Network on Chip Interconnect. In *Fifteenth International Symposium on Quality Electronic Design*, pages 292–299, March 2014.

[46] Martin Langhammer. Floating Point Datapath Synthesis for FPGAs. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 355–360. IEEE, 2008.

[47] Lattice Semiconductor. LatticeMico System Development Tools. `http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/EmbeddedDesignSoftware/LatticeMicoSystem.aspx`.

[48] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 5 2015.

[49] Charles E Leiserson and James B Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6(1-6):5–35, 1991.

[50] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The Stratix™ 10 Highly Pipelined FPGA Architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 159–168. ACM, 2016.

[51] Michael Papamichael. CONNECT Network Generator. http://users.ece.cmu.edu/ mpapamic/connect/.

[52] Micron Technology, Inc. Hybrid Memory Cube. https://www.micron.com/products/hybrid-memory-cube.

[53] Thomas Moscibroda and Onur Mutlu. A Case for Bufferless Routing in On-chip Networks. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 196–207, New York, NY, USA, 2009. ACM.

[54] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo. Designing Application-Specific Networks on Chips with Floorplan Information. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 355–362, Nov 2006.

[55] S. Murali and G. De Micheli. An Application-Specific Design Methodology for STbus Crossbar Generation. In *Design, Automation and Test in Europe*, pages 1176–1181, 2005.

[56] Eriko Nurvitadhi, James C Hoe, Timothy Kam, and Shih-Lien L Lu. Automatic Pipelining from Transactional Datapath Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–454, 2011.

[57] Marc Galceran Oms, Alexander Gotmanov, Jordi Cortadella, and Michael Kishinevsky. Microarchitectural Transformations Using Elasticity. *JETC*, 7:18:1–18:24, 2011.

[58] Michael K. Papamichael and James C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing Nocs in the Context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, New York, NY, USA, 2012. ACM.

[59] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Efficient Synthesis of Networks On Chip. In *Proceedings 21st International Conference on Computer Design*, pages 146–150, 2003.

[60] PUC-Rio. The Programming Language Lua. http://www.lua.org/.

[61] Alex Rodionov, David Biancolin, and Jonathan Rose. Fine-Grained Interconnect Synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 46–55, New York, NY, USA, 2015. ACM.

[62] Alex Rodionov, David Biancolin, and Jonathan Rose. Fine-Grained Interconnect Synthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 9(4):31:1–31:22, August 2016.

[63] Alex Rodionov and Jonathan Rose. Automatic FPGA System and Interconnect Construction with Multicast and Customizable Topology. In *Proceedings of the 2015 International Conference on Field-Programmable Technology*, 2015.

[64] Alex Rodionov and Jonathan Rose. Synchronization Constraints for Interconnect Synthesis. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 95–104, 2017.

[65] Alex Rodionov and Jonathan Rose. Automatic Topology Optimization for FPGA Interconnect Synthesis. In *Proceedings of the 2018 International Conference on Field Programmable Logic and Applications*, 2018.

[66] K. Srinivasan, K. S. Chatha, and G. Konjevod. An Automated Technique for Topology and Route Generation of Application Specific On-chip Interconnection Networks. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, pages 231–237, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. Network-on-Chip Programmable Platform in VersalTM ACAP Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 212–221, New York, NY, USA, 2019. ACM.

[68] Henry Wong, Vaughn Betz, and Jonathan Rose. Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2067–2080, 2014.

[69] Xilinx Corporation. Accelerating Integration. `http://www.xilinx.com/products/design-tools/vivado/integration/`.

[70] Xilinx Corporation. Xilinx White Paper WP284: Advantages of the Virtex-5 FPGA 6-Input LUT architecture. 2007.

[71] Jiang Xu, Wayne Wolf, Joerg Henkel, and Srimat Chakradhar. A Design Methodology for Application-specific Networks-on-chip. *ACM Trans. Embed. Comput. Syst.*, 5(2):263–280, May 2006.

[72] Mark Zastrow. Machine outsmarts man in battle of the decade. *New Scientist*, 229(3065):21 –, 2016.

[73] Wei Zhang, Vaughn Betz, and Jonathan Rose. Portable and Scalable FPGA-based Acceleration of a Direct Linear System Solver. *ACM Trans. Reconfigurable Technol. Syst.*, 5(1):6:1–6:26, March 2012.