Dataflow Optimized Overlays for FPGAs

Siddhartha

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2019

Thesis Abstract

Dataflow Coprocessor Overlay (DaCO) is an FPGA-tuned dataflow-driven overlay architecture that offers fine-grained parallelism capable of delivering speedups of up to 2.8× on sparse, irregular computations over competing architectures (*e.g.* modern microprocessors and existing dataflow overlays). DaCO delivers these improvements with a custom instruction datapath that exploits the raw parallelism exposed by the dataflow triggering rule – instructions execute asynchronously as soon as their operands are available. However, this simple triggering logic can expose large amounts of irregular instruction-level parallelism that can be hard to manage. This thesis addresses this challenge in three steps: (1) design of a lightweight scheduling circuit inside each DaCO soft-processor that enables large-scale out-of-order instruction execution at runtime, (2) design of a priority-aware communication framework that delivers improved quality of service to critical communication packets, and (3) compiler support that optimizes the dataflow graph structure for improved runtime execution. DaCO is optimized for the Arria 10 AX115S (20nm SoC) FPGA board in order to take advantage of the hard on-chip floating-point DSP blocks. Overall, when benchmarked with sparse-matrix vector multiply kernels, DaCO improves throughput performance by up to 2.4× over existing in-order dataflow overlays, and delivers a peak operational throughput of up to 38 MFLOPs/processor, or a peak total throughput of 3.5 GFLOPs/sec.

The DaCO engine is composed of a custom dataflow-inspired soft-processor and a priority-aware Network on Chip (NoC) communication framework. Each soft-processor has a custom datapath that operates directly on the dataflow graph stored in local memory. We design a novel criticality-aware scheduling circuit inside the soft-processor that allows large-scale out-of-order node execution with minimal resource overheads. This is achieved by using a one-time memory reorganization strategy together with a lightweight leading-ones detector circuit.

The datapath is fully-pipelined and employs data-forwarding for achieving high-performance, while the block RAMs (BRAMs) are multipumped to ensure efficient resource utilization. On the target Arria 10 chip, we can fit up to 600 soft-processors, where each DaCO soft-processor consumes 779 ALMs, four BRAMs, and three DSP blocks, and operates at a 3.7ns clock.

The NoC communication framework is built with Hoplite-Q*, a novel FPGA-friendly router that augments the existing Hoplite router to support priority-aware routing features. Together, the DaCO soft-processor and Hoplite-Q* manage and prioritize critical compute paths that were left unaddressed in prior work. On its own, Hoplite-Q* can accelerate high-priority communication packets by up to 90% when compared to the baseline Hoplite router. Each Hoplite-Q* router consumes 215 ALMs (64b packet with 32b payload) and can operate at a 3.3ns clock. DaCO also supports a clustered topology, where soft-processors in the overlay can be grouped and connected by a local crossbar, while out-of-cluster communication is serviced by a Hoplite-Q* network. This strategy improves performance by up to 1.8–2× with only 15–40% resource overhead from the crossbar (cluster size of two to four).

Finally, this thesis also explores the importance of criticality in dataflow workloads and the importance of compilation support. We explore the limits of recursive unrolling and tree balancing on dataflow graphs and quantify the tradeoffs between excess computation and reductions in the critical path with these techniques. We then develop a Huffman-inspired reassociation scheme that optimizes the dataflow graph based on a statically computed node/edge criticality. Together with fanin and fanout decomposition, we quantify the effect of all these software transformations on the dataflow graph and demonstrate the performance tradeoffs when run on hardware. These software transformations are packaged as compiler optimizations that provide an easy-to-use programming model for the DaCO engine.

In the future, our aim is to develop the DaCO ecosystem further to support various flavors of dataflow-driven soft-processors. In particular, an asynchronous dynamic dataflow graph processor would map well to iterative problems from domains such as graph convolutional networks, molecular dynamics, and PageRank. In addition, we hope to improve the DaCO programming model by extending the existing ISA and supporting a codelet-based model, where the compute abstraction assumes a more coarse-grained instruction-graph.

Acknowledgements

First and foremost, I would like to express sincere gratitude towards my PhD advisor, Professor Nachiket Kapre, whose constant patience, guidance, and willingness to invest time in my development made this whole journey possible. His attention to detail and insistence on good research practices improved my technical and research skills year on year, and I could **not** have asked for a *better* advisor for this PhD endeavour.

I would also like to extend my sincerest thanks to the past and current members of my thesis advisory committee: Professor Srikanthan Thambipillai (SCSE), Professor Arindam Basu (EEE), and Professor Sylvain Barbot (EOS). Their input in the committee meetings served as valuable markers that helped guide my research progress to its final goal. I would also like to give a special thanks to Professor Arvind Easwaran for handling the administrative duties in the absence of Professor Nachiket. He was always very kind and forthcoming in our interactions, and only made my time at the university easier.

I would like to extend thanks to the many fellow students that I had the pleasure of interacting and/or collaborating with in the HESL group. A special mention to Abhishek Jain, whose undying enthusiasm for all things FPGAs is very contagious. Lastly, a shoutout to Jeremiah, whose promptness and upbeat attitude only helped ensure that students in the lab, including me, were given more than adequate support on all matters.

Finally, none of this would have been possible without my family and friends. To Mum, Dad, and Shruti, *thank you* for the unconditional love and undying support. To all my friends, thank you for keeping me sane with encouragement and humour in my difficult times. To my family and friends, I will forever cherish each and every one of you.

Contents

List of Acronyms

DaCO **Da**taflow **C**oprocessor **O**verlay

FPGA **F**ield **P**rogrammable **G**ate **A**rray

ALM **A**daptive **L**ogic **M**odule (programmable logic element in Intel FPGAs)

DSP **D**igital **S**ignal **P**rocessor

ASIC **A**pplication-**S**pecific **I**ntegrated **C**ircuit

NoC **N**etwork **O**n **C**hip

PSNoC **P**acket-**S**switching **N**etwork **O**n **C**hip

OoO **O**ut **O**f **O**rder

DFG **D**ata **F**low **G**raph

DAG **D**irected **A**cyclic **G**raph (used interchangeably with DFG in this thesis)

PE **P**rocessing **E**lement

L Matrix **L**ower Triangular Matrix

U Matrix **U**pper Triangular Matrix

KLU Clark-**K**ent **LU** Solver

SPICE **S**imulation **P**rogram with **I**ntegrated **C**ircuit **E**mphasis

HDL **H**ardware **D**escription **L**anguage

HLS **H**igh-**L**evel **S**ynthesis

List of Figures

List of Tables

Chapter 1

Introduction

Ever since the end of Dennard scaling [41] circa 2006, computer architects have had to keep the power utilization wall [40, 35] in mind when designing next generation processors. Embarassingly parallel problems (*e.g.* graphics, dense matrix-multiplications) were easy targets for vector processors (*e.g.* the ARM NEON vector processor, Intel AVX instructions, Graphic Processing Units), while superscalar/VLIW (Very Long Instruction Word) [9] and multicore [100] architectures were able to push the envelope further by exploiting limited forms of instruction-level parallelism. Eventually, as process technology transistor trends continued to scale over the years, it only got exponentially harder to keep large percentages of chip area active due to power constraints – referred often to as the *dark silicon* problem [43, 118, 117, 35]. Studies have shown that simply increasing the number of cores on the chip does not allow us to maintain cadence with the Moore's Law [36, 117, 12, 63], and hence, new strategies and roadmaps to tackle dark silicon have been proposed in recent years. There is a plethora of literature on many different techniques (*e.g.* dynamic voltage and frequency scaling / near-threshold voltage [53, 34], computational sprinting [99, 47]), which is beyond the scope of this thesis. Instead, this thesis proposes a hardware specialization model for delivering power-efficient acceleration that leverages on the customizability and reconfigurability of Field Programmable Gate Arrays (FPGAs). The acceleration model explored in this thesis aligns well to the coprocessor-dominated architecture (CoDA) paradigm proposed in [117, 141], and the heterogeneous multi-core architectures proposed in the International Technology Roadmap for Semiconductors (ITRS) report 2.0 [17].

This thesis introduces the DaCO (Dataflow Coprocessor Overlay) accelerator engine that is tuned to deliver high-performance on FPGAs. DaCO is a token dataflow overlay architecture that addresses compute bottlenecks found commonly in sparse graph problems, where irregular compute and memory access patterns limit achievable performance on modern commercial off-the-shelf platforms (*e.g.* desktop/server computers, graphics processing units). Some examples of such problems include circuit simulation [49], computational fluid dynamics [39], molecular dynamics [108], sparse convolutional neural networks [77], and more. When evaluated with benchmarks from some of these domains, DaCO demonstrates speedups of up to 2.8× over existing competing architectures like modern microprocessors and existing token dataflow overlays. We achieve this speedup by exploiting fine-grained irregular parallelism exposed by the dataflow graph and the dataflow triggering rule – instructions encoded in the dataflow graph execute asynchronously as soon as their operands are available. This event-driven model exposes raw parallelism in the benchmarks, and this thesis addresses the challenge of *managing* this form of instruction-level parallelism (ILP) effectively. There are three broad strategies that help achieve this goal: (1) large-scale out-of-order criticality-aware scheduling at runtime to prioritize computation along the critical path; (2) design of a criticality/priority-aware router that tunes the performance of the communication framework to the dataflow graph abstraction; and (3) software compiler optimizations that restructure dataflow graphs optimally to match the underlying hardware. Each of these strategies is discussed in greater detail in Chapters 3 – 5.

1.1 Contributions

We design a high-performance overlay architecture tuned for the Arria 10 FPGAs from the bottom up. The work described in this thesis takes inspiration from the token dataflow architecture, and improves existing designs by revisiting three major components: (1) the soft-processor, (2) the communication framework, and (3) the software backend (compiler).

Dataflow Soft-Processor: The token dataflow processor is vastly different to the traditional von Neumann microprocessor. Token dataflow processors operate in an event-driven mode, where arriving tokens at the input trigger arithmetic instructions in the processor pipeline, which leads to the generation of new resultant tokens at the output. This consumption and generation of tokens is dictated by a

dataflow graph, which represents the computation desired by the programmer. In Section 1.3 below, we give our motivations for picking the dataflow graph abstraction as an acceleration model for this work. Chapter 2 gives a literature review of existing dataflow-inspired processors, in particular the token dataflow processors. Existing token dataflow processors evaluate on an input dataflow graph in an *in-order* fashion, where tokens are processed in the same order that they arrive at the input. We demonstrate how this limits performance, and in Chapter 3, we describe the design of an out-of-order scheduler that overcomes this limitation. Our results demonstrate speedups of up to 2.8× over existing token dataflow processors and server-grade CPUs.

Communication Framework: The communication framework is an integral part in any distributed computing model, especially in the Network on Chip overlay architecture proposed in this thesis. There is a wide variety of options and design decisions that have to be made, which can dictate the behaviour, performance and size of the routing framework – some prime examples include switching technique (circuit vs packet switching), network topology (ring, mesh, torus, etc), arbitration strategy (dimension-ordered routing, wormhole routing, etc), and more. Chapter 4 gives a brief background on these design decisions, and builds motivation towards the design of a fast, FPGA-friendly priority-aware router that is able to discern and prioritize network traffic based on packet prioritiy levels. We call this design Hoplite-Q*, and we demonstrate its applicability to token dataflow overlay architectures in Chapter 4.

Software Backend: The final contribution in this thesis focuses on the software backend that maps an application to the token dataflow overlay. As with any computing system, good software is essential to achieving maximum runtime performance on the hardware, often exposed as optimization passes in compilers. Chapter 5 delves into three possible optimization opportunities that restructure an application dataflow graph to overcome various forms of limitations. This chapter quantifies the performance of each optimization pass incrementally on SPICE circuit simulation benchmarks, which serve as a case study for problems that are plagued by long sequential chains of arithmetic operations that are notoriously hard to parallelize. Overall, these software optimizations are capable of delivering up to 1.9–4.6× improved speedup on SPICE (Simulation Program with Integrated Circuit Emphasis) circuit benchmarks.

We call the complete ecosystem the **Da**taflow **C**oprocessor **O**verlay (DaCO) acceleration engine, and tune its performance and resource utilization to the Arria 10 AX115S line of FPGAs. DaCO relies on a host processor to generate and optimize dataflow graphs, which then get loaded and executed on the FPGA overlay coprocessor. Overall, we can scale DaCO to 600 processors on the target FPGA, operating at an F_{max} of 250 MHz.

1.2 FPGAs Today

Since its inception in 1984, the Field-Programmable Gate Array (FPGA) has come a long way, both in terms of capacity and architecture. With increasing capacity, FPGAs have been able to acquire larger market share year on year, as abundant logic/wiring resources enable increasingly larger designs to be synthesized to the FPGA fabric. Other redeeming factors such as low development costs, shorter time-to-market, field programmability, and more have allowed FPGAs to be competitive not only to ASICs (Application-Specific Integrated Circuits), but also to other programmable architectures like PLAs (programmable logic arrays) and CPLDs (complex programmable logic device). Architecturally, FPGAs have also changed radically since inception, adapting naturally to the demands of industry and the research community. For example, the lookup table (LUT) design has changed significantly, from simple three-input LUTs to the six-input LUTs packed in configurable logic blocks today. Another example of the architectural evolution of FPGAs is the addition of hard blocks like block RAMs (random-access memories) and DSPs (digital signal processors) to the soft/programmable fabric. These hard blocks have further enabled hardware architects to reduce the performance gap between ASICs and FPGAs, and deliver power-efficient computation with custom computing machines. For a more comprehensive description on the history of FPGAs, [122] is an excellent retrospective on the evolution of FPGAs.

In the past two years, FPGAs have taken another significant leap forward. Intel's acquisition of Altera in 2015 signaled a new era of FPGA-assisted computing. Today, FPGAs have started to appear in datacenters. Microsoft's Project Catapult [97, 19] paved the way forward for FPGA-assisted power-efficient computation on server-class workloads. Amazon's F1 instance [5] has also improved the accessibility of FPGAs to cloud computing developers, while programming frameworks like Reconfigure.io [101] and Pynq [98] have helped address the programmability

barrier associated with using hardware-description languages (HDLs). High-Level Synthesis (HLS) tools like VivadoHLS and LegUp [16] offer the power of static analysis and static scheduling to generate hardware from popular languages like C/C++, which has further improved the visibility of FPGAs as acceleration tools. Most recently, there has been a cambrian explosion of FPGA-accelerated inference accelerators for machine learning problems [126]. All of these developments point towards a future where FPGA-based coprocessors are an attractive option to meet the demands of next-generation computing. The work on DaCO proposed in this thesis is inspired with such a future in mind.

1.2.1 Arria 10 FPGA

In 2018, Intel announced the first server-class central processing unit (CPU) with an integrated FPGA: the Intel Xeon Gold 6138P [59]. The integrated FPGA on this device is the Arria 10 AX115S. One of the attractive features of this FPGA is the availability of the on-chip floating-point DSP blocks [123]. Implementing floating-point arithmetic on FPGAs is challenging, often resulting in bloated designs and long compute latencies when compared to fixed-point hardware. By hardening the floating-point computation as a configurable DSP block, the Arria 10 offers an enticing platform to build custom scalable compute engines that support IEEE compliant floating-point arithmetic. Hence, developing for the Arria 10 AX115S FPGA presents a compelling case and is the target FPGA for the work in this thesis.

Finally, the work in this thesis could also be synthesized to the Stratix 10 FPGAs, which also come equipped with the floating-point DSP blocks. Similarly, if floating-point computation is not required, DaCO can be synthesized to other families of FPGA devices as well. For practical reasons, we limit our scope to creating a **resource-balanced** DaCO architecture that is tuned for the target Arria 10 AX115S FPGA.

1.3 Why Dataflow?

A computer's architecture determines how data and instructions are handled, which directly influences where bottlenecks that slow performance down can be found. Today, most modern computers are designed based on the von Neumann model – control flow design with complex memory hierarchies (*e.g.* caches) and data synchronization protocols. Dataflow concepts have, nevertheless, creeped into several facets of modern computing, especially in compilers. To expose instruction level parallelism, compilers construct directed acyclic graphs (DAGs) from procedural code, which then gets optimized (where possible) and serialized into a list of instructions. On a von Neumann machine, these series of instructions are then executed one at a time, dictated by a program counter. Modern out-of-order scheduling circuits are also capable of extracting DAGs at runtime from limited-sized windows, which delivers instructions per cycle (IPC) gains at the cost of large hardware overheads. Instead of relying on dataflow principles yet operating sequentially under a program counter, why not operate explicitly on the dataflow graph itself?

In dataflow architectures, synchronization policies are greatly simplified as there are no program counters, while execution of instructions can happen in parallel asynchronously at each processor node, or processing element (PE). The dataflow model implicitly exposes parallelism and allows masking of memory access latency times with concurrent execution where possible. In modern microprocessor architectures, this effect is achieved with considerable hardware resources and/or software overheads.

Dataflow computers offers two key advantages when considered as accelerators:

1. *Power-efficiency*: The dataflow datapath is event-driven. Unlike the typical von Neumann processor pipeline, where the power-hungry instruction fetch/decode stage is active every cycle, a dataflow processor only needs a tracking circuit that wakes up when inputs are ready (*i.e.* an event). In the dark silicon age, where constantly switching gates every cycle is not feasible any longer, the dataflow trigger abstraction is an attractive strategy for

scalability. Combined with the asynchronous evaluation, the dataflow processor could theoretically achieve high ILP without complex, power-hungry overheads.

2. *Generality*: Unlike highly-tuned application specific accelerators, the dataflow graph abstraction offers generality to the programmer, as any computation can be effectively represented as a dataflow graph. Our goal in this work has been to provide a high-performance dataflow accelerator to which compute heavy traces of an application can be offloaded easily at runtime. With adequate compiler support, it would be possible to automate the identification, configuration, and acceleration of dataflow graph regions in an input application. This goal is similar to the programming frameworks like CUDA [91] for Graphics Processing Units (GPUs).

3. *Development Time* : Creating high-performance designs for FPGAs can be challenging, especially with RTL-based (register-transfer level) workflows using classic hardware-description languages like Verilog and VHDL. High-level synthesis (HLS) tools like VivadoHLS and LegUp [16] have helped lower the programmability barrier, but are still plagued by long synthesis and place & route runtimes. Often, these HLS tools require some degree of hardware expertise as well in order to create highly-tuned implementations. Additionally, as FPGAs continue to scale in capacity in the future, these designs have to be recalibrated, fine-tuned, and recompiled for the newer technology node. With overlay architectures, this responsibility falls solely on the overlay hardware architects, while the general programmer can reap benefits of fine-grained hardware acceleration with minimal development overheads. We believe that the simple token dataflow abstraction, together with adequate compiler support, offers a good programming model that minimizes overall development time for larger performance gains.

In the next chapter, we give a deeper background on the dataflow principles and challenges, and a brief overview of existing dataflow architectures.

Chapter 2

Background

This chapter gives a background to directed acyclic graphs (DAGs) and the basic principles behind evaluating these DAGs explicitly on hardware, also known as dataflow computing. We give a brief history and lessons learnt from the old and new dataflow architectures conceived since the 1970s. Finally, we end this chapter by giving a background on resources available on our target FPGA, and how we intend to utilize them.

2.1 What are directed acyclic graphs?

A directed acyclic graph is a collection of nodes (also known as actors or vertices) connected by edges (also known as arcs). In acyclic graphs, the edges can only be unidirectional, and feedback loops are not permitted, *i.e.* information in the dataflow graph only flows in one direction. Edges represent data dependencies between nodes, whereas nodes could have different meanings depending on the abstraction. For example, the simplest form of a DAG represents individual instructions (arithmetic/control/load/store) as nodes.

Figure 2.1 gives an example of a computation over an array expressed as a C-style for-loop, and Figure 2.2 shows one way in which we can construct the DAG of the computation by only using arithmetic instructions as nodes (*i.e.* no control flow in the DAG). Note that $ax^2 + bx + c$ is computed on an array x, and the equivalent DAG representation is a completely-unrolled representation. By unrolling the for-loop into a DAG, we obtain two benefits: (1) the overheads associated with

```
// for-loop that computes y = ax^2 + bx + c
for (int i = 0; i < 3; i++) {
    y[i] = a*x[i]*x[i] + b*x[i] + c;
}
```

FIGURE 2.1: Equation of a motivating example in C-style coding

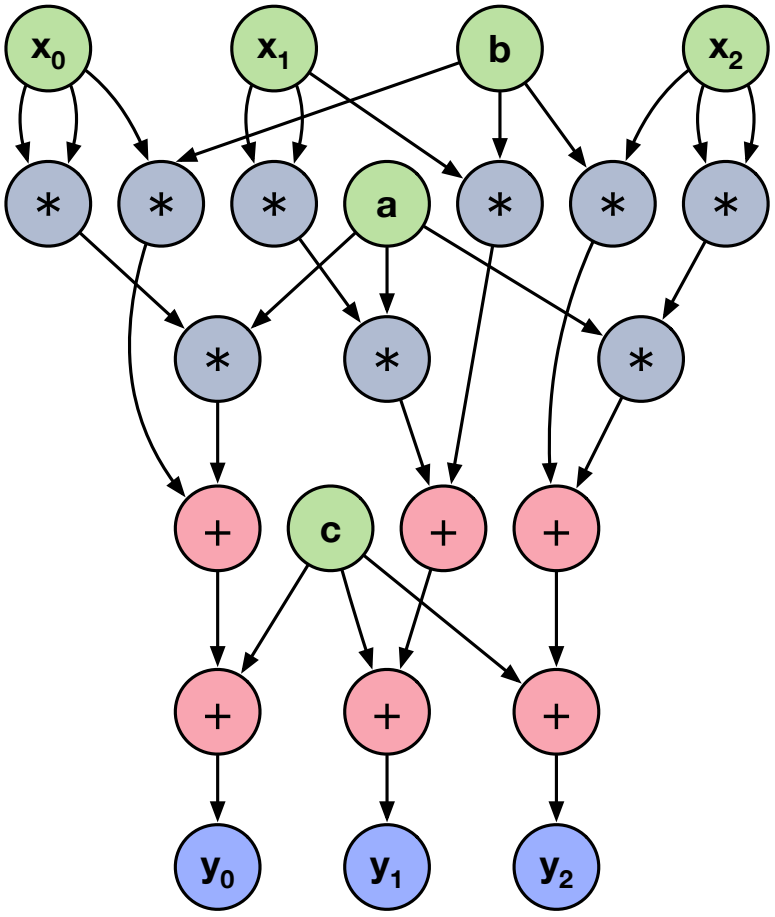


FIGURE 2.2: Unrolled DAG of equation in Figure 2.1

iteration control is eliminated, resulting in a raw instruction-level representation of the same computation; and (2) the DAG representation allows us to instantly identify nodes (*i.e.* instructions) that are side-effect free and can be independently evaluated in parallel (for example, the 6 multiplications at the top-level of the DAG can all be evaluated in parallel if we have $\geq$6 functional units available). Unfortunately, the DAG representation requires more instruction memory, which can grow exponentially with more complex computations if we choose to do a full-unroll everytime. Nevertheless, with careful compilation strategies and runtime management, it is possible to take advantage of the fine-grained parallelism offered

by dataflow computing. This is especially beneficial for large iterative applications, where we only unroll loops in bottlenecked regions of execution in order to deliver overall performance gains.

2.2 High-Level Synthesis

High-Level Synthesis (HLS) is an actively growing field [86] that has significantly improved hardware development time by relegating low-level hardware design decisions to the compiler. Programmers can express their computation in a commonly used language (typically C/C++, as seen in VivadoHLS and LegUp [16]), and with the help of pragmas, can guide the compiler to generate a desirable hardware implementation. The HLS tools can essentially be viewed as static dataflow schedulers that map and schedule blocks of computations (raw instructions or functions) onto the FPGA fabric. For small applications, individual instructions can be mapped to single logic elements (*e.g.* digital signal processors), whereas larger designs get folded and time-multiplexed for shared execution. This statically-scheduled style of dataflow is an attractive option, as it gives the programmer customizability and control over the final outcome. However, HLS tools cannot avoid the long runtimes associated with the hardware design flow (synthesis, and place & route). Generating a bitstream takes orders of magnitude longer runtime than simple DAG generation and optimization. Furthermore, the programmer has to be proficient in hardware design strategies to a certain extent, which limits the reach of HLS tools for mass adoption. Finally, as FPGAs scale, or design targets are changed, the HLS approach requires a recalibration and recompilation that can once again take up significant development and verification time.

Overlay architectures can address this limitation by trading away some of the control and customizability offered by HLS tools. In the next section, we do a brief overview of FPGA-based overlays that are well-known in literature.

2.3 FPGA Overlay Architectures

A popular way to virtualize FPGA resources is to design an overlay (also referred to as an intermediate fabric [23]), which is a custom programmable architecture

built using the underlying FPGA fabric primitives. The reconfigurability of the FPGA enables overlay designers to create customized high-performance overlay architectures that target specific domains or workload types. Overlay designers also provide simpler programming frameworks that allow users to program the FPGA at almost software-like turnaround speeds, unlike traditional HDL or HLS toolflows that are plagued by long place-and-route runtimes. There is, however, a performance tradeoff, since compared to hard-processors (integrated circuits baked into silicon using primitive gates/wires), soft-processors (built using "soft" logic in the FPGA, *i.e.* lookup tables, flip-flops, digital signal processors, etc) in the overlay are limited by the speed of the underlying FPGA fabric. For example, hard processor circuits can often be clocked $\geq$1 GHz, whereas FPGA overlays typically have an operating range in the hundreds of MHz. Nevertheless, factors such as design cost and adaptability make overlays an attractive option to leverage FPGA features. For example, one proposed way to raise popularity of FPGAs in data centers is by offering a suite of overlay architectures that lowers the programmability barrier associated with hardware design [124], while offering efficient domain-specific acceleration.

Overlay architectures come in many flavors depending on their target domain. Fine-grained overlays [80, 13, 71] create low-level primitives like lookup tables (LUTs) and multiplexers to deliver flexible and portable virtual FPGAs (*i.e.* FPGA on FPGA). These overlay architectures have been demonstrated to be useful in providing just-in-time (JIT) compilation for FPGAs [80, 81] or augmenting soft-processors with custom instructions that are portable across different FPGA devices [71]. Fine-grained FPGA overlays are also useful for carrying out FPGA architecture exploration.

Coarse-grained overlays [60, 124], on the other hand, are composed of more complex primitives, often referred to as processing elements (PEs). PEs could be von Neumann style soft-processors [20, 87, 137, 16], vector/single-instruction-multiple-data (SIMD) processors [105, 22, 106], or a reconfigurable array of functional units connected by a communication framework [44, 61, 69, 78] (also known as coarse-grained reconfigurable arrays – CGRAs). The communication framework can also be designed in various ways – *e.g.* crossbars [45], network on chips [45, 69, 95, 55, 67], or static/dynamic point-to-point connections [96, 51, 61].

There is a large body of work on FPGA-based overlays, as the increasing capacity, coupled with the programmability barrier, make overlays an attractive option

to virtualize FPGA resources. Detailed reviews/surveys on FPGA-based overlay architectures can be found in [113], [60], and [124]. Our work described in this thesis focuses on the token dataflow computing model, and can be viewed as an extension to the coarse-grained FPGA-based dataflow accelerator proposed in [69]. In the next section, we look at how we can design a statically-placed, dynamically-scheduled overlay based on the principles of dataflow computing.

2.4 Dataflow Computing

Unlike von Neumann architectures, dataflow computers have no program counters. Instructions are executed only when the node in the dataflow graph has received all its inputs/operands. There are no expensive synchronization policies as the basic dataflow firing rule implicitly guarantees functional correctness, and instructions (or nodes) can be executed in parallel asynchronously inside each processing element (PE). Here, each PE is a processor with a custom dataflow-inspired datapath that evaluates nodes and edges of the graph explicitly. On a distributed architecture, data is communicated via a communication network between all available PEs. The data is sent as tokens, or packets, which are typically composed of addressing and payload fields. This distributed dataflow computing model is known as a token dataflow machine, which is the architectural inspiration behind the work in this thesis.

For the DAG shown in Figure 2.2, the computation is kicked off by the constant-nodes (green nodes in the figure) in the DAG. As the constants' value is communicated to the nodes downstream as packets/tokens, the arithmetic instructions "wake" up as soon as all their operands are available, and the PE can then issue that instruction, packaged with its operands, to an arithmetic logic unit (ALU). If multiple ALUs are available (*e.g.* in a superscalar or distributed model), then nodes are evaluated asynchronously in parallel until the result-nodes (dark blue in the figure) are computed (*i.e.* no more outgoing edges from the evaluated node). Unlike the power-hungry instruction fetch/decode stage in von Neumann architectures that executes on every cycle, the dataflow paradigm uses a more power-efficient instruction-tracking circuit that is purely event-driven. The dataflow computer is also, in theory, more scalable, as there are no synchronization overheads associated with out-of-order execution of instructions.

Despite these advantages, dataflow computers have had limited success as general-purpose processors. The following sections give a brief overview/history of dataflow computers, issues in dataflow computing, and the state of dataflow computers today.

2.4.1 Dataflow Computing: Early Years (pre 2000s)

The very first dataflow designs were based on the *static* dataflow model. In the static dataflow model, also sometimes known as the pure dataflow model, arithmetic instructions are represented as nodes in a dataflow graph, sections of which could be evaluated in parallel by multiple processors. One of the earliest static dataflow architectures was the MIT Static Dataflow [32], proposed back in 1975. The static dataflow model, unfortunately, is limited in its suitability to general-purpose computing. In the static dataflow model, each edge can only have at most one token during the evaluation of the DAG. Hence, when faced with iterative loops (*e.g.* a for-loop), the static dataflow model does not permit parallel interleaved evaluation across loop iterations, as only one iteration can be active at any given time. This model is too restrictive for general-purpose computing, but nevertheless, it can still be usefully applied to derive improved performance on application traces.

The *dynamic* dataflow model was then proposed to counter this very limitation. In the dynamic dataflow model, multiple iterations of the same dataflow graph could be active in any given cycle. Some of the earliest dynamic dataflow machines were the MIT tagged-token dataflow architecture [24] and the Manchester Dataflow Computer [48]. Soon after, several improvements to the dynamic dataflow execution model were proposed, most notably the Explicit Token Store (ETS) [93], which addressed some of the crippling issues faced by the early dynamic dataflow machines. The ETS model inspired several dataflow projects, such as the Monsoon Project [93] and the EM-4 Chip [103]. We give a brief overview of each project below.

MIT Static Dataflow (1975) : The static dataflow model kicked off research into dataflow computing principles. The static dataflow model required a fairly simple architecture for execution: (1) memory cells held information about instructions (opcode and destination instruction addresses), (2) a control circuit that monitored the memory cells to dispatch ready instructions, (3) a pipelined network

route to the arithmetic units that executed the instruction, and (4) a distribution network that communicated and committed the results from the arithmetic unit into the target address in the memory cells. Since all instructions were mapped out statically, only one instance of each instruction could be active at a time *i.e.* only way to express iterative loops was to statically unroll the entire computation and map it to the memory cells. With computing technology still in its infancy back in 1975, the static dataflow model was viewed as very restrictive for general-purpose computing.

MIT Tagged-Token / Manchester Dataflow Computer (1978) : The tagged-token architectures were the first dynamic dataflow models put into practice. Unlike static dataflow, multiple data packets/tokens could be active on each arc/edge in dynamic dataflow graphs. This enabled compact representation of iterative code regions, and allowed the programmer to extract higher degrees of parallelism from the application where available. Each packet/token, aside from carrying a payload and destination address, also had a tag that identified the iteration number associated with the token. An instruction could only be sent to the arithemtic units when two operands with the same destination address *and* tag were found. If a match was not found, then the token had to be stored in a buffer. The Manchester Dataflow Computer used an associative memory, which enabled single-cycle tag-matching at runtime, but at a high logic utilization cost. Eventually, this tag-matching problem was the main reason behind the limited impact of the tagged-token dataflow model.

Issues with tagged-token dynamic dataflow architectures : There were several challenges associated with the early dynamic dataflow architectures. They were plagued by:

- An inefficient and expensive tag matching problem. In tagged-token dynamic dataflow, the tokens are each given a tag that identifies the iteration number of the token. In order to preserve functional correctness, operands have to be matched by their tags before being issued as an instruction. This resulted in the design of complex tag-matching circuitry that often relied on associative memories or multi-cycle pseudo-associative strategies, both of which required significant silicon real estate and dissipated a significant amount of power.
- Deadlock. In the event that an input token's tag is not matched, the token is then committed to a buffer where all unmatched tokens are stored. It

is easy to overcommit to this resource, which would result in a deadlock in the system, where computation in the pipeline is unable to proceed further. Some strategies to prevent deadlock involved designing very deep buffers (inefficient use of resources), or limiting the number of parallel iterations active at a time (restricting parallelism offered by dynamic dataflow). Eventually, both strategies failed to deliver competitive performance to make tagged-token dynamic dataflow viable.

- The tag-matching problem further added latency to the processor pipeline, which further restricted the performance impact of dynamic dataflow models.

Explicit Token Store : The Explicit Token Store (ETS) was a new model that was proposed to address the tag-matching problem. In ETS, a program was compiled into a collection of activation frames (akin to the basic block formulation used in modern compilers). Each activation frame has a fixed size and a detailed map of memory allocation for all tokens associated with the frame. While constructing activation frames was the job of a compiler, the allocation of activation frames was carried out dynamically at runtime. This ensured that a simple memory controller circuit in hardware could control the memory utilization at runtime, since the size of the activation frame is known upfront. Inside each activation frame, the instructions executed in dataflow order. Along with the payload, the tokens/packets carried a frame and instruction pointer field that indicated where the result was written to in memory. Each memory location also had presence bits, which act as the dataflow triggers for each instruction. Finally, unlike a stack, an activation frame could invoke multiple activation frames to become active at runtime, thus exploiting the dataflow graph-styled parallelism, but at the granularity of a code-block. A future direction of work for DaCO is to explore this codelet-based model for new applications like convolutional neural networks (a code-block could be a convolution, for example).

Monsoon (1990): The Monsoon architecture was a direct implementation of the ETS model. Monsoon came about as a joint venture between MIT and Motorola, with the goal of delivering a high-performance general-purpose dataflow computer. Monsoon applied the ETS model in a distributed computing topology, where a collection of processing elements evaluated hundreds of active activation frames in parallel, and communicated with each other using a packet-switched network. The dataflow principles, coupled with the ETS model, enabled fast context-switching

and operand matching, which in turn enabled each processing element to support a pipeline of tokens from different activation frames. Contrast that to the von Neumann architectures, where context switching can be expensive and hurt runtime performance. The largest implementation of Monsoon was a design with 8 processing elements with 8 I-structure processors (for managing the memory), which communicated with each other using a two-stage packet-switch butterfly network. Each PE ran at 10 MHz, and the overall system was capable of processing up to 10 million tokens per second. Issues reported with Monsoon were mostly in regards to idle cycles due to interleaving pipelines, and no prioritization support for critical portions of the application [50]. Eventually, the Monsoon processor, while initially promising, was unable to compete with the RISC/CISC (Reduced/Complex Instruction-Set Computer) processors of the 90s.

EM-4 (1992) : The EM-4 multiprocessor was developed by the Electrochemical laboratories in Japan. EM-4 introduced the concept of strong arcs – critical regions of the dataflow graph were connected by strong arcs instead of normal arcs. This allowed the processor at runtime to recognize a critical region for execution at runtime, and use local registers to store and prioritize the evaluation of the strong arc region. When a strong arc is detected, the processor pipeline is stalled until all the instructions in the strong arc region have been processed. The EM-4 combined the dataflow-styled processing for the overall graph with the control-flow styled execution for the strongly-connected-arc regions. The dynamic dataflow model adopted by EM-4 was similar to the ETS model described above, where each activation frame was instead referred to as an operand segment. Our work in DaCO has a slight similarity to the strong arc model in EM-4, as we provision for runtime hardware that prioritizes critical paths in the dataflow graph. However, unlike EM-4, our overheads are cheaper, and they do not stall the processor pipeline.

Threaded Abstract Machine (1993) : The threaded abstract machine (TAM) was radically different to the dynamic dataflow techniques proposed in this era. While most dynamic dataflow projects focused on the hardware aspects of exploiting dataflow principles, TAM focused on delegating the compiler with the task of exploiting dataflow-driven performance. The compiler's job is to create code blocks and determine the exact size of the memory required by each code block. The memory associated with each code block is referred to as its frame. A code block is a collection of threads, which in turn are a stream of instructions.

The threads use the allocated memory in a frame in a statically determined manner computed by the compiler. A continuation vector in the frame keeps track of the active threads. A cheap counter-subtraction strategy is used to ensure synchronization between threads, which can be likened to the dataflow trigger style of event-driven wakeup/synchronization. Again, all of these memory allocations are pre-computed by the TAM compiler. TAM demonstrates the power of good compilers, and serves as an inspiration for the compiler work described in Chapter 5.

Mini-conclusion on early dataflow : Most of the early research effort in dataflow computing was focused on dynamic dataflow architectures to develop general-purpose computing machines. Unfortunately, the difficulties in realizing high-performance general-purpose computers limited their impact. The technology node scaling of the 1990s also favored the von Neumann model significantly, as single-core performance rode the wave of frequency (Dennard [31]) and density (Moore [84]) scaling to mass adoption.

2.4.2 Dataflow Computing Today (2000s – Present)

As clock speeds began to plateau early in the 21st century, dataflow architectures began to receive renewed interest in order to keep up with Moore's Law, which continued to scale. With dark silicon limiting active chip area, the event-driven model of dataflow computing became an attractive alternative to the power-hungry pipelines of conventional von Neumann processors. Dataflow also found its way purely as a software abstraction in works such as [21, 102, 133, 33], as it exposed high degrees of ILP in sparse workloads. This century, so far, has seen far fewer implementations of pure dataflow computers, as most projects are opting to pursue dataflow in the form of a domain-specific accelerator [69, 70]. The merits of heterogeneous computing also motivates a coprocessor-dominated [116] future. SEED [90] motivates the role of tightly-coupled dataflow accelerators alongside traditional control-flow von Neumann processors, where application regions with some control/data regularity offer a sweet-spot for dataflow to shine. Other systems like DySER [44], Softbrain [88] and Maxeler [131] Technologies focus on spatial, streaming dataflow architectures for small, irregular traces that remain static for long iterative applications. We give a brief overview of two key dataflow-inspired recent works below.

WaveScalar [115] : WaveScalar brings the tagged-token dataflow paradigm into the 21st century by focusing on the communication network design and adding a customized cache (WaveCache) that manages computation at runtime. Computation in WaveScalar progresses in "waves", where a wave is a portion of an acyclic graph. The WaveScalar compiler breaks a computation down into these wave iterations, and a runtime system schedules and manages these wave computations. Overall, by optimizing the communication framework and exploiting dataflow locality, WaveScalar demonstrated some potential as a general-purpose processor. We focus on similar communication framework optimization goals with our work on DaCO, where we create a hierarchical-cluster communication framework to take advantage of the data locality property of real world benchmarks.

TRIPS [104] : TRIPS was developed at the University of Texas Austin, and is based on the EDGE (Explicit Data Graph Execution) ISA (instruction-set architecture). The EDGE ISA [15] organizes computation as blocks of instructions, where inside each block, the instructions execute in dataflow order. This coarse-grained representation is known as "static placement, dynamic issue", where both the compiler and the hardware work together to deliver better performance. At the block-level, the compiler constructs a dataflow graph of blocks to optimize the overall schedulability of the application, whereas, at runtime, once issued, the instructions inside each block progress in a parallel asynchronous fashion. TRIPS takes this EDGE ISA concept and adapts it to create a TRIPS ISA. Blocks in TRIPS have at most 128 instructions, and the compiler can schedule blocks to run on a particular processing element. TRIPS/EDGE serves as an inspiration for the next design iteration of DaCO, as we plan to extend the architecture to support a codelet-based token dataflow model.

Very recently, the Qualcomm research prototype dataflow chips revealed at ISCA 2018 were shown to be competitive with their conventional processors. The dataflow abstraction can be the foundation of an acceleration model for extracting high ILP from non data-parallel regions that are commonly found in sparse workloads [89].

2.5 Arria 10 AX115S FPGA

In this section, we give a short summary of the resources available on the Arria 10 (20nm SoC) AX115S FPGA, and our design objectives in regards to each resource.

ALM (Adaptive Logic Module) : Basic reconfigurable building block composed of two 6-input LUTs, four registers, and datapath control logic. The ALM supports up to any 6-input boolean function, and certain 7-input functions. We use the ALM to quantify the soft logic resource utilization of each design.

LAB (Logic Array Block) : A group of reconfigurable logic resources composed of 10 ALMs and interconnect. There are broadly three types of interconnect inside each LAB: (1) local routes from ALM-to-ALM inside the LAB, (2) adjacent routes from neighbouring LABs, and (3) global routes from other LABs on-chip via row/column interconnect. The Quartus compiler is responsible for packing logic optimally into LABs. The LAB can also be configured into a memory mode known as MLAB (memory LAB, described below).

MLAB (Memory LAB) : The AX115S has 20,774 out of 42,720 (≈49%) LABs which can be configured as MLABs. Each MLAB has a maximum capacity of 640b (64b per ALM × 10 ALMs per LAB), which can be configured as a 64x10b or 32x20b simple dual-port SRAM (static random access memory). The MLAB is a useful mode for creating pockets of small distributed memory in the soft processor logic, which can deliver savings to both on-chip hard memory block (M20K) utilization and read/write latency. On the AX115S, the total MLAB available capacity is 12,984 Kb.

M20K Block RAM (BRAM) : The AX115S only has one type of hard on-chip memory block: the M20K. Each M20K has a capacity of 20Kb, and with a total of 2713 M20K blocks, the total on-chip capacity of the FPGA from M20K BRAMs is ≈53Mb. Our goal is to maximize the resource efficiency of the M20K BRAMs, such that the largest possible graphs can fit completely on-chip. By eliminating/reducing off-chip data movement, we can achieve significant performance/watt improvements. Each M20K can be configured in simple/true dual-port mode, and can operate at a peak frequency of 645 MHz. For timing targets in the 250–300 MHz region, it is feasible to multipump (double-clock) the M20K blocks to create additional virtual read/write ports.

DSP (digital signal processor) block : The DSP block is the workhorse of the FPGA. It supports several different operating modes (*e.g.* fixed-point multiply-accmmulate, floating-point multiply, etc), which have to be statically configured at compile-time. While the fixed-point modes allow dynamic operations (*e.g.* dynamic subtract/negate), the floating-point mode does not support any dynamic control. Hence, we have to design our ISA carefully to ensure that we can execute all supported instructions in the dataflow graph. The floating-point DSP blocks can also be configured at compile time to have varying number of pipeline stages (supported: 1–4). This is a latency-F_{max} tradeoff, which can be tailored to the design requirements. In our case, a 250–300 MHz goal allows us to operate the DSP blocks at a pipeline depth of just 1 (*i.e.* latency of instruction = 1). Our goal is to achieve the highest possible throughput with DaCO, and hence, to maximize the resource efficiency of the DSP blocks as much as possible.

2.5.1 Resource Balance

TABLE 2.1: Ratio of resources on Arria 10 AX115S

	ALMs	**Regs**	**M20Ks**	**DSPs**
ALMs	1:1	1:4	157:1	281:1
Regs	4:1	1:1	630:1	1126:1
M20Ks	1:157	1:630	1:1	2:1
DSPs	1:281	1:1126	1:2	1:1
Total	427,200	1,708,800	2713	1518

Table 2.1 shows the available resources and their ratios to one another. The ratio between ALMs to M20Ks (157:1) and DSPs (281:1) is particularly challenging to design for, as that is a small budget for designing a fully-featured dataflow soft-processor. In order to balance the overall resource utilization, we allocate multiple M20K and DSP blocks to each processor. We also focus on maximizing the resource efficiency of M20K blocks – ideally, M20K blocks should only be used to store the dataflow graph in order to maximize the largest application sizes that can fit inside the on-chip memory.

Chapter 3

Dataflow Soft-Processor Design

3.1 Introduction

FPGAs have assumed an important role in modern computing systems through deployments in cloud environments like Microsoft Azure [19], and Amazon F1 [5]. New products like the Intel Xeon-FPGA 6138P hybrid SoC [58], and the Xilinx Everest [136] platforms further bolster their growing relevance. FPGAs are now firmly in the mainstream and have successfully demonstrated the long promised benefits of performance and energy efficiency of reconfigurable hardware.

To make FPGAs easy to program, vendors are investing in high-level synthesis (HLS) methodologies through programming languages like C/C++, OpenCL, as well as embedded design ecosystems like Xilinx PYNQ [98]. Intel's latest Xeon 6138P SoC with an integrated Arria 10 FPGA make it possible for software developers to easily offload critical portions of their software code to the FPGA attached as a tightly-coupled coprocessor. Another way to leverage this capacity is through soft-processor + network on chip (NoC) overlays such as the 1680-core GRVI-Phalanx system[45]. Thousands of small, customized soft-processors can deliver improved application-specific performance and energy efficiency, while reducing the parallel programming challenge and software development effort at the same time. The NoC interconnect backbone simplifies data movement and offers scalability and flexibility of integration.

Our goal is to demonstrate a FPGA-based design of a Dataflow Coprocessor Overlay (DaCO) that maximizes the resource efficiency (ALMs, M20Ks, DSPs) on an

Arria 10 FPGA to deliver a many-core dataflow acceleration engine. Figure 3.1 shows an example N-cluster DaCO engine. DaCO is a collection of customized dataflow soft-processors that communicate with each other over a hierarchical packet-switching network on chip (PSNoC) communication framework. Packets addressed to another processor in the same local cluster are routed by the local crossbar arbiter, whereas packets addressed to an out-of-cluster processor are routed over the Hoplite-Q* NoC [111]. While the local crossbar arbiter employs a round-robin arbitration strategy to route local packets in a single-cycle (best-case), the Hoplite-Q* NoC adopts a 2D unidirectional torus toplogy that is prone to longer packet communication latencies. This chapter focuses on the design of two of the three main components of DaCO :

1. Design of the dataflow PE in DaCO, and how out-of-order scheduling can be added to the soft-processor with minimal overheads.
2. Design of the crossbar that supports local communication in a cluster, and its impact on resource utilization and performance.

The design of the third component – the Hoplite-Q* NoC – is the focus of Chapter 4. All experiments in this chapter, unless otherwise stated, use the baseline Hoplite [64] NoC in order to isolate the effects of the improved PE design and/or clustered topology more clearly. A background on the Hoplite deflection router can be found in Chapter 4, Section 4.2.3.

3.2 Contributions

We propose several techniques to address the challenges of implementing dataflow soft-processors on FPGAs. Our contributions are:

- We devise a hardware-friendly criticality-aware OoO (out-of-order) scheduling technique that uses a bit-vector to capture node readiness which is then supported by a hierarchical lookup approach. This technique avoids squandering precious on-chip Block RAMs (BRAMs) on active-ready queues used by contemporary dataflow systems and instead frees them up to accommodate larger dataflow graphs. We use a static criticality-aware memory organization to pick the most important node for execution at runtime. Our hardware is able to

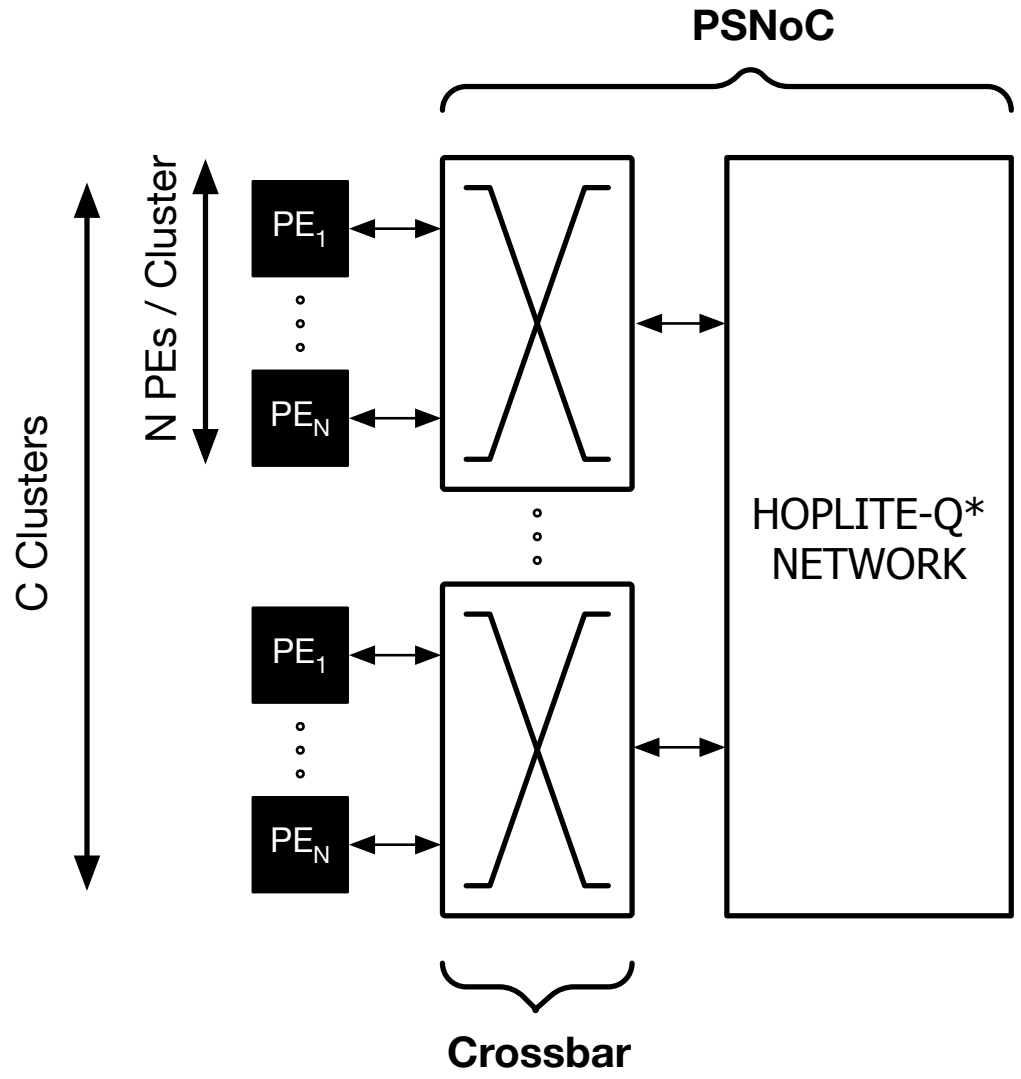


FIGURE 3.1: DaCO topology: C clusters of N processing elements (PE) connected by local crossbar arbiters; inter-cluster communication is supported by the Hoplite-Q* NoC.

select from 1000s of active nodes to determine the most critical ready node to process at runtime.

- We reduce the overheads of deflection routing in the FPGA-friendly communication networks by using a local crossbar interconnect to exploit data locality and route dependencies within the cluster much more efficiently. We adapt the PSNoC to support configurable clustering to determine the right balance of resource cost and dataflow execution time.
- We pay close attention to the ALM, M20K, DSP balance on the Arria 10 FPGA

to determine how to best provision resources in our DaCO array to boost compute density and memory efficiency. Alternatively, the DaCO overlay can be configured to support various combinations of resource balances desired to support other device mixes as well, which is beyond the scope of this chapter.

3.3 Background

3.3.1 The Good, The Bad, and The Ugly

The Good: Unlike conventional soft-processors, dataflow processors have *no program counter*. Instead, the token dataflow processor operates directly on a directed acyclic graph (DAG) using a simple dataflow firing rule: execute an instruction only when all its operands are available. This encourages the programmer or HLS compilers to expose concurrency directly in the form of dataflow graphs. In our abstraction, the nodes in a DAG encode an instruction, while the edges represent any data dependencies between these instructions. The edges can be viewed as communication *send* and *receive* instructions over the PSNoC. A DAG is partitioned and stored across multiple dataflow processors, and instructions execute in parallel at each dataflow processor independently. This dataflow-style parallelism is captured in the DAG representations, and is a very useful feature when parallelizing sparse workloads characterized by irregular instruction level parallelism and irregular memory access patterns (*e.g.* indirect pointer addressing).

The Bad: Dataflow implementations, however, introduce an out-of-order scheduling challenge at runtime. Fortunately, unlike out-of-order scheduling in existing CPUs, we have to tackle a simpler problem since the concurrency between instructions is already known upfront and does not need to be rediscovered. However, this still means that at runtime, at any given cycle, multiple unpredictable subset of nodes can be ready for evaluation. Figure 3.2 shows an example trace of a benchmark (`bomhof2`) on a 4x4 overlay instance – an average of up to 800 nodes can be ready *per processor*, which not only stresses the resource budget to maintain an active-ready queue, but could also inhibit overall performance if critical nodes are not prioritized for evaluation. The active-ready queue in existing dataflow soft-processors typically gets synthesized into long FIFOs (first-in first-out) using on-chip BRAMs, which is an inefficient use of the scarce hard block resource. Utilizing BRAMs for active-ready queues also limits the size of the dataflow graph

that can fit in the on-chip memory, further limiting scalability and performance of the overlay.

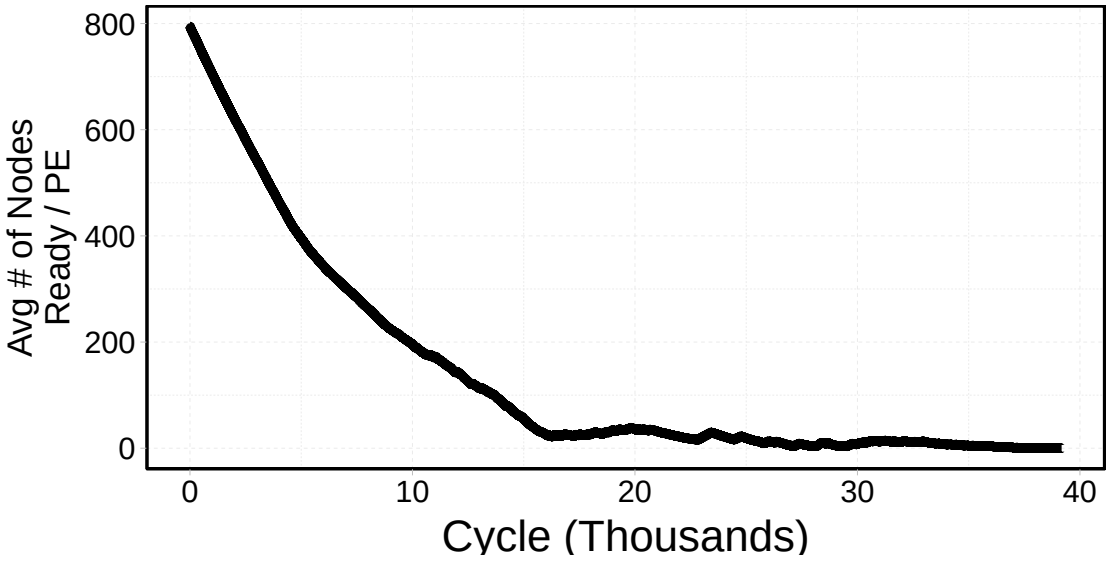


FIGURE 3.2: Average number of nodes ready / *per processor* at every cycle. Trace obtained by evaluating benchmark `bomhof2` on a 4x4 overlay instance (cluster size = 1)

The Ugly: The tail-end of the execution trace in Figure 3.2 shows a distinct lack of parallelism. An active-ready queue implementation of this structure will only store a few elements, and be massively underutilized. Furthermore, any data communication over a deflection-routed NoC like Hoplite-Q* will only exacerbate the packet routing latencies of latency-sensitive dataflow parallel evaluations in this critical, mostly sequential phase of the problem and prolong execution time.

3.3.2 Out-of-order execution

Modern microprocessors are typically based on the von Neumann computer architecture [125], which fundamentally, is an in-order serialized processing engine that executes instructions one at a time under a program counter. While the architecture is simple and elegant, the main reasons behind its popularity are steadfast improvements in semiconductor technology, strong market forces, and a stable software ecosystem that has promoted continuous research and development for the past four decades. When hierarchical memory organization (*e.g.* caches) were introduced to the von Neumann model in the 1990s, the popularity of OoO execution increased signficantly, as it enabled the processors to tolerate variable memory latencies across the different memory hierarchies. Today, as we approach

the physical limits of photolithography for semiconductor fabrication, OoO execution remains important to extract dynamic instruction-level parallelism for better overall performance.

OoO in von Neumann microprocessors was introduced with a centralized scheduling technique known as scoreboarding [120]. In scoreboarding, each issued instruction's data dependencies are tracked in a "scoreboard", which is a centralized data structure that maintains various status flags to identify and prevent runtime data hazards [49] (*e.g.* read-after-write, write-after-write, write-after-read). Instructions are issued from an instruction queue (IQ) as soon as it is hazard-free and an appropriate functional unit (FU) is available. Scoreboarding, however, is a centralized scheduling model that is susceptible to frequent stalls, as a single overcommitted FU can stall an instruction deeper in the queue that might have been free to execute on a different FU (*e.g.* a floating-point instruction on an floating-point unit).

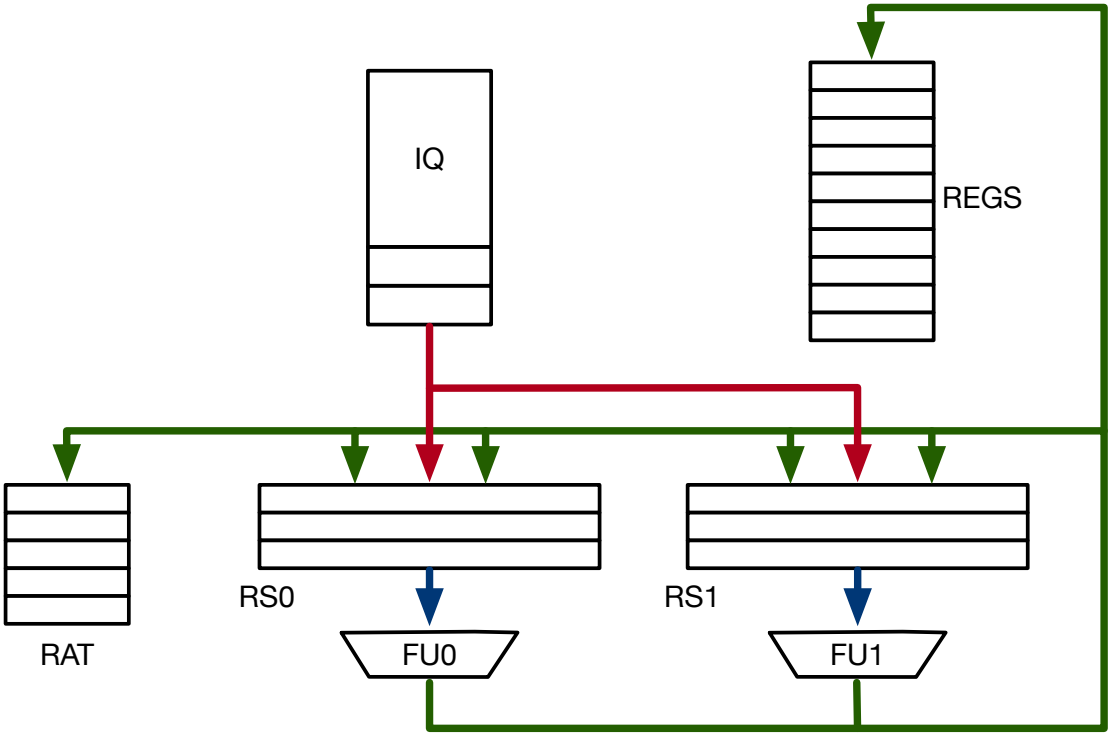


FIGURE 3.3: Tomasulo out-of-order execution circuit for an architecture with two functional units (FUs): Multi-Issue stage that commits decoded instructions in the instruction queue (IQ) into the reservation stations (RS0 and RS1), Dispatch stage that assigns an instruction to an FU, and Broadcast stage that transmits results on the common data bus (CDB) back to the FUs, the register allocation table (RAT), and the data registers.

The Tomasulo algorithm [121] addressed this issue by introducing a decentralized scheduling model that can capture out-of-order execution regions across multiple FUs effectively. Figure 3.3 shows a high-level picture of the hardware organization to implement dynamic out-of-order scheduling that is the basis even for modern micprocessors today. Unfortunately, this OoO circuit does not scale well with increasing number of FUs for the following two reasons:

1. The wiring density of the broadcast bus (also known as the common data bus) increases significantly as it must fan out to all FUs in the design.
2. The broadcast bus can only ever be used to transmit data by a single FU in each cycle, which gives diminishing returns as number of FUs is increased.

Modern microprocessors have typically hundreds (generally <500) of in-flight instructions executing in an out-of-order fashion (also known as restricted dataflow execution order), at the expense of area-expensive OoO execution hardware that has high dynamic power consumption.

Essentially, these techniques attempt to reconstruct small dataflow graph regions *at runtime* to expose instruction-level parallelism in the application despite the sequential ordering of instructions in von Neumann machines. Unfortunately, this comes at a signficant hardware overhead, as these designs require expensive circuitry in the form of reservation stations, allocation tables, reorder buffers, and complex data buses. Naturally, they are difficult to map efficiently on FPGAs due to significant logic and wiring requirements, and in the next section, we look at some of the existing related work on introducing OoO execution to FPGA-based soft-processors.

3.3.3 OoO in FPGA-based soft-processors

Most existing FPGA soft-processors are simple in-order processors (*e.g.* MicroBlaze [137], NIOS [87]), including existing token dataflow soft-processors [69]. Implementing out-of-order scheduling for FPGA-based soft-processors can be challenging due to the underlying FPGA substrate limitations (*e.g.* limited number of read/write ports on BRAMs). Well-known out-of-order scheduling techniques like scoreboarding [120] and Tomasulo algorithm [121] do not scale well on FPGAs,

often resulting in bloated and slow designs (>1k LUTs and <200 MHz operating frequency for 32-entry OoO scheduler [1]). Recent work [132] in this domain took on the challenge of designing FPGA-friendly out-of-order schedulers for traditional von Neumann soft-processors, which has improved the operating frequency significantly, and can now match that of state-of-the-art FPGA-based soft-processors like the NIOS II/f soft-processor (240 MHz). However, the resource utilization efficiency remains poor. The EDGE ISA processor [46] supports OoO within a fixed 32-size instruction window. Our work surpasses these previous attempts by supporting massive OoO (1000s of instructions) with a hierarchical scheme, which relies on static compiler-style optimizations that reduce the complexity of the OoO circuitry significantly.

3.4 Dataflow Coprocessor Overlay (DaCO)

In this section, we go into detail on the architectural implementation of DaCO. In particular, we focus on the design of the dataflow PE, and our efforts on adding out-of-order scheduling features to it. We also touch on clustering, and the design of the crossbar that supports communication between all PEs inside a cluster.

3.4.1 Processing Element (PE)

Each PE is a custom dataflow soft-processor built with five components:

1. On-chip node/edge memory,
2. Arithmetic logic unit (ALU),
3. Packet consumer,
4. Packet generator, and
5. A scheduler for managing computation at runtime.

Figure 3.4 shows these five modules and their layout within the processor design. Each PE communicates by sending/receiving packets to/from the PSNoC communication framework. The processor is fully-pipelined, which guarantees that a

new packet can be injected into the processor every cycle, *i.e.* a processor cannot backpressure the communication network. This guarantees a deadlock-free PSNoC architecture, as a packet is always allowed to exit into a PE, and ensure progression in the network. The scheduler is a key contribution in this chapter, where we improve upon the existing in-order scheduler to support large-scale out-of-order scheduling at runtime.

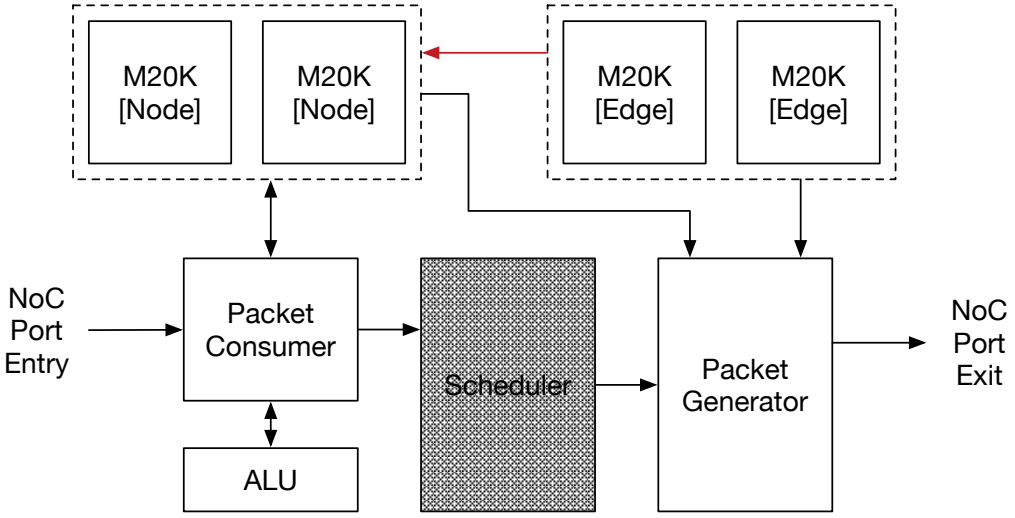


FIGURE 3.4: Dataflow soft-processor design

3.4.1.1 Node and Edge Memory

We fracture the dataflow graph memory into two distinct node and edge memory structures. This decision is motivated by the differing memory access patterns to node and edge state in the processor. Node state is read from and written to throughout the datapath, whereas edge state is read only by the packet generator module. We use M20K BRAMs for storing both the node and edge state inside each processing element. From our synthesis experiments, we settled on a design where each processor is allocated four M20K BRAMs, as it gives us a balanced resource utilization ratio (see Table 2.1). We allocate two BRAMs each to store node and edge state respectively. Due to the decoupled node/edge memory design, there is a small memory overhead to store pointer information that connects the two memory structures, which forms a coherent graph description locally inside each processor. Figure 3.5 shows how the node and edge state is packed into each addressable slice in a M20K BRAM. As a consequence, each M20K BRAM can pack 512 × 40b node slices or 1024 × 20b edge slices. There is an ≈14% overhead

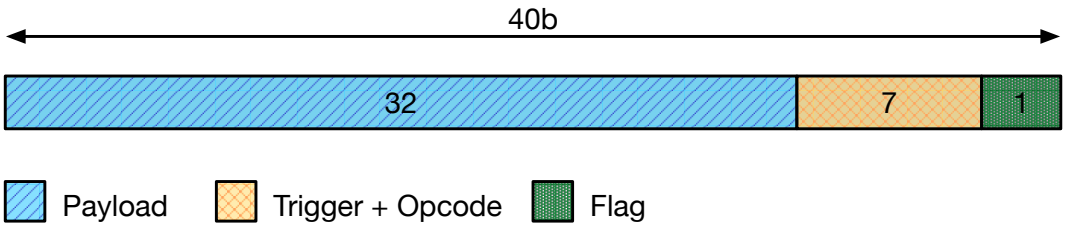
to store node-to-edge pointers, which is absorbed into the edge memory module. Overall, each processing element can pack 1,024 nodes and 1,536 edges. This balance is motivated by the observation that dataflow graphs in our benchmark set typically have more edges ($\approx$ 1.2–1.3×) than nodes.

Multipumping: Multipumping [73] is a well-known overclocking technique that can be used to create virtual read and write ports on the on-chip M20K BRAMs on the FPGA. We configure the node memory BRAM to operate in the single dual-port mode (1×read port + 1×write port) and multipump it to create a total of 2×read/write ports. This gives the advantage of having dedicated read/write ports for each stage in the processor pipeline, thereby simplifying the processor design logic and eradicating any non-determinism in the memory operations (*e.g.* due to the time-multiplexed sharing of the read/write ports). Multipumping incurs a small control logic overhead (≈30 ALMs) and easily achieves the target 250MHz system clock, since the hard M20K blocks can be clocked up to 645MHz.

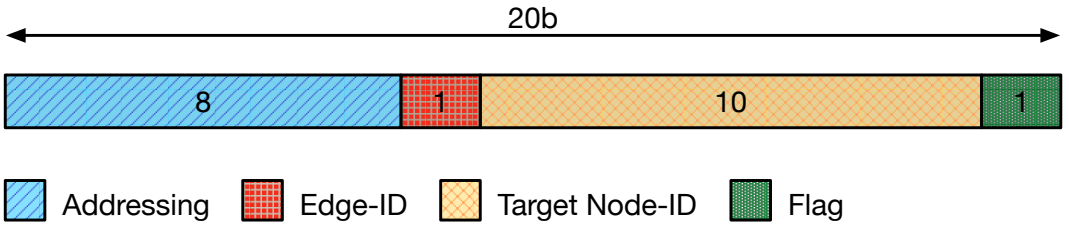
3.4.1.2 Arithmetic Logic Unit (ALU)

The ALU is the main workhorse of the processor and is responsible for executing the arithmetic instructions encoded in the dataflow graph. The hardened floating-point DSP (FPDSP) block is an attractive feature of the Arria 10 FPGA that enables high-performance IEEE-compliant *single-precision* floating-point computation without the need for instantiating high-latency, resource-heavy floating-point IP cores. The Arria 10 DSP blocks can be statically configured into several floating-point instruction modes, such as multiply, add, or multiply-accummulate. We provision three FPDSP blocks per ALU inside each processor and configure them statically at compile time to support the ISA (instruction set architecture) required for all benchmark types. In the future, if the abstraction of the DAG is changed (*e.g.* nodes in the DAG correspond to composite instructions instead), the ALU can be re-designed and re-synthesized to offer a new flavor of DaCO that supports a different ISA. We could also explore heterogeneity in the ALU design like in [110] (*i.e.* different PEs support different sets of instructions for area savings).

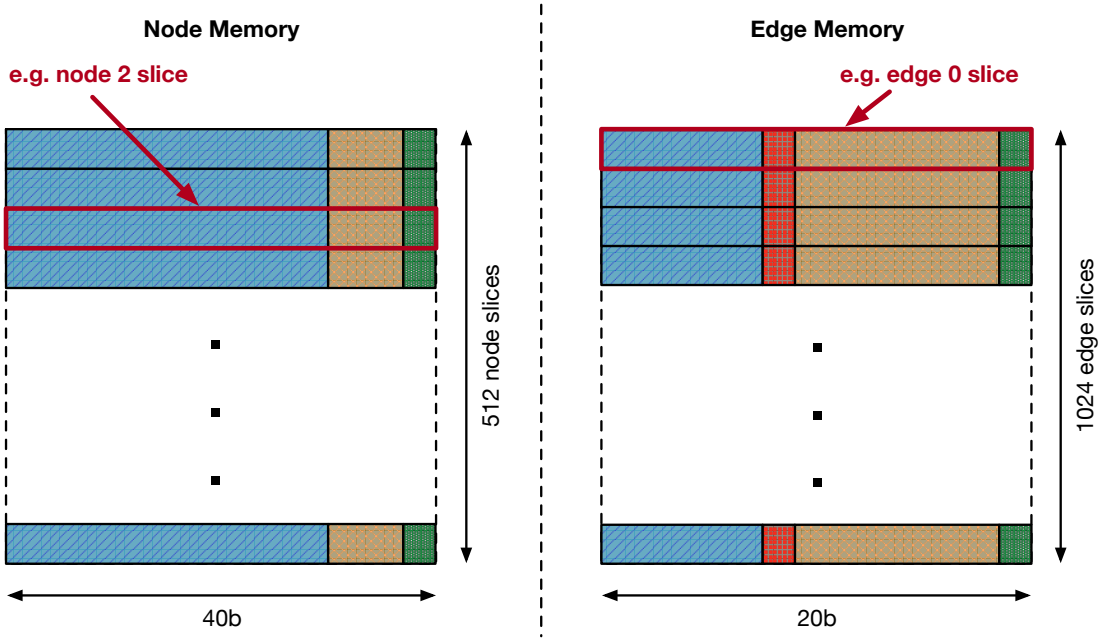
The number of pipeline stages inside the FPDSP can also be statically configured during synthesis to 1–4 stages, which affects the achievable clock frequency. We configure the FPDSP blocks to single-cycle mode, as their operating frequency



(A) Breakdown of how a node state is stored in a 40b addressable slice in a BRAM. The 1b flag indicates if the node has any outgoing edges.



(B) Breakdown of how an edge state is stored in a 20b addressable slice in a BRAM. The 1b flag indicates if the current edge slice is the final outgoing edge of the active node being processed.



(C) Visualization of how node and edge states are packed as addressable slices inside a *single* M20K BRAM in each PE. Note the different number of addressable slices in each data structure.

FIGURE 3.5: Breakdown and visualization of how dataflow graph data is packed as node and edge state data in M20K BRAMs on the Arria 10 FPGA.

still meets our clock targets, while simultaneously, reducing the processor pipeline depth to save logic utilization (discussed in the next section).

3.4.1.3 Packet Consumer

The dataflow processor is starkly different to a conventional microprocessor as there is no program counter or instruction memory to direct the flow of the computation. All instructions, encoded as nodes, are self-managing and they activate instructions downstream after being evaluated. These simple dataflow triggering principles are the key properties that expose implicit parallelism and scalability opportunities for many-core architectures without complex synchronization or shared-memory overheads. The packet consumer implements these dataflow triggering principles by managing the dataflow triggers of all locally-stored nodes in the node memory. This involves fulfilling 4 key tasks:

1. Storing the payload of arriving packets at each processor into the local graph memory,
2. Sending instructions, with payload, to the ALU when all operands of a local node have been received (*i.e.* node is ready to be evaluated),
3. Storing the result from the ALU back into the node memory, and
4. Notifying the next pipeline stage that a node is ready for edge evaluation to create new downstream communication packets.

When a packet arrives at the inputs of the dataflow processor, the packet consumer issues a read to the local node memory for the node slice addressed by the incoming packet (see Figure 3.5c). The node slice contains a trigger bit-vector (see Figure 3.5a), which is a small state machine that keeps track of the status of the node – *e.g.* given a 2b trigger,

- `00`: node has received no operands,
- `01`: node has received 1 operand,
- `10`: node has received both operands, and
- `11`: node has been completely evaluated and its result is stored in the node slice.

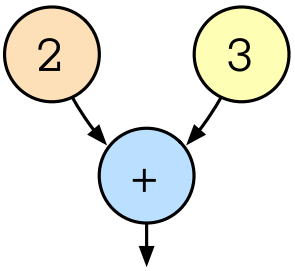
Based on the trigger value, the incoming payload is either stored in the local memory or bypassed directly to the ALU packaged as an instruction. Any valid computed output from the ALU is written back to memory and the active node's trigger value is updated. Overall, throughout the processor datapath, the packet consumer module tracks and updates the dataflow trigger of all nodes in the local

node memory. This can be likened to the scoreboarding [120] technique used in von Neumann CPUs, except in this case, the scoreboard is a distributed data structure that is managed by the packet consumer to handle data dependencies of *all* nodes local to the dataflow processor.

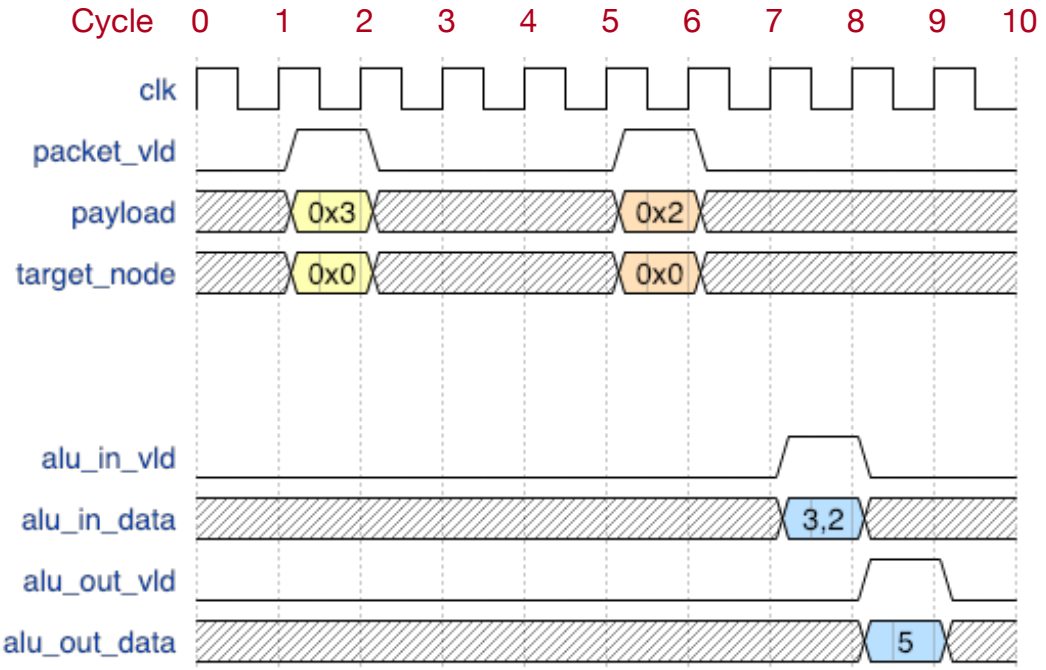
We design a 3-stage pipeline to implement the described packet consumer behaviour. Figure 3.6 shows a contrived example that highlights how this pipeline behaves at runtime. In this example, we focus on a single PE that manages the state of just a single `ADD` node, as shown in the dataflow graph in Figure 3.6a. The inputs in this case are nodes with values 2 and 3. These inputs are sent from another source (*e.g.* host CPU, or another PE) through the communication network, and arrive at different cycles, as depicted in the waveform diagram in Figure 3.6b. As soon as the final operand (*i.e.* 2) of the `ADD` node is received in cycle 6, the stored operand (*i.e.* 3) is fetched from the node memory, and together, both operands are issued as an `ADD` instruction to the ALU. Once the result is computed, it is written back into the node slice allocated to this `ADD` node (address 0x0). The dataflow trigger is updated throughout this operation, which is shown in the snapshots of node memory in Figure 3.6c.

The packet consumer datapath is fully-pipelined in order to support an initiation interval (II) of 1. This is achieved by designing a robust data-forwarding circuit that prevents read-after-write data hazards. When both operands targeting the same node are received close together (*e.g.* back-to-back cycles), we need to forward the payload of the earlier arriving packet up the pipeline such that the ALU instruction is issued despite an outdated trigger value being read by the later arriving packet. Since we have a 3-stage pipeline in the packet consumer, our data-forwarding logic multiplexes payload data across these 3 stages. This is a key design consideration, as the data-forwarding circuit consumes significant resources, and its resource utilization is proportional to the number of pipeline stages. Hence, our goal is to minimize the number of pipeline stages as much as possible, while supporting an II of 1. Note that the example in Figure 3.6 does not show this data-forwarding feature, but its design was important to achieving high-performance and functional correctness.

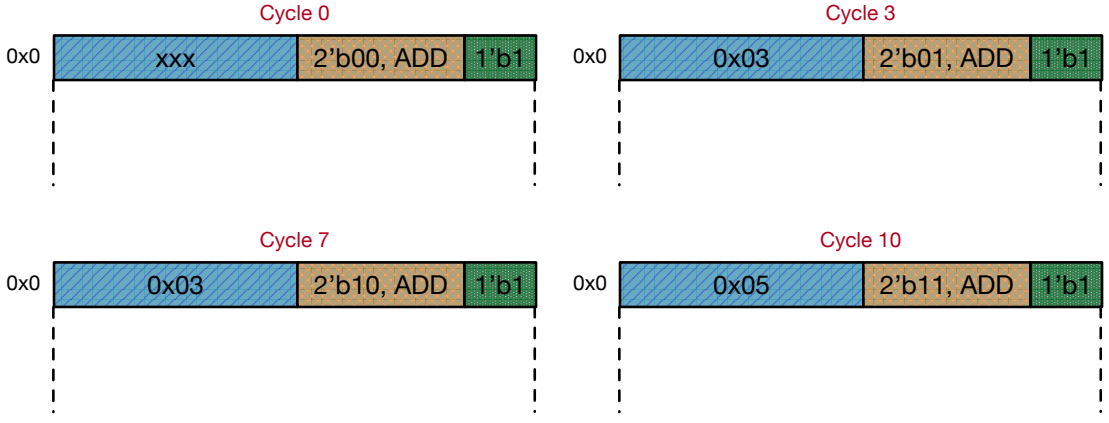
Finally, the packet consumer notifies the next stage that a node is ready for edge processing. This notification is dependent on the scheduling strategy that we discuss in detail in Section 3.4.1.5. The flag in the node slice (see Figure 3.5a) also plays a minor role at this stage, as the flag indicates whether the node has any



(A) Example: Dataflow graph with a single add node with values 2 and 3 as inputs. The node state is stored at address 0x0 inside the local node memory.



(B) Example: Waveform diagram of packets arriving at the PE targeting the add node in the example above, which is stored at address 0x0.



(C) Example: Snapshots of how the node slice at address 0x0 inside the node memory is updated at different cycles, based on arriving inputs as shown in Figure 3.6b.

FIGURE 3.6: An example dataflow graph and how the packet consumer handles incoming packets at runtime to update the relevant node states.

outgoing edges. A node with no outgoing edges does not need to be pushed into the scheduler pipeline, thereby saving cycles at runtime.

3.4.1.4 Packet Generator

Once a local node result has been computed and stored in memory, the result has to be communicated along its outgoing edges. The scheduler determines a ready node and passes its address to the packet generator. The packet generator is responsible for iterating over and generating a packet for each of the outgoing edges of all nodes. These packets are then injected into the PSNoC communication framework and routed to their destination as efficiently as possible. The packet generator is a finite state machine (FSM) that manages the entire process fluidly. The 1b flag in each edge state guides the state transitions in the FSM.

Since the 1b flag is only available once the read transaction from the edge memory is complete, back-to-back reads ahead in the edge memory are issued to avoid introducing stalls in the pipeline. When the final edge of the active node is found, all forward-reads are discarded and the node is marked complete by the packet generator as soon as the final edge is injected into the network. We implement these speculative edge memory read-aheads using a small 4-deep FIFO, and hence, the packet generator can support back-to-back packet injection into the network without stalls (assuming there is no congestion).

The FSM has a total of 9 states, and has a fixed setup cost associated with the evaluation of each active node. This setup cost is due to the memory read latency associated with fetching data from the node and edge memory. There are three distinct memory reads that need to happen for each active node:

1. Read active node's state, which is sent as a payload in each outgoing packet,
2. Read node-to-edge pointer value, which instructs the packet generator where the active node's fanout data is stored in local edge memory, and
3. Read packet header for each outgoing edge from the active node.

While reads (1) and (2) can be issued in parallel, read (3) cannot be issued until the node-to-edge pointer read is complete. Hence, this results in a fixed 4-cycle setup cost for evaluating each active node, after which, packets can be injected

into the network in back-to-back cycles. This set-up cost also gives us enough cycles to notify the scheduler that the active node has been completely processed and can be marked complete, and safely move on to the next scheduled node. We discuss this in greater detail in the next section.

3.4.1.5 Scheduler

At any given cycle during the graph execution, several nodes may be ready for edge processing to generate new packets (see Section 3.4.1.4). Picking an appropriate scheduling strategy is the key contribution of this chapter, as it can influence performance of the dataflow processor significantly, results of which are detailed in Section 3.6. We develop an out-of-order leading-ones detector (LOD) scheduling strategy and compare it to a naïve in-order FIFO-based scheduler to demonstrate the performance advantage of criticality-aware scheduling.

In the simplest implementation, we can connect the packet consumer and the packet generator with a sufficiently-deep FIFO: nodes that have computed their result are pushed into a queue, which is dequeued by the packet generator for edge processing. This essentially creates an in-order dataflow processor, i.e. input packets arriving at the processor and their respective packets on the outgoing edges are evaluated in arrival order.

However, in dataflow graphs, all nodes do not have an equal importance to the computational flow (*i.e.* some nodes are more critical than others), and scheduling nodes along the critical path with higher priority can ensure that the computation completes sooner. In addition, the naïve FIFO design is susceptible to unpredictable effects, such as network congestion, that can further delay computation along the critical path. Finally, the FIFO-based design also wastes precious M20K resources, as we have to provision for sufficiently-deep FIFOs to ensure that the network cannot deadlock. We observe the M20K(s) used as a FIFO to be severely under-utilized during the program's execution, and as such, the M20K memory blocks would be better utilized for storing graph (nodes and edges) data structures.

In order to implement a node scheduling strategy in hardware for dataflow graphs, we first identify the critical nodes in the dataflow graph. To achieve this, we run a **one-time** software pass that labels each node with a criticality heuristic. This criticality heuristic is based on a well-known slack analysis technique designed for

dataflow graphs, where a compiler statically assigns each node a real number, C_n, between zero and one that indicates its criticality in the dataflow graph. A C_n close to one indicates that the node is critical to the dataflow graph evaluation, and hence should be prioritized for evaluation inside the processor. Similarly, edges connecting to these critical nodes should also be prioritized in the communication network to achieve optimal performance.

The criticality heuristic, C_n, can be computed in 3 steps:

1. Label each node, n, with its as-soon-as-possible ($\mathrm{L}_{n,ASAP}$) and as-late-as-possible ($\mathrm{L}_{n,ALAP}$) latency.

2. Compute slack (S_n) at each node

$$S_n = L_{n,ALAP} - L_{n,ASAP}$$

   which indicates the scheduling freedom a node has in the dataflow graph. A higher slack indicates that the node is less critical to the compute flow.

3. Finally, compute C_n with the following expression: $C_n = 1 - \frac{S_n}{w_{ASAP}}$, where w_{ASAP} is the length of the critical path in the dataflow graph as evaluated by the ASAP algorithm. This expression gives a criticality-heuristic (C_n) that has a range between 0 and 1. A node with C_n close to 1 indicates that it lies on the critical path, whereas a node with C_n close to 0 indicates that we can schedule this node with more freedom.

Note that this criticality heuristic computation has to be done **only once** after generating the dataflow graph. This is a software-pass that is amortized over an iterative evaluation of the dataflow graph in the application.

Once we have identified the critical nodes in the dataflow graph, we sort the nodes, and their respective edges, in memory in descending criticality order (*i.e.* most critical node at the first address in node memory). This is an important step that reduces the challenge of picking the most critical ready node at runtime to simply picking the ready node with the smallest local memory address. That can be achieved in hardware by creating a bit-vector of 1b ready flags, where each bit has a one-to-one correspondance to each address in the node memory (*e.g.* if bit-7 is set to 1, it means that the node stored in address 7 of the local node memory is ready for evaluation by the packet generator). Then, we can simply feed this

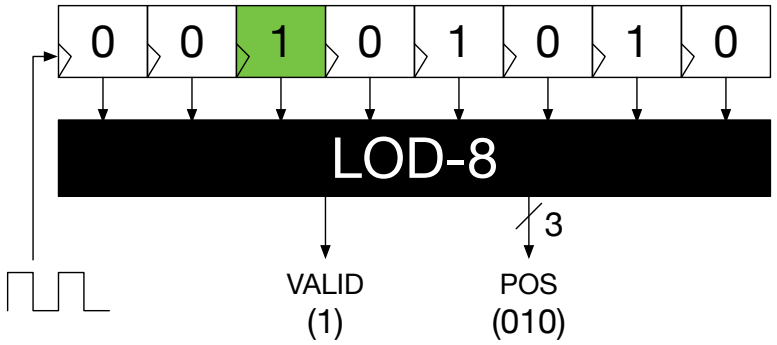


FIGURE 3.7: An example LOD-8 circuit in action, where the position of the first 1 (colored green) from the left in the input bit-vector is identified correctly at 010.

bit-vector into a leading-ones-detector (LOD) circuit, which would evaluate the position of the leading one in the input bit-vector. Figure 3.7 shows an example LOD-8 circuit that outputs the desired result. We write an LOD-based scheduler that manages the set and reset operations to the bits in this bit-vector, which come from the packet consumer and packet generator stages respectively.

However, the size of the LOD module needed is $\propto$ number of locally-addressable node slices in each processor, which is an intractable strategy for designing a single-cycle fully combinational LOD circuit. A fully combinational LOD circuit not only grows in size very quickly as the input bit-vector length increases, but also affects the system clock performance (*e.g.* f_{max} of an LOD-512 circuit drops to below 200MHz). In our experiments, when synthesized with this naïve strategy, the LOD-based scheduler takes up almost 800 ALMs on its own, which is a very inefficient use of FPGA resources. To address this challenge, we design a multi-cycle hierarchical LOD scheduler that trades off latency for a much leaner design and improved clock performance. Fortunately, the scheduler latency can be hidden at runtime since the packet generator is busy for a longer number of cycles servicing the current active node. Figure 3.8 shows the design of the LOD scheduler.

The scheduler is designed using two back-to-back LOD circuits of size N and M respectively. The node-ready flags are all stored in a MLAB memory structure, since the overhead to store all the ready flags is only 1024b (*i.e.* the maximum number of nodes per PE). MLABs in the Arria 10 FPGA are 640b simple dual-port memory structures made up of 10 ALMs each. Hence, the node-ready flags memory can be realized with a small 20 ALM overhead. The node-ready flags are fractured and packed as a N$\times$M memory structure, and the two LOD circuits are used to determine the most critical ready node in any given cycle. The LOD

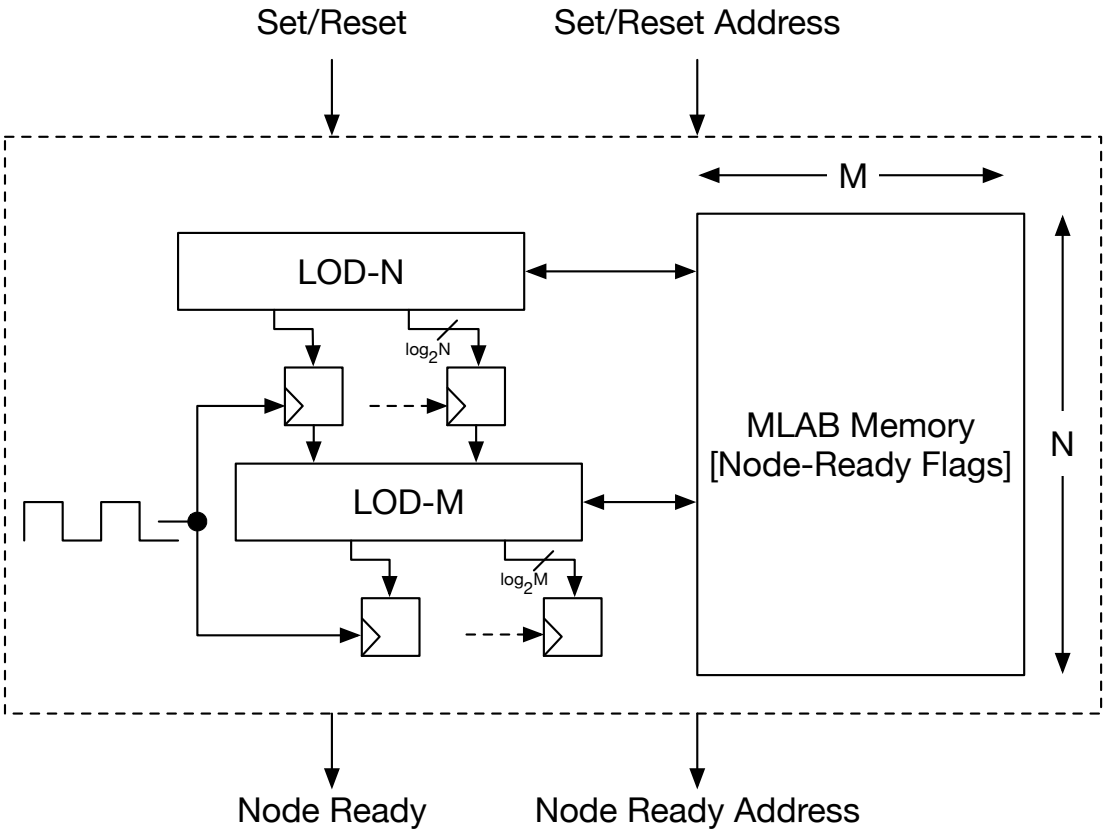


FIGURE 3.8: Hierarchical (depth = 2) LOD scheduler design

scheduler now has a latency of 3 cycles, but is fully-pipelined to support back-to-back set/reset operations. The choice of N and M can have a significant impact on the overall resource utilization and clock performance, and by doing a small design-space exploration, we determined N = 32 and M = 32 to give us the best performance-resource tradeoff.

Finally, decoupling the node and edge memory as described in Section 3.4.1.1 has a subtle positive effect on the LOD scheduler as well, since now we only need to allocate and track node-ready flags for addresses of the node memory. If nodes and edges were stored contiguously in the same memory structure, then the one-to-one mapping of bit-flag to memory address would have resulted in a larger, and highly under-utilized bit-vector (bits corresponding to addresses where an edge is stored is never utilized).

Later in Section 3.6, we compare the performance of the LOD-based scheduler shown in Figure 3.8 against the baseline FIFO-based in-order scheduler. We refer to the baseline PE with the FIFO-based scheduler as PE-Baseline, and the improved PE with LOD-based out-of-order scheduling as PE-DaCO.

3.4.2 Crossbar design in the PSNoC

As depicted in Figure 3.1, we adopt a hierarchical communication framework topology that we refer to as the packet-switching network on chip (PSNoC). The PSNoC is composed of crossbars to support intra-cluster communication, and a Hoplite-Q* network to support inter-cluster communication. The crossbar allows fast local communication between PEs inside each cluster at the expense of a higher resource utilization budget. The size of a crossbar grows quadratically with the size of the cluster and hence, at some point, we expect diminishing returns from moving to larger cluster sizes. The crossbar also puts a strain on the wiring resources of the FPGA, and hence, larger clusters also limit achieveable system F_{max}.

For a cluster size of N, the crossbar is composed of (N+1) round-robin arbiters – one round-robin arbiter for each PE in the cluster + one round-robin arbiter to the Hoplite-Q* network. An arriving packet from the Hoplite-Q* network is always given priority over other local packets, if there is a conflict. The round-robin arbiters are designed for fairness, such that requests are given an equal chance, on average, to receive a grant. This is achieved by updating the round-robin counter appropriately after each request is granted [130].

We partition our dataflow graphs in a cluster-aware manner such that a significant portion of the communication edges are absorbed into the richer crossbar network. This is achieved by using a high-quality partitioner in a two-step process, where the graph is first partitioned into clusters, following which the sub-graph assigned to each cluster is further partitioned to the individual PEs inside the cluster.

3.5 Methodology

3.5.1 Experimental Setup

All components of DaCO are designed and written in Verilog and synthesized using Quartus Prime v18.0. We use Verilator [112] for our simulation experiments, where we explore the impact of different DaCO configurations on performance – *e.g.* varying cluster sizes, enabling/disabling OoO scheduling, and choice of the NoC router. The synthesis experiments focus on achieving the right resource utilization balance on the target FPGA – factors such as LOD design, datapath

pipelining, memory instantiation, etc have an impact on the performance of the final design (latency, F_{max}, resource utilization). We ensure that the behavioural simulation matches the final DaCO design we settle on.

We run each benchmark with varying system and cluster sizes. We vary total system size from 1x1 to 16x16 (256 PEs), and group PEs into clusters of size 1–16 (powers of two). We write a C++ software backend that converts each benchmark into a dataflow graph, and generates the necessary configuration files to run each simulation. The backend is capable of analyzing the DAGs and producing criticality-aware optimizations described in this chapter. We use PaToH [18] to do cluster-aware graph partitioning.

TABLE 3.1: Properties of benchmarks evaluated in this study

Benchmark	**Nodes**	**Edges**	**Critical Path**
bomhof1	1925	2408	57
bomhof2	35609	45796	501
bomhof3	75305	90264	494
s953	37671	44052	171
s1423	52310	60852	470
s1488	86035	101608	655
s1494	86444	102060	628
hamm	115616	135416	337

We extract traces from eight different sparse matrix benchmarks from the circuit simulation domain. `bomhof` and `hamm` matrices are available from the Matrix-Market collection, while the remaining matrices are selected from the ISCAS89 benchmark set [14]. The traces are extracted from the sparse matrix factorization phase, which is evaluated millions of times in an iterative fashion, forming the compute bottleneck. The properties of benchmarks used in this study are tabulated in Table 3.1.

3.6 Results

We compare performance of the new DaCO overlay against a baseline token dataflow overlay that has no clustering and OoO scheduling features. We refer

to this design as *DF Baseline* in this section. As mentioned previously, all experiments with DaCO / DF Baseline in this chapter, unless otherwise stated, are carried out with the Hoplite router in the PSNoC. This is done to isolate the impact of criticality-aware scheduling inside PE-DaCO, and quantify the effect of the clustered-PE topology. More details on the Hoplite and Hoplite-Q* NoC can be found in Chapter 4.

3.6.1 Resource Utilization

Question: How does the resource utilization of the new OoO soft-processor compare to the older designs? Are the tradeoffs worth it?

TABLE 3.2: Soft-processor resource utilization breakdown

Sub-Module	**ALMs**	**Registers**	**M20Ks**	**DSPs**	**Clock (ns)**
ALU	16	17	0	3	2.9
Packet Consumer	89	301	0	0	2.0
Node Memory[1]	80	243	2	0	1.8
Edge Memory	16	64	2	0	1.6
Packet Generator	138	335	0	0	2.3
Scheduler (LOD)	433	279	0	0	2.2
Scheduler (FIFO)	61	116	1	0	2.1
Total (PE-DaCO)	779	1292	4	3	3.7
Total (PE-Baseline)[2]	457	1121	5	3	3.9

[1]Multipumped module

[2]BRAM usage scaled to match graph memory capacity of PE-DaCO

Table 3.2 gives a breakdown on the resource utilization of the soft-processors (PE-DaCO and PE-Baseline). PE-DaCO is ≈70% larger than PE-Baseline in ALM utilization. However, the 1 × on-chip BRAM saving, alongside the OoO scheduling capability, makes this tradeoff worthwhile in two ways: (1) OoO scheduling delivers up to 2.8× speedup over PE-Baseline overlay designs (see Table 3.3 and Section 3.6.2), and (2) as we scale overlay sizes towards maximum BRAM utilization, DaCO can accommodate 20% larger graphs in the on-chip memory than the DF Baseline overlay (≈ 700k/1.2m nodes/edges vs 550k/950k nodes/edges respectively).

With larger number of BRAMs per PE, PE-Baseline also limits maximum overlay size to ≈540 PEs, while DaCO can continue scaling up to 600 processors.

Finally, in other devices (*e.g.* Stratix 10) or future design iterations where number of BRAMs/PE balance ratio is higher, the number of BRAMs wasted on active-ready FIFO queues could potentially increase, thereby exacerbating the bad and ugly effects pointed out in Section 3.3.1.

Overall, the 70% ALM utilization overhead is a desirable tradeoff to make in return for performance and resource efficiency.

Answer: The new soft-processor design, PE-DaCO, takes up ≈70% extra ALMs than the original PE-Baseline design. However, the 1 × BRAM saving, coupled with improved performance and better resource efficiency/scalability makes this tradeoff worthwhile.

3.6.2 Overall Performance

Question: How does the overall performance of DaCO compare to the DF Baseline, and competing commercial off-the-shelf CPU solutions?

Note: Results with DaCO shared in this subsection are carried out with the Hoplite-Q NoC router, as we want to compare best-case performance with DaCO against other appropriate baselines.*

Table 3.3 summarizes the runtime results across all 8 benchmarks. All benchmarks, with the exception of `bomhof2` beat the CPU baseline when evaluated with DaCO. For two benchmarks (`s1488` and `s1494`), DaCO improves baseline performance significantly enough to overturn the performance outcome against the CPU baseline.

`bomhof2` presents an interesting outcome – despite offering the best runtime improvement over the dataflow baseline, DaCO still fails to beat the CPU baseline. On further investigation, we discover that the `bomhof2` trace has the least sparsity. While the non-zero density allows DaCO to exploit ILP better than the dataflow baseline design, the microprocessor is also able to take advantage of that significantly. In order to beat the CPU implementation in the future, we hypothesize

that a better locality-aware partitioning strategy can close the performance gap further.

Nevertheless, from a power-efficiency point of view, the FPGA-based DaCO can deliver more power-efficient computation. For ≈60W FPGA TDP vs 130W CPU TDP, `bomhof2` now delivers ≈50% improved GFLOPs/W performance over the CPU.

Answer: DaCO demonstrates improvements of up to 2.4× and 2.8× against DF Baseline and Intel Xeon E5-2680 implementations respectively. However, since modern day microprocessors and the linear algebra libraries are optimized for dense computations, it can be challenging to beat an optimized CPU implementation for low sparsity traces. Further investigation and optimization strategies should be explored to close this performance gap.

3.6.3 Effect of criticality-aware scheduling

Question: Is criticality-aware scheduling, on its own, useful?

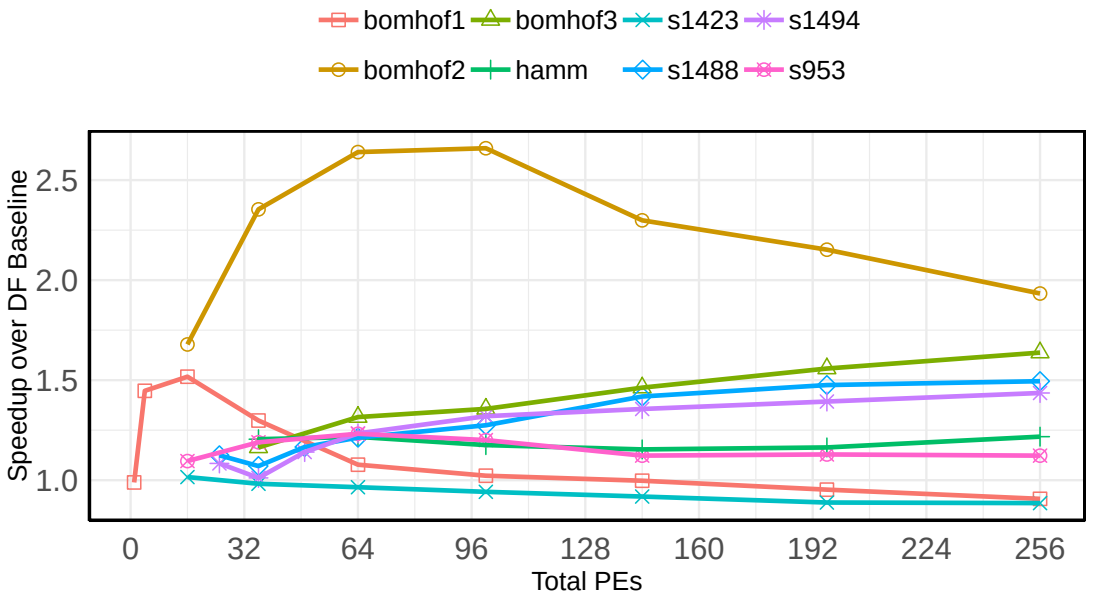


FIGURE 3.9: Effect of replacing Baseline PE (in-order) with DaCO PE (out-of-order). Cluster size fixed to 1.

Figure 3.9 shows the speedup observed when we isolate the effect of criticality-aware scheduling inside the DaCO PEs at varying system sizes. In this experiment,

the DF Baseline overlay is identical to the DaCO overlay, except for the criticality-aware out-of-order schedulers inside each PE in the DaCO overlay. We also turn off optimizations such as clustering and criticality-aware routing in the NoC to isolate the effect of criticality-aware scheduling in the PE as much as possible.

For most benchmarks, we observe an improvement of 1.1–1.6×. `bomhof2`, however, showcases improvements of up to 2.6×, while `s1423` *slows down* by ≈10%. As discussed earlier, the higher non-zero density in `bomhof2` allows DaCO to exploit more ILP. Furthermore, `bomhof2` has a relatively long and pronounced critical path (*i.e.* only few nodes lie along the critical path), which is better exploited by DaCO. In contrast, `s1423` has many parallel critical paths, as the average criticality across all the edges is ≈ 0.84 (vs 0.70 in `bomhof2`). Hence, criticality-aware routing does not benefit `s1423` as most nodes are of equal importance, and the performance loss is due to the multi-cycle scheduling overhead in the LOD-based scheduler and unpredictable runtime effects (*e.g.* congestion/deflection).

Answer: Yes, it delivers speedups from 0.9–2.6× at varying system sizes across all benchmarks. The slowdown can be attributed to the graph structure, where existence of many critical paths dilutes the effect of strong criticality-aware scheduling. This observation is benchmark-specific.

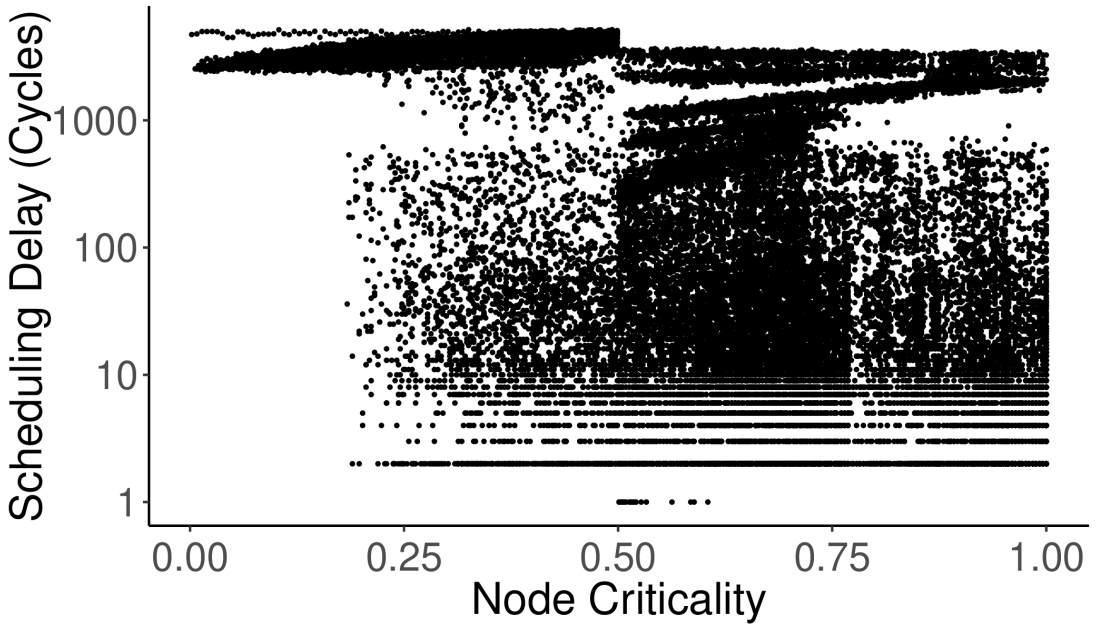
TABLE 3.3: Best-case benchmark runtimes with different types of PEs, compared against a baseline CPU implementation

	DF Baseline		DaCO				CPU[1]		
Benchmark	**PEs**	**Time (us)**	**C**	**N**	**PEs**	**Time (us)**	**Time (us)**	**vs Baseline**	**vs DaCO**
bomhof1	256	4.2 (1.0×)	16	16	256	3.8 (1.1×)	8.9	0.5×	0.4×
bomhof2	256	81.9 (1.0×)	16	16	256	34.4 (2.4×)	23.8	3.4×	1.4×
bomhof3	256	79.1 (1.0×)	16	16	256	36.2 (2.2×)	81.1	1.0×	0.4×
s953	256	15.8 (1.0×)	16	16	256	12.9 (1.2×)	32.0	0.5×	0.4×
s1423	256	30.4 (1.0×)	16	16	256	32.2 (0.9×)	48.0	0.6×	0.7×
s1488	256	83.5 (1.0×)	16	16	256	47.4 (1.8×)	70.8	1.2×	0.7×
s1494	256	77.1 (1.0×)	16	16	256	45.3 (1.7×)	69.5	1.1×	0.7×
hamm	256	48.0 (1.0×)	16	16	256	32.9 (1.5×)	93.5	0.5×	0.4×

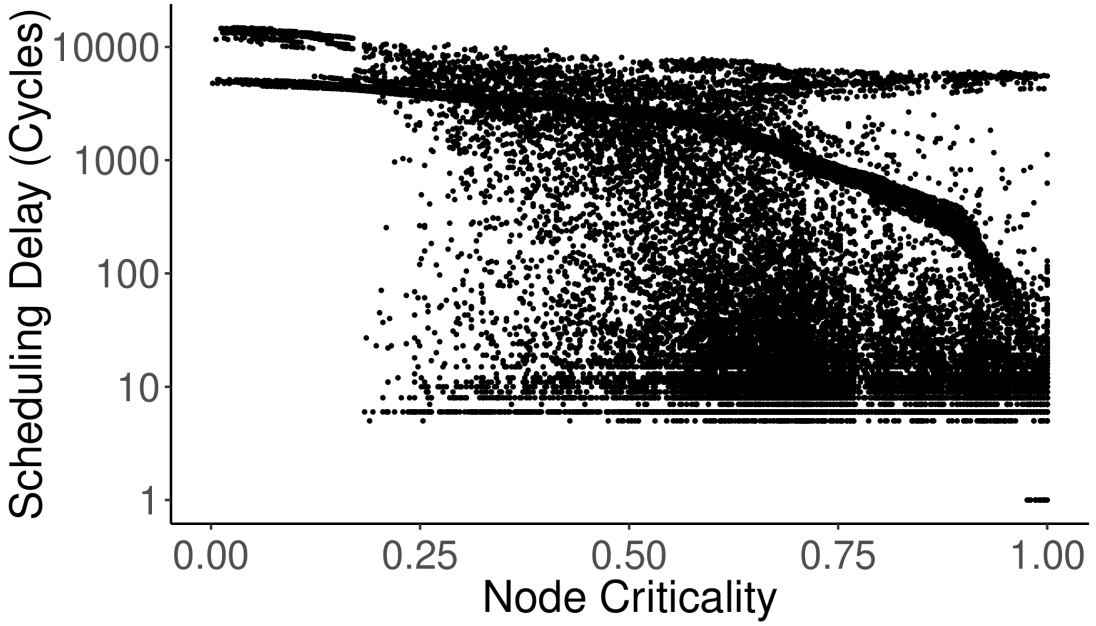
[1]Measured on an Intel Xeon E5-2680 using Eigen 3.3.4 Linear Algebra Library

3.6.4 Scheduler Efficiency

Question: How good is the LOD-based hierarchical scheduler at scheduling active-ready nodes by criticality?



(A) Scheduling delay suffered by nodes in PE-Baseline



(B) Scheduling delay suffered by nodes in PE-DaCO

FIGURE 3.10: Scheduling delay suffered by node vs node criticality (`bomhof2`)

Figure 3.10 shows the scheduling delay suffered by nodes in an example trace from `bomhof2`. The LOD scheduler produces a desirable criticality-aware scheduling trend, where nodes on the critical path are prioritized for scheduling, which is unlike the FIFO implementation. There are, however, some high-criticality nodes in PE-DaCO that still suffer from large scheduling delay (> 5000 cycles). That is due to the long sequential tail of the dataflow graph, which is unavoidable in the current implementation.

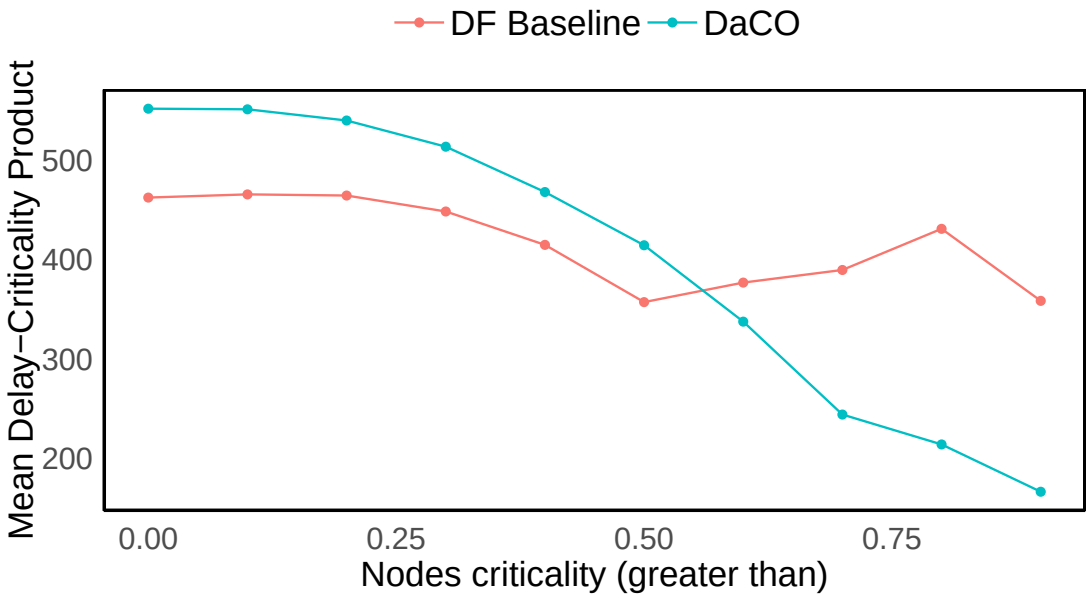


FIGURE 3.11: DF Baseline vs DacO on `bomhof2` benchmark : Average delay-criticality product of subset of nodes with criticality greater than (x-axis).

A rudimentary delay-criticality product (DCP) heuristic can be computed for different sets of nodes to estimate the effectiveness of the criticality-aware scheduling in DaCO. Figure 3.11 shows, for the experiment in Figure 3.10, that DaCO prioritizes the critical nodes ($\geq$ 0.9 node criticality) by $>$ 2$\times$ for the DCP heuristic. Note, despite an overall higher scheduling delay observed for all nodes (*i.e.* node criticality $\geq$0), DaCO still delivers significant speedups over the DF Baseline implementation, further highlighting the importance of criticality-aware scheduling. The extra scheduling overhead introduced by DaCO can be attributed to the multi-cycle hierarchical LOD-based scheduler described in Section 3.4.1.5, compared to the simpler single-cycle push-pop read/write model of the FIFO.

Answer: The LOD circuit, unsurprisingly, performs better than the naïve FIFO scheduler ($>$ 2$\times$ better at scheduling high-criticality nodes), which highlights the importance of criticality-aware execution. However, the long sequential tail, i.e. *chain of data-dependent arithmetic operations, at the end of the dataflow graph limits the effectiveness of the LOD scheduler. In the future, dual-issue packet generators or tighter instruction-coupling in the datapath could help overcome this limitation.*

3.6.5 Effect of clustering

Question: Does clustering improve performance? What cluster size offers the best resource utilization vs performance tradeoff?

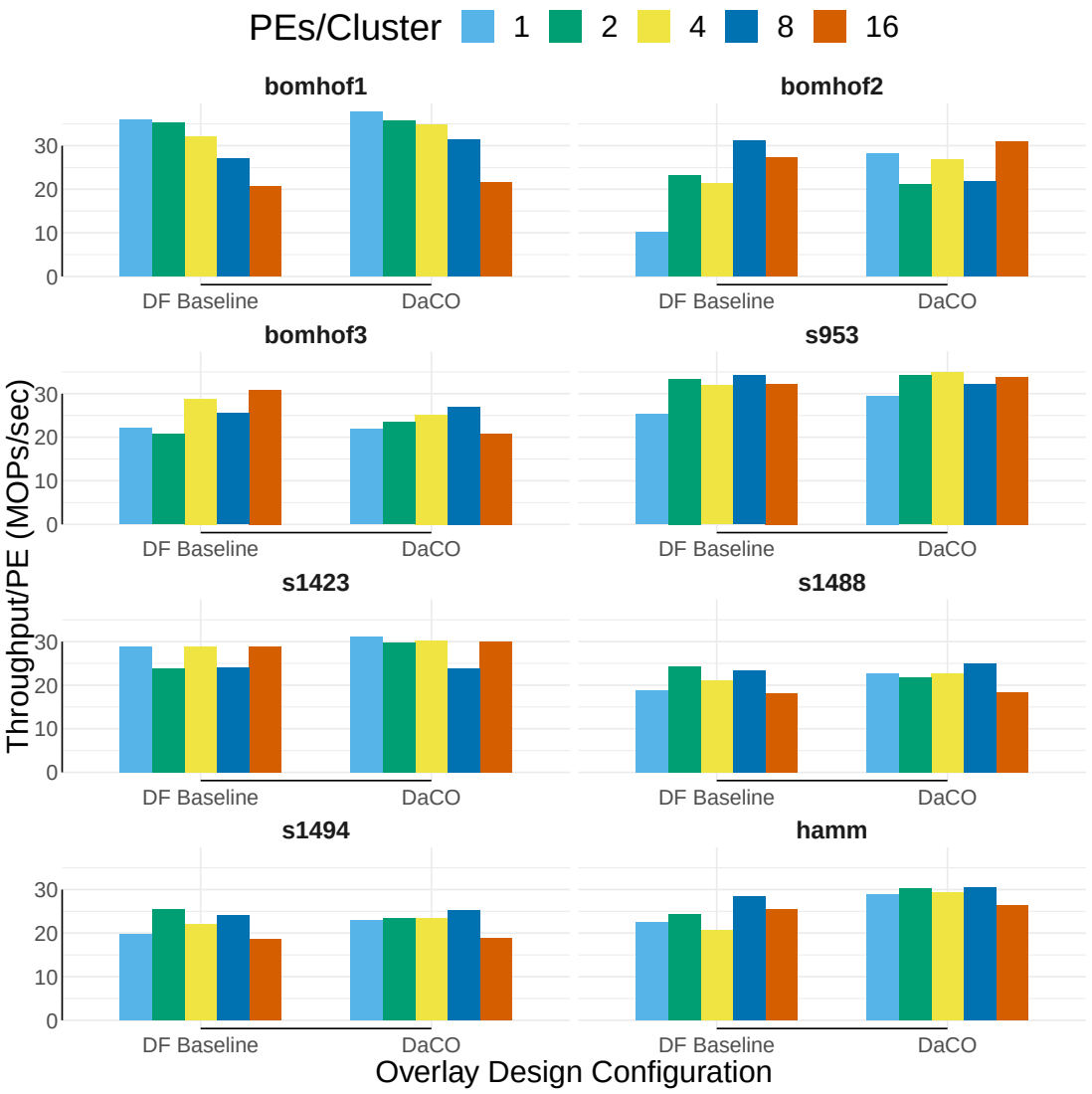


FIGURE 3.12: Throughput/PE with varying cluster size across different benchmarks.

In Figure 3.12, we show the effect of clustering on both the DF Baseline and DaCO across all benchmarks. For small benchmarks like `bomhof1`, few PEs in a small cluster configuration deliver the best overall throughput efficiency. DaCO delivers better throughput/PE for most data-points, while clustering delivers significant benefits to DF Baseline implementations as well.

Note that the total number of PEs for each benchmark/cluster-size configuration in Figure 3.12 may not necessarily be the same, as we pick the best performing system

size for each cluster size/benchmark combination in the plot. This discrepancy is a contributing factor to the non-monotonic trend observed in the processor efficiency in Figure 3.12. There are potentially two other factors behind this observation as well:

1. Each benchmark has its own sparsity pattern and node/edge distribution. Smaller benchmarks like `bomhof1` predictably prefer small overlay sizes without clustering, as there is not much communication to optimize in the first place. As benchmarks become larger, the DAG structure also has larger variance in graph properties (*e.g.* number of nodes, edges, and critical path). This variance could also be contributing to some of these observed non-monotonic trends.

2. Clustering introduces a partitioning problem where our objective is to localize regions of computation to clusters as much as possible. We implement this currently in a naïve greedy manner, where PaToH is first used to partition the dataflow graph into clusters, and then the sub-graph assigned to each cluster is further partitioned into its member PEs in a greedy fashion. This entire partitioning strategy is agnostic to the NoC topology, and can be improved for more predictable behaviour.

Table 3.4 shows the resource utilization overhead of the crossbar with different cluster sizes. As expected, the size of the crossbar grows quadratically to cluster size, thereby incurring significant area overheads at larger cluster sizes. From Figures 3.12, 3.13, 3.14, and Table 3.4, we conclude that a cluster size of 2 or 4 delivers the best resource-performance tradeoff, delivering up to 1.5× throughput improvement for 15–40% ALM overhead.

Answer: Yes, clustering improves performance as expected, as low-latency communication between local PEs supported by a crossbar helps to improve overall throughput. However, at a large cluster size of 16, resource efficiency is low as the throughput per PE degrades below that of smaller cluster sizes. Furthermore, since crossbar resource utilization grows quadratically with cluster size, we find that the best performance-area tradeoff occurs at cluster size of 2 or 4 (benchmark-dependent), where we obtain up to 1.5× throughput improvement for 15–40% ALM utilization overheads.

3.6.6 Performance vs Resource Utilization

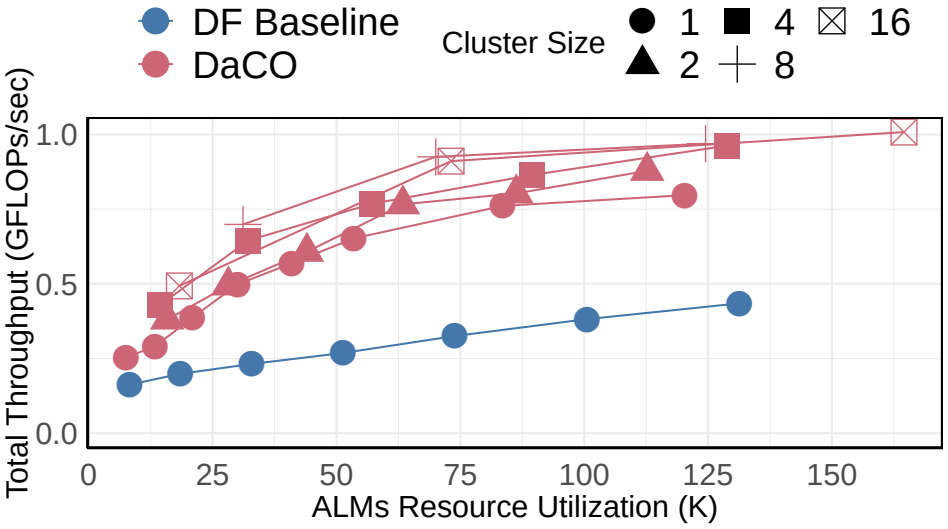
Question: How does the raw GFLOPs/sec throughput performance scale with resource utilization in mind?

TABLE 3.4: Resource utilization breakdown (ALMs) and clock performance (ns) of the crossbar as cluster size is varied.

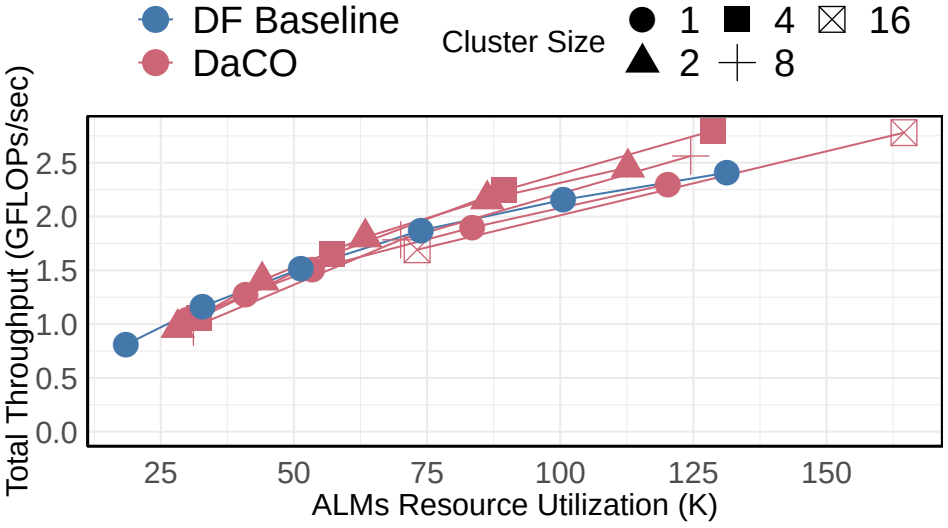
Cluster Size	Round-Robin Arbiter	Muxes	Total	Clock (ns)
2	4	140	147	3.7
4	28	376	404	3.7
8	245	1248	1493	3.8
16	960	4802	5762	3.9

Figures 3.13 and 3.14 show the total throughput against ALMs and M20K BRAMs for three representative benchmarks. Overall, DaCO delivers better overall throughput with the same resource budget, especially when considering M20K utilization. On `bomhof2` and `hamm`, DaCO improves peak GFLOPs/sec throughput by up to 1.2–2.4×, while the more stubborn `s1423` benchmark shows 0.9–1.0× throughput against the DF Baseline. This observation motivates the design of a selective offloading strategy that only schedules acceleratable dataflow regions in an application to the DaCO coprocessor on the FPGA.

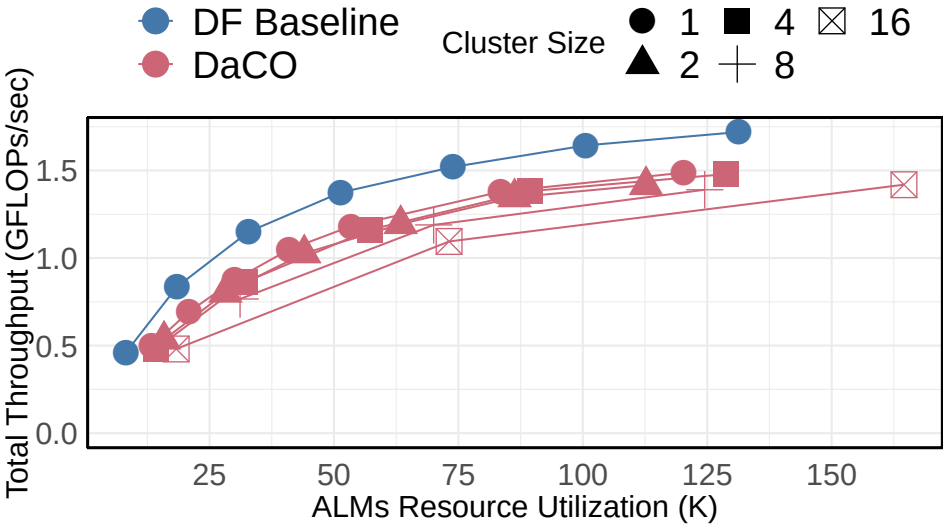
Answer: DaCO improves throughputs by 0.9–2.4× over the DF Baseline as overlay size is scaled across three representative benchmarks. However, a careful strategy to identify and schedule desirable dataflow regions is important to ensure that the entire application benefits with this coprocessor style execution.



(A) `bomhof2`

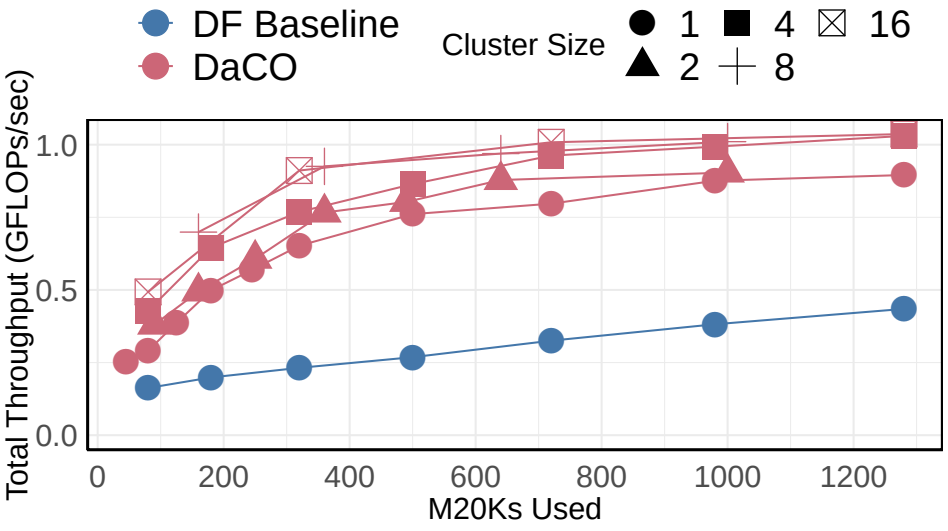


(B) `hamm`

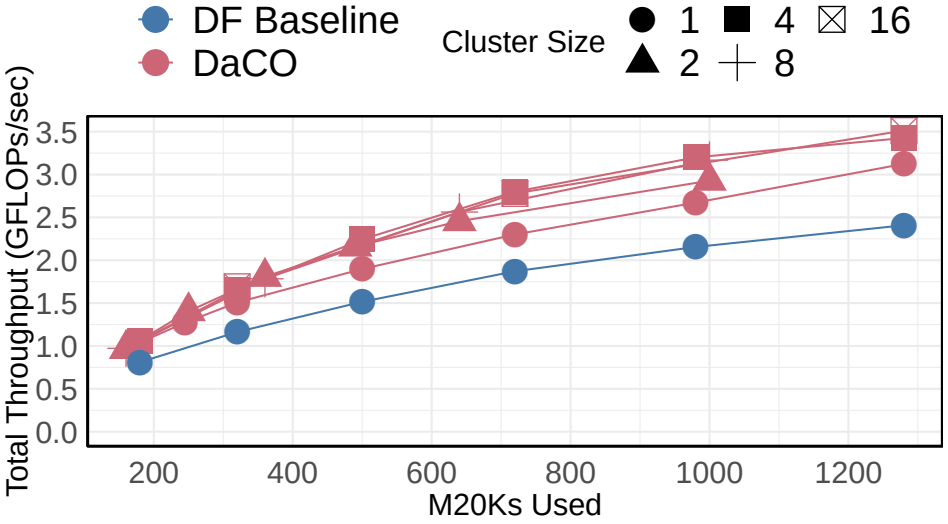


(C) `s1423`

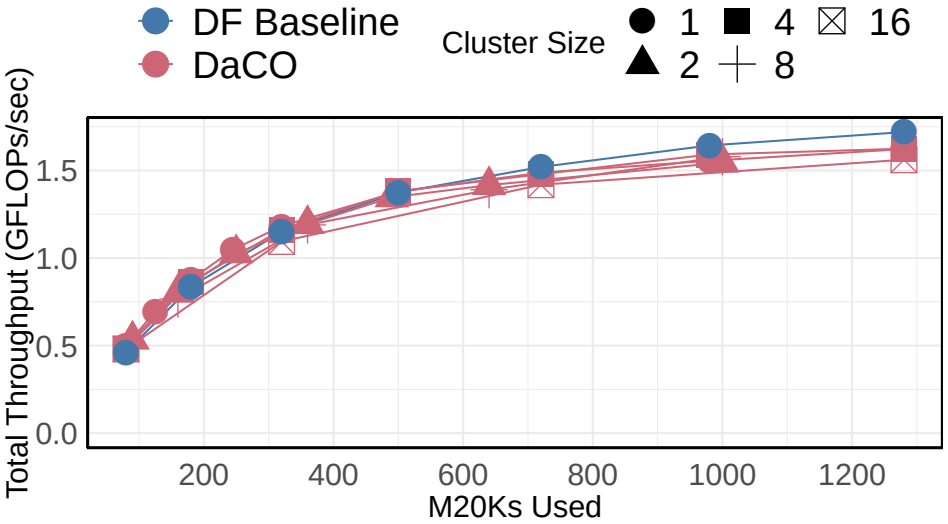
FIGURE 3.13: Total throughput vs ALM utilization observed on three representative benchmarks with varying system sizes



(A) `bomhof2`



(B) `hamm`



(C) `s1423`

FIGURE 3.14: Total throughput vs M20K utilization observed on three representative benchmarks with varying system sizes

3.7 Future Work

While designing the DaCO soft-processor, several important observations gave us motivations for future improvements. We list the top three improvements that we hope to make to the DaCO soft-processor in the near future:

1. The 4-cycle setup cost in the packet generator can be avoided by performing an area tradeoff, where we replace part of the finite state machine with a pipeline of active-ready nodes instead. By multipumping the edge memory as well, we can create additional read ports that service this pipeline. As such, as soon as an active-ready node's packets have been completely generated and injected into the network, the next active-ready node in the pipeline and its packet headers are already ready for injection, thereby achieveing back-to-back packet injection *across nodes* in the packet generator. This would help mask the 4-cycle latency in the packet generator, and improve the offered throughput into the communication network.

2. We could also create a dual-issue packet generator design that is capable of processing two active-ready nodes at a time. This would also improve the dataflow injection rate and improve the sustained throughput in the communication network. This optimization would be especially beneficial at the start of the DAG evaluation, as we observe a large number of active-ready nodes in the beginning (see Figure 3.2).

3. To address the long sequential tail at the end of dataflow graphs, a tightly-coupled datapath could improve the performance of the overlay. In the scenario where a sequential chain of data-dependent instructions is identified inside a processor, a special datapath could be designed that evaluates this chain in a back-to-back fashion. This idea can be likened to the strong arc abstraction used in EM-4 [103].

3.8 Conclusions

This chapter introduced three key strategies that were instrumental in the design of DaCO: (1) adding support for criticality-aware out-of-order scheduling for 1000s of nodes inside each PE, (2) customization of the communication framework into a

hierarchical topology to target the DAG sequential tail by trading off resource for runtime performance, and (3) careful RTL design to maximize resource utilization on the target Arria 10 AX115S FPGA. Overall, DaCO delivers up to 2.5× speedup over the dataflow baseline, and up to 2.8× over a baseline CPU implementation, and incurs a small 15–40% resource overhead from clustering (cluster size 2–4). In the next chapter, we look into greater detail at the design of Hoplite-Q*, a priority-aware packet-switching router that is used to create a network on chip for communication between DaCO clusters.

3.9 Publications

The body of work presented in this chapter has been published in the following peer-reviewed conference proceedings:

1. Siddhartha, Nachiket Kapre, *Out-of-order dataflow scheduling for FPGA Overlays*, Overlay Architectures for FPGAs (OLAF) Workshop (co-located with FPGA 2017), February 2017 *(Position Paper)*

2. Siddhartha, Nachiket Kapre, *DaCO: A High-Performance Token Dataflow Coprocessor Overlay for FPGAs*, Field-Programmable Technologies (FPT) 2018, December 2018 *(Full Paper)*

Chapter 4

Network on Chip Design

4.1 Introduction

The shift from designing bus-based shared communication networks to network on chip (NoC) communication overlays has been motivated by the continued development in parallel manycore heterogeneous architectures and the increasing challenge of designing sophisticated on-chip interconnect. Interconnect, in particular, has been a stumbling block in realising fast and power-efficient communication medium on System On Chip (SoC) architectures [8], as increasing wire density results in parasitics that restrict performance and/or increase power dissipation [107]. For example, while crossbars improve bandwidth and reliability over shared busses, the high tradeoff in the cost of wires ($O(N^2)$ scaling) makes crossbars impractical for rich SoC designs. Modern VLSI systems suffer from increasing wire delay (>> gate delay) and high power consumption from wires [28, 12]. Eventually, as the number of interconnected IP cores on an SoC increases, a network on chip becomes an attractive engineering option for designing the on-chip communication framework [10]. For the many-core DaCO architecture discussed in Chapter 3, a NoC is an obvious fit for the target implementation.

NoCs come in many flavors [4]. Design choices such as routing strategy (packet-switching vs circuit-switching), topology (*e.g.* mesh, torus, ring, butterfly fat trees, etc), routing function (Wormhole, Hot-Potato/Deflection, West-Side First), number of channels (including virtual channels), buffering strategy, etc are some of the important features that shape the NoC behaviour. In this work, we focus on packet-switching NoC routers, primarily because of their (1) ease of design,

and (2) suitability to dataflow/bulk-synchronous parallel (BSP) communication patterns. While circuit-switching networks can deliver close to peak bandwidth when a link is established, the cost of the crossbar and switching circuitry grows at $O(N^2)$ [51]. More importantly, for dataflow-style traffic, the irregularity in the communication pattern would incur significant overheads due to setup and idle time. As such, since we do not know the communication pattern upfront, we opt for a packet-switching network for the communication overlay design. Other secondary design choices enumerated above are also explained later in this chapter in section 4.2.1.

High-performance custom routers are typically designed with an ASIC (application-specific integrated circuit) or FPGA target in mind. While ASIC routers are capable of delivering the best achievable throughput, the long design cycle and heavy investment overheads often limit their applicability to many real-world applications. Furthermore, gate delay is significantly lower than wiring delay on ASICs [28], which motivates the design of rich arbitration policies that minimize wire lengths. FPGAs, on the other hand, are wire-rich architectures that incur relatively smaller development costs, and their reprogrammability also offers flexibility suitable for fast-changing workload environments. However, naïvely porting an ASIC router design to an FPGA would create a bloated and slow communication framework, which is not appropriate for the FPGA-efficient dataflow overlay goal in mind.

In the datacenter domain, we forsee large NoC-enabled FPGA chips hosting multiple applications with varying communication constraints simultaneously on the same chip. These FPGA-based NoCs can provide high-bandwidth and low-latency connectivity between hardware components (IP blocks, system-level interfaces) on the FPGA in a resource-shared, cost-effective manner. FPGAs also support a rich set of external connectivity in the form of PCIe (peripheral component interconnect express), DRAM (dynamic random-access memory), and Ethernet interfaces, which can be supported by NoCs to allow external data streams to reach different parts of the FPGA at full bandwidth. NoCs are also suitable for the accelerator domain as a communication fabric for routing traffic between spatial FPGA datapaths.

In this chapter, we detail the design of a high-performance, resource-efficient, FPGA-based NoC architecture that is capable of exploiting criticality in dataflow

graph paths to deliver improved throughput in our DaCO architecture. Priority-aware routing is also crucial to delivering multi-tier Quality of Service (QoS) across a range of concurrently running workloads. For example, in the datacenter domain, delivering different levels of service based on a pricing policy could be desirable. This can be achieved with priority-aware NoCs in a shared environment that are capable of discriminating and prioritizing traffic based on the customer grade. To address these needs, this chapter introduces **Hoplite-Q***, a high-performance, lightweight, priority-aware deflection router for FPGAs.

4.2 Background

4.2.1 Network on chip basics

Network on chip, as the name suggests, creates a communication network between different processing elements (PEs) on a chip. As the number of PEs increases, it becomes intractable to support point-to-point connections between each soft-processor (*i.e.* crossbars), whereas a shared bus can bottleneck performance due to the shared bandwidth [11, 7]. NoCs allow hardware designers to create scalable communication frameworks that can be tailored to deliver the required routing bandwidth for different workload types or application domains. NoCs are built by creating a network of interconnected routers, and hence, the design choices made on the router architecture and inter-router connectivity can have significant impact on the NoC performance. We list three key design decisions when designing a NoC:

- **Flow control** : *e.g.* Packet-switching, circuit-switching, wormhole-switching, etc – For long, persistent, and predictable communication patterns, a circuit-switching NoC (*e.g.* [51]) is suitable, which operates by reserving routes between communicating PEs at runtime. A packet-switching NoC (*e.g.* [69]) is better suited for short, unpredictable communication patterns between processors, a feature of the benchmarks evaluated in this thesis.
- **Topology** : *e.g.* Mesh, torus, ring, butterfly fat trees (BFTs), etc. The topologies have a significant impact on the wiring requirements of the NoC.

For applications which require lots of irregular inter-PE communication, a more dense topology with high bisection bandwidth (*e.g.* mesh or BFT) is more suitable, while a more sparse connectivity in a ring/torus is suitable for applications with light and regular communication patterns.

- **Routing function** : The routing function impacts the size of the arbiter inside each NoC router, which can have a significant impact on the size of the router, and the overall performance of the NoC. Routing functions can generally be classified into deterministic and adaptive methods [25]. In deterministic algorithms, packet routes are determined based purely on the destination address, whereas, in adaptive algorithms, packet routes can also be influenced by network health (*e.g.* congestion). Fully-featured adaptive algorithms require more complex hardware implementations, which result in bloated designs on FPGAs. In this chapter, we focus on instead on deterministic routing functions to realize lightweight and scalable NoC architectures for FPGAs. Some popular deterministic arbitration strategies include dimension-ordered routing (also known as XY-routing [11]) and hot-potato/deflection routing. The richness of the deterministic routing function also depends on the available resources such as virtual channels or on-chip buffers that further contribute to the size of the NoC router.

A more comprehensive survey of NoC architectures can be found in [4, 11, 25].

4.2.2 Existing NoC Routers for FPGAs

Routers can be designed with or without local buffers. Buffers inside each router typically boost achievable routing throughput as they open up design optimization strategies not available to bufferless designs. However, on FPGAs, buffer-rich routers consume precious on-chip memory resources (registers or dedicated block RAMs) and the routing function complexity can also limit resource utilization and achieveable clock frequency. For example, FPGA routers, such as the CMU Connect [94] and UPenn Split-Merge [54, 66], comprise of long buffers and rich crossbars that consume thousands of LUTs and operate at $\leq$200MHz. On the other hand, ASIC routers like Aérgia [26] and MinBD [38] can deliver exotic features like virtual channels (VCs), input buffers, and prioritization frameworks at $\geq$1GHz clock speeds. It is impractical to design such routers for FPGAs, and hence,

this chapter focuses on designing high-performance but FPGA-friendly bufferless routers. A case for bufferless routers for NoCs was made with BLESS [85], and since then, new bufferless router designs like CHIPPER [37] (ASIC) and Hoplite [65] (FPGA) have been competent additions to the router ecosystem. A summary of deflection routers in NoCs can be found in [79]. Hoplite [65] is a state-of-the-art bufferless deflection router that consumes only ≈60 LUTs and runs at 2.9ns (see Section 4.2.3 below), and serves as the baseline for the work detailed in this chapter.

In [3], the authors compare hard (baked into silicon, fixed architecture) and soft (built with FPGA primitives, flexible design) NoCs. They observed that hard NoCs, on average, are 20–23× smaller and 5–6× faster than soft NoCs. With the latest Xilinx Everest [136] FPGA announcement in late 2018, which comes with a hard integrated NoC, there is now further interest in this topic, and offers overlay designers new avenues to exploit advantages of NoC-based communication on FPGAs.

4.2.3 Hoplite NoC

The Hoplite [65] deflection router is a cheap, low-cost, FPGA-friendly NoC router that uses deflection routing coupled with a unidirectional 2D torus topology to deliver an FPGA-optimized NoC solution. In the unidirectional 2D torus topology,

TABLE 4.1: Existing NoC Routers for FPGAs

Router	Tech.	Routing Strategy	Clock
Hoplite	FPGA	Packet-Switching	2.9ns
PNoC	FPGA	Circuit-Switching	6-8ns (Virtex II)
CHIPPER	ASIC	Packet-Switching	1.9ns
CMU CONNECT (2 VCs)	FPGA	Packet-Switching	9.6ns
Aérgia	ASIC	Packet-Switching	$< 1ns$
UPenn Split-Merge	FPGA	Packet-Switching	4.5ns
MinBD	ASIC	Packet-Switching	<1ns

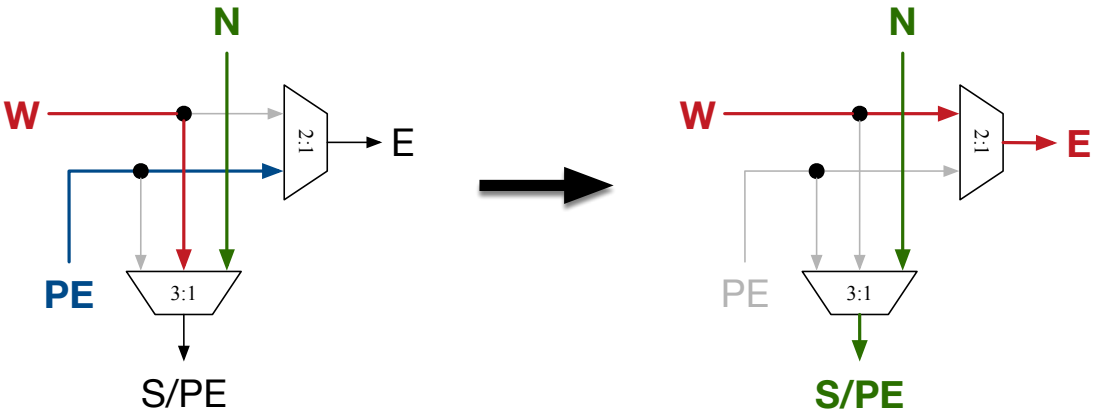


FIGURE 4.1: DOR deflection routing illustrated in Hoplite. W and N packets are both contesting for S output port, so W gets deflected to E and has to traverse the entire west-to-east plane, while the packet at PE is denied injection in this cycle.

packets can only traverse in a single direction in each dimension (*i.e.* West-to-East, and North-to-South), and they have to wrap around to reach destination processors that are located in the opposite direction (see Figure 4.2 for an example 4x4 system). The torus topology is especially suited to FPGAs, as the switching at the outputs can be packed efficiently into the available six-input lookup tables (LUTs) on FPGAs, and the unidirectional topology saves wiring costs. Hoplite routes packets using the Dimension Ordered Routing (DOR) algorithm, where packets are routed in the West-to-East direction (X-plane) first, before being routed along the North-to-South direction (Y-plane). The deflection router is also a buffer-less switch design – packets are deflected, instead of buffered, whenever there is contention for a routing path inside a switch in any given cycle. This has a few implications: the design is lightweight in LUT cost due to the simple control flow logic required, but packets can suffer from high communication latencies due to deflections. Despite this, deflection routers can be competitive for latency-tolerant, throughput-sensitive real workloads, as their resource-light design is desirable for scalability.

In Figure 4.1, we show a high-level block diagram of the Hoplite router [65]. Packets can enter the router from three input ports – the Processing Element (PE), West (W), and North (N) – and can exit from two output ports – East (E) and South (S). The S port is shared to deliver packets to the PE and a separate valid signal distinguishes this scenario, *i.e.* a packet from the N input port can only continue down S, or exit into the PE if it has reached its destination. In the event of contention for an output port, the packet at input PE is given the

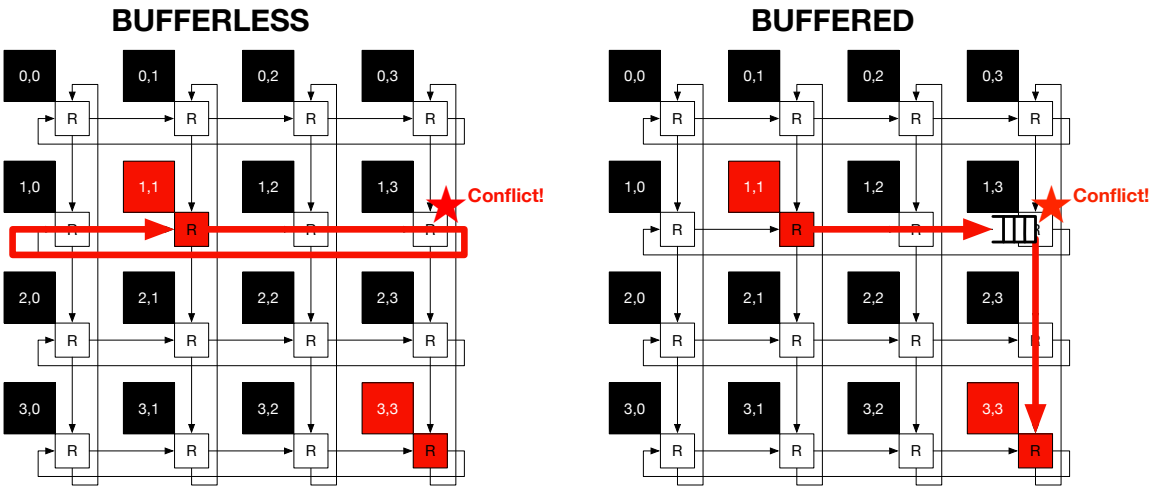


FIGURE 4.2: Bufferless vs Buffered Routers (R), where PE at (1,1) is sending a packet to (3,3). In the instance there is a conflict, the packet is deflected to the next available port (East in this case) in the bufferless network. In contrast, in a buffered network, the same packet can be stored locally in that cycle, and then safely injected in another cycle towards its destination.

lowest priority, *i.e.* no packet is accepted into the network from the PE (*e.g.* soft-processors, system interface) in that cycle. If there is contention for the S output port from valid packets at N and W input ports, then the N packet is always given routing priority, while the W packet is deflected to the E output port. This arbitration strategy trades off packet latency for lightweight, bufferless switch design. Figure 4.1 shows an illustration of this deflection DOR-based routing employed in Hoplite.

4.2.4 Hoplite Limitations

The lightweight Hoplite [65] NoC provides an FPGA-optimized solution for composing large chip-wide NoCs scaling to thousands of processing elements on the chip. Hoplite is known to outperform competing FPGA NoCs such as CMU Connect [95] and Penn Split-Merge [55] designs by 1.5× in throughput (packets/cycle) while being 20–25× smaller in size, and operating at a 3–5× faster clock frequency.

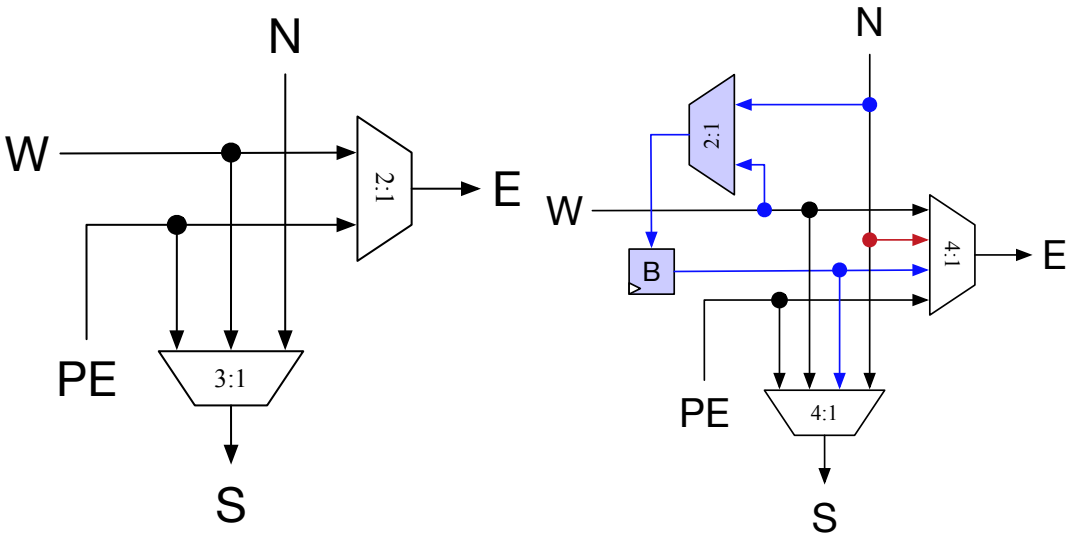
However, this low cost comes at a price: high deflection routing penalty for a single application, and inability to discriminate between traffic from multiple distinct applications. Figure 4.2 shows an example where there is a high conflict rate at (1,3), and hence packets traveling from (1,1) to (3,3) can get perpetually deflected in the bufferless Hoplite network. In contrast, with buffers, it is possible to design

arbitration strategies that guarantee progression in the communication network. This phenomenon is known as livelock, which Hoplite networks are susceptible to and hence care has to be taken when partitioning workloads on the overlay. HopliteRT [127] addresses this issue elegantly by modifying a single arbitration rule, but can still fail under special scenarios and does not offer any solutions for other above-mentioned limitations.

In this work, our motivation is to address these issues by introducing best-effort support for quality of service (QoS) outcomes in the Hoplite NoC router. We show a high-level view of the structural adaptations to the router in Figure 4.3, which we call Hoplite-Q*. To enhance routing choice in the Hoplite routers, we add a *single* packet buffer to the Hoplite router. The buffer and associated multiplexing circuitry helps to improve the overall NoC throughput and mitigate long packet latency delays significantly. We show how to exploit this choice by including priority-driven routing inside the buffered deflection router. We augment the dimension-ordered routing policy with rules that enable priority-aware selection of which packets to deflect. This requires tagging each packet with extra priority bits to help determine the routing decision. Finally, together with a sensible arbitration strategy, Hoplite-Q* also prevents livelock by ensuring progression in the communication network at all times. This strategy revolves around updating packet priority-tags on every deflection, which is explained in greater architectural detail in Section 4.3.3 later in the chapter.

4.2.5 Quality of Service (QoS) in existing routers

The majority of existing bufferless routers [65, 37, 85, 76] dedicate limited attention to delivering varying QoS for mixed-priority multi-application workloads. In fact, any priority-aware arbitration rules, such as the *Golden packet rule* [37], *Silver packet rule* [38] or *Oldest-First priority scheme* [76], are designed to reduce the number of total deflections and provide guarantees against livelock for the entire workload rather than individual applications. MinBD [38] uses a side-buffer to reduce resource utilization and power costs associated with buffering, which is similar to our buffering strategy developed in this work. Unlike MinBD, however, we use a 1-flit (1-packet) deep side-buffer, as opposed to their 4-flit deep side-buffers. This strategy is sufficient for our work due to the unidirectional 2D torus topology in use, which generates fewer deflections than the bidirectional 2D mesh topology



(A) Hoplite Router

(B) Hoplite-Q* Router

FIGURE 4.3: Hoplite-Q* switch organization with enhancements. (1) Addition of priority-bits accompanying each packet, (2) Addition of buffer B to store deflected W and N packets. (3) Enhanced arbiter (not shown) for selecting between W, N, PE, and B inputs, and (4) Adders to update priority-bits of deflected packets (not shown).

in the MinBD NoC. HopliteRT [128], another flavour of the Hoplite router, delivers QoS latency bounds with minimal hardware modifications, but is also incapable of distinguishing QoS demands across different priority bins while also suffering from limited choice in the routing. Buffer-rich ASIC routers (*e.g.* [26]) can employ QoS techniques (*e.g.* live slack estimation [26], Globally-Synchronized Frames [74], Stall Time Criticality [27]) which are exorbitantly expensive in logic to use directly on an FPGA fabric, as discussed earlier. Furthermore, these techniques are primarily designed to ensure routing fairness in the network and prevent catastrophic events such as livelock. Our work on Hoplite-Q* builds on the low-cost Hoplite router instead and delivers both routing guarantee (*i.e.* no livelock) and programmable QoS features useful for mixed-criticality and/or multi-tenant ecosystems.

4.2.6 Contributions

The contributions detailed in this chapter are as follows:

- The design of a refined, parameterized Hoplite NoC router with a single buffer (Hoplite-B), and a fully-featured priority-aware design (Hoplite-Q*). Hoplite-Q* supports both static and/or dynamic packet priority management features

that allow the routers to route packets based on a priority-tag embedded inside each packet's header.
- Evaluation of router designs with various real-world dataflow workloads with both the baseline dataflow processor and our proposed PE-DaCO (see Chapter 3) overlay architecture.
- Evaluation of router designs on various statistical and real-world benchmarks to quantify improvements in throughput, latency and cost.
- Evaluation of mixed-priority multi-application workloads and measurement of associated QoS outcomes for statistical and real-world benchmarks.

4.3 Priority-Aware Hoplite

In state-of-the-art deflection routed NoCs, all runtime routing decisions are based on simple deterministic routing algorithms like DOR that depend solely on the destination address. However, real-world systems need to route traffic from mixed-priority, multi-application workloads where not all packets are created equal. We can add priority awareness to a NoC by adding additional priority bits in each packet (we call it the *priority-tag*) and using those bits to determine packet route. When used in buffered NoCs like those in [79, 26], the lower priority packets can simply wait in the buffer a little longer. If we apply this naïvely to the Hoplite NoC, there is limited impact on QoS because of fewer routing choices. Thus, we must rethink how we can make priority-awareness more amenable to the Hoplite NoC design.

In this section, we look at three main design adaptations necessary to Hoplite that balance FPGA cost and resulting NoC performance:

1. The design of a priority-aware routing function,
2. Introduction and sizing of the buffer in the router to boost routing choice, and any associated buffering policy design, and
3. Priority assignment method (statically at compile time, and/or dynamically updating packet priorities at runtime).

We perform a careful cost-benefit analysis and quantify the effect of each proposed change.

4.3.1 Priority-Aware Routing Function

The DOR routing function in the original Hoplite statically prioritizes N packets over W packets, where PE packets have the least priority. To support priority-awareness, at the very least, we must allow W packets to be able to deflect N packets (*e.g.* see Figure 4.2, where a high-priority packet on the west-to-east plane must win the conflict at (1,3) to progress to its destination). This necessitates the addition of the $N \rightarrow E$ turn in the switching crossbar (shown as the red turn in Figure 4.3b). For this configuration, we can design a priority-aware routing function on two NoC input ports – N and W – such that the less critical packet is always deflected to the E output port. Under this model, packets at PE are still given the least importance, regardless of their priority just like in the original Hoplite design.

Unfortunately, simply adding a N→E turn does not result in a good priority-aware router since the packet at PE would still always be denied injection. This is unacceptable as our goal is to design a router that prioritizes packets based solely on their priority, and not their port of origin. To address this, we add a buffer, B, in the router to hold N or W contending packets (shown in blue in Figure 4.3b), and allow the PE packets an opportunity to enter the NoC if they have higher priority. We then adjust the priority function to consider four inputs at N, W, PE and B ports. We also support *buffer redirection*, where a low-priority packet in a buffer can be ejected to allow a higher-priority packet to be buffered instead. The addition of such a buffer is counter-intuitive as high buffering costs were the key motivating factor behind the design of Hoplite. Our experiments show that a *single* buffer location delivers most of the benefits we desire while only increasing design cost modestly. The local buffer also mitigates the number of deflections at runtime even without priority, which we quantify later in Section 4.5.2.

When designing the priority-aware router, we were faced by two key research questions: (1) how do we update the packet priority-tag, and (2) how do we determine which packets get to use the buffer? We answer these questions in the next three sections.

FIGURE 4.4: Example packet format in Hoplite-Q network. Packets now have an additional P_s-bit static priority-tag (in blue)

4.3.2 Static Priority

For FPGA applications where communication pattern is known at upfront, we can identify priority classes statically at compile time. A good example of this can be found in dataflow graphs, where nodes and edges along the critical path(s) can be assigned higher priority classes at compile time to improve runtime performance. These are static priority bits that do not change value as they traverse the NoC. We call this NoC router design *Hoplite-Q*, which only uses the statically-assigned priority-tag to route NoC traffic.

- For applications with dataflow-type dependencies, the compute structure (dependency graph) can be extracted at compile time. We can then run traditional slack analysis [83] algorithms on the dataflow graphs to determine the priority of each edge in the graph. Such analysis reveals the packets along the critical paths in the application that must be prioritized in the network for faster completion.
- For applications with no dataflow dependencies, such as Bulk-Synchronous Parallel (BSP) workloads, there are no dependency chains or critical paths in the design and a Manhattan-distance metric can be used to estimate priority. A packet that must traverse a longer source-to-sink route is likely to suffer from more deflections at runtime.
- In multi-application scenarios, we can assign priority to packets based on the quality of service desired by each application. In the extreme case, each application is assigned its own unique priority class. We focus on these mixed-priority multi-application scenarios for evaluation of Hoplite-Q.

Packets in the Hoplite-Q network have a P_s-bit priority-tag (see Figure 4.4), the value of which is computed at compile time. This value can be computed as a function of two main factors: (1) the criticality of the packet within the application (intra-application priority, *e.g.* critical path in dataflow graph), and (2) the priority of the application in a multi-application shared environment (inter-application priority), set by the overlay administrator.

The resource impact of this design is the addition of P_s extra bits of wiring on every NoC link. Consequently, the DOR arbitration logic is also modified to implement the priority-aware routing in the NoC. We empirically explore the choice of P_s to decide what is the best configuration for realistic FPGA designs. More bits will deliver greater discrimination in application behaviour on the NoC, but will require more wires and logic resources. Figure 4.5 shows the effect of increasing P_s on the logic utilization cost of the router.

While this is a start, static priority alone is insufficient for several reasons:

- Static priority assignment may also lead to livelocks where packets with lower priority are always deflected without eventual delivery.
- Packets may get trapped in the buffer (B) for long durations, thereby limiting the usefulness of the buffer.
- Static priority tagging does not account for NoC congestion effects and other dynamic events in the system, which may limit progression of packets in the network.

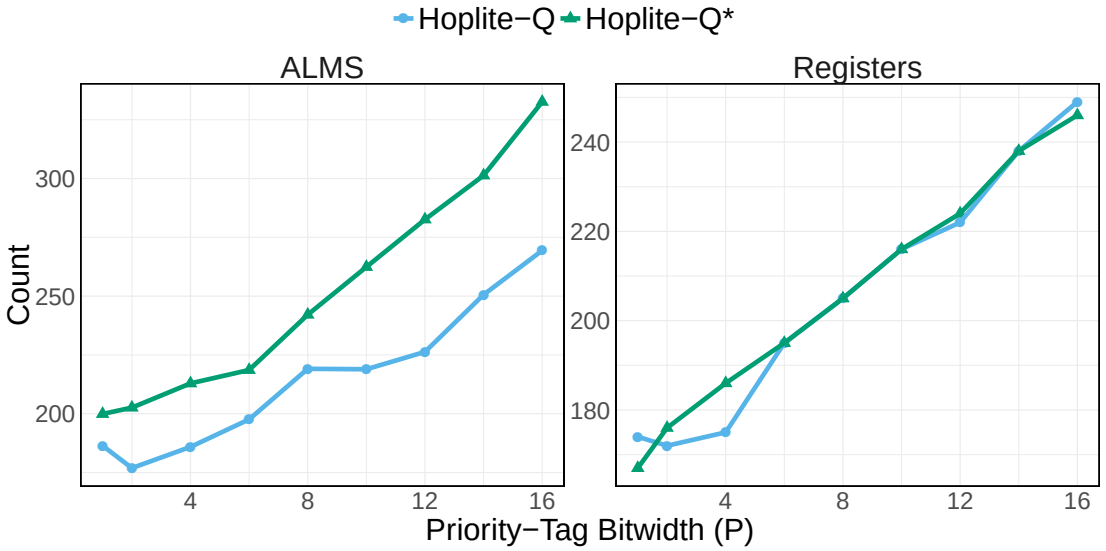


FIGURE 4.5: ALM and register cost of Hoplite-Q and Hoplite-Q* as bitwidth of priority-tag, P, is increased.

4.3.3 Dynamic Priority

As shown in the previous section, there are undesirable side-effects with priority-aware routing using statically assigned priority-tags alone. This motivates us

to consider a dynamic adaptation to the priority-aware routing function, which improve the NoC in the following ways:

- We can increment packet priority on each *deflection* to account for the age of the packet in the NoC. Thus older packets will get higher priority and facilitate their traversal through to NoC to eventual delivery at their destination. These dynamic updates are important for preventing livelock as well, since static priority assignments could repeatedly deny service to the same low-priority packet.
- Expedited delivery of older packets also allows faster release of valuable NoC resources back to the active applications on the overlay.
- For multi-application workloads, dynamic priority updates can occasionally allow lower priority applications to acquire higher priority to accelerate delivery. For best-effort QoS deliveries, this results in better utilization of NoC resources and avoids denial-of-service (DoS) to lower priority applications during runtime.
- Finally, dynamic priority updates are also applied to the packets waiting in the buffer. This allows them to acquire higher priority and eventually vacate the buffer slot.

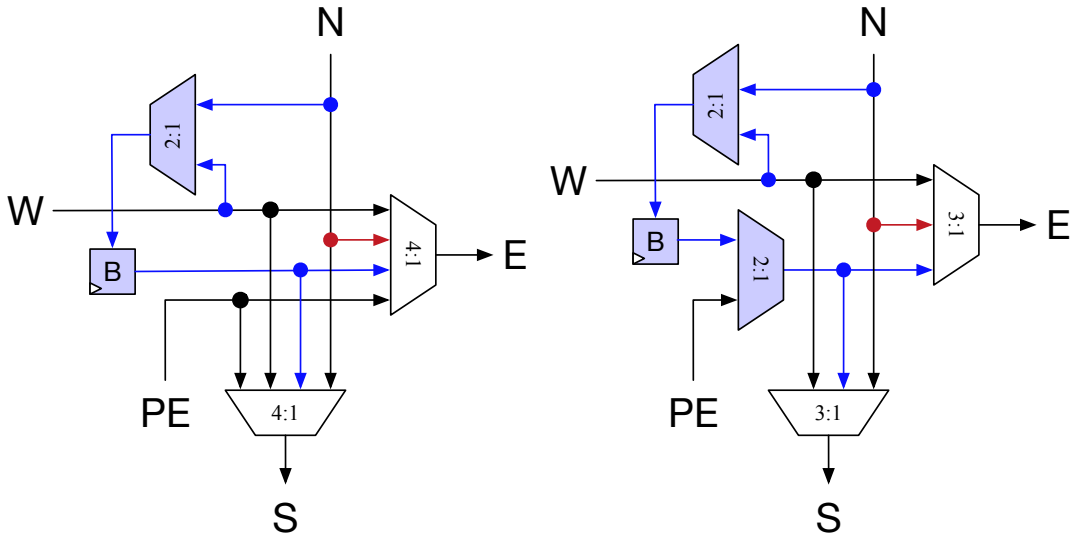
When dynamic priority update support is added to Hoplite-Q, we denote the new router design as *Hoplite-Q** in this thesis.

FIGURE 4.6: Example packet format in Hoplite-Q* network. Packets now have an additional (P_s+P_d)-bit static and dynamic priority-tags respectively(in blue)

The resource impact of this design is the addition of a P_d-bit adder in each router for each outgoing network port, and extra P_d bits (see Figure 4.6) of wiring in each NoC link. Figure 4.5 shows the logic utilization cost of Hoplite-Q* as the total size of the priority-tag is increased. As expected, compared to Hoplite-Q, Hoplite-Q* requires more logic resources for implementing the adders and richer arbitration logic.

4.3.4 Buffering

As discussed earlier, we need to add a buffer to Hoplite to increase the number of routing choices in each switch hop. This increased choice comes at the cost of ALMs and FFs to implement the buffer, and a 2:1 multiplexer for writing



(A) Hoplite-B (B) Hoplite-B* (Variant)

FIGURE 4.7: Hoplite-B switch design and a variant Hoplite-B* design. Hoplite-B* uses 3:1 multiplexers at the output E and S ports, while utilizing an extra 2:1 multiplexer to multiplex between B and PE input ports.

to the buffer. We empirically observe a depth of 1 to offer the best balance between cost increase and performance improvements, which we discuss in greater detail in Section 4.5.1 below. Packets will enter the buffer in the event of a NoC conflict where two packets desire the same output port. Packets will leave the buffer when their priority is higher than other packets at the router. The buffered packet can also get forcibly ejected in special scenarios, such that the buffer is always utilized to service the highest-priority packets at every cycle. With dynamic priority schemes, we can ensure that the waiting packets will eventually acquire sufficiently high priority to make progress in the NoC.

Even in the absence of priority, the presence of a buffer can mitigate the effect of deflection penalties. If the injection rates in the NoC are moderate, the occasional buffering event can avoid the long deflection round-trips in the ring and boost performance. To separate the effect of buffering from priority, we consider a *single* buffer router (Hoplite-B) design, and compare its performance and area tradeoffs against other priority-aware Hoplite-Q(*) designs.

4.3.5 Hoplite-B variants

Figure 4.7a shows how a buffer can be added to the original Hoplite router design to create a new design which we refer to as Hoplite-B. The arbiter (not shown in figure) in Hoplite-B now takes in up to four input packets (N, W, B, and PE) and routes packets to three output destinations (S, E, and B). Since packets can come in from four different input ports, the output multiplexers at both S and E output ports have to be upgraded from 3:1 to 4:1. Note that the $N \rightarrow E$ route (colored red) is not used by the Hoplite-B arbiter, and can be removed if priority-aware routing is not required. However, this path is crucial for enabling the desired priority-aware routing features as explained in Section 4.3.1 above.

There is, however, another way to design Hoplite-B that offers an area-performance tradeoff. Figure 4.7 highlights the architectural difference between the utilized design (Figure 4.7a) and the variant (Figure 4.7b), which we refer to as Hoplite-B* in this section. In Hoplite-B*, an additional 2:1 multiplexer switches between B and PE input packet ports, such that the buffered packet always has priority over the input packet at the PE port (*i.e.* the select line of the new 2:1 multiplexer is a `buffer_valid` signal for the packet in B). This reduces the size of the multiplexers required at the output of the router from 4:1 back to 3:1. When synthesized, the Hoplite-B* design delivers a small resource utilization advantage (≈4 ALMs/router) over the Hoplite-B design. If QoS is not a concern and the 4 ALMS resource overhead associated with Hoplite-B is unacceptable, then the Hoplite-B* design should be utilized instead.

However, when designing Hoplite-Q, there is a subtle but key behavioural difference that limits the Hoplite-B* router's applicability to serve as a building block. Due to the implicit $B > PE$ priority rule enforced by the new 2:1 multiplexer in Hoplite-B*, a buffered packet will starve injection from the connected PE. This is further exacerbated in mixed-priority multi-application scenarios where a low-priority packet stuck in the buffer can starve injection of high-priority packets from the PE. Even a smarter select line for the new 2:1 multiplexer, which allows a high-priority packet from the PE to *bypass* the low-priority buffered packet, does not help. In practice, this "bypass" design quickly reduces to the baseline Hoplite design at high injection rates, as low-priority packets get stuck in the buffers indefinitely, thereby eliminating any throughput advantages offered by local buffers. From our experiments, we observed that a low-priority packet that is stuck in a

high-priority application region occupied the buffer for a substantial number of cycles, thereby limiting the ability to provide desirable QoS to in-flight packets in the network. This negative effect was particularly pronounced when the offered injection rate into the network exceeded the rate that the NoC could sustain. In conclusion, a priority-aware router must prioritize delivering QoS to in-flight packets to allow progression in the network to achieve the desirable routing behaviour. From our experiments with Hoplite-B*, we concluded that any minor resource utilization savings are not worth the QoS routing penalty, and hence, we build our QoS-driven routing function on top of Hoplite-B.

4.3.6 Summary of Hoplite-Q* Adaptations

We summarize the structural changes to the Hoplite router shown in Figure 4.3:

- We need a wider east multiplexer to support $N \rightarrow E$ and $B \rightarrow E$ routes. Thus, we now need a 4:1 multiplexers instead of the 2:1 multiplexer in the original design.
- We also need a wider south multiplexer to support $B \rightarrow S$ route. This increases multiplexer width from 3:1 in the original configuration to 4:1 in Hoplite-Q*.
- We introduce an extra buffer B and an associated 2:1 multiplexer to determine which of the two incoming packets from N or W gets written to the buffer. The PE logic does not need to be modified and can continue to use the "ready/valid handshake" network interface in the original Hoplite NoC.
- We design a priority-aware routing function that selects between packets from N, W, B, and PE ports. The multiplexers in the switching fabric must be widened by P bits as the NoC ports now carry P bits of extra priority information. For dynamic updates to the priority-tag, we need a P_d-bit adder to revise the priority values based on deflection events.

4.4 Methodology

In this section, we provide details on our implementation and experimental design methodology. We also give a brief background on the benchmarks used to characterize our new router designs.

4.4.1 RTL Implementation and Simulation

We describe RTL for all router designs and compile the designs with Altera Quartus Prime Standard Edition 16.0 targeting the Arria 10 AX115S device to generate post-fitting FPGA implementation metrics. We summarize the ALM logic utilization results in Table 4.2. It is clear that the design adaptations to Hoplite cost extra resources.

TABLE 4.2: Routers Resource Utilization (ALMs), 8b priority tag where applicable (56b–64b packet length, with 32b payload)

Switch	Crossbar	%	Arbiter	%	Total
Hoplite	33	59	4	7	56
Hoplite-2×	72	60	10	8	121
Hoplite-B	40	52	8	10	77
Hoplite-2B	40	31	9	7	127
Hoplite-Q	40	22	108	61	178
Hoplite-Q*	40	19	127	59	215

Table 4.2 also shows logic utilization costs of two other router designs (Hoplite-2B and Hoplite-2×), which are described in detail in Section 4.5.1 below. It is important to observe that the Hoplite-2× design, which replicates the NoC, doubles both the ALM and wiring cost. Hoplite-2B adds resources because of a larger buffer but preserves wiring cost.

Hoplite-Q and Hoplite-Q* require a more complex arbitration function but are within 3–3.5× the ALM cost of baseline Hoplite while keeping wiring costs similar. As we will see later in Section 4.5, the increase in resource cost gives us the priority-awareness properties we desire in our system.

We run cycle-accurate simulations of the RTL using Verilator [112], which generates fast C++ code from synthesizable RTL. Our C++ testbench can route traffic from various synthetic patterns, as well as communication traces from real-world benchmarks.

We also run synthetic workload experiments, where all PEs are configured to inject 2048 packets under various traffic patterns (see Table 4.3) and at varying injection

rates from 1% (1 packet injection attempt every 100 cycles) to 100% (1 packet injection attempt every cycle). We evaluate the performance of Hoplite-Q(*) on multi-application traces which simulate multiple instances of the application operating in different regions of the NoC. We explore various NoC system sizes: 1×1 (single PE) to 16×16 (256 PEs) configurations. Each packet carries a 32b payload, and 8b address information along with P bits of priority information. Data communication between PEs is restricted to single-flit packets. Our experiments measure in-flight NoC latency, source queueing time, total packet latency, and sustained throughput metrics of the resulting implementation. We quantify the effect of system size, priority-tag bitwidth, static/dynamic priority-aware routing, along with variations due to real-world dataset.

4.4.2 Benchmarks

Synthetic NoC Traffic Patterns

Table 4.3 summarizes the synthetic NoC traffic patterns used in this study. These are well-known synthetic traffic patterns [2] that are useful for stress-testing a target NoC architecture. We develop a custom PE that is capable of generating and injecting packets into the network that mimic each traffic pattern.

TABLE 4.3: NoC Statistical Traffic Patterns

Pattern	**Formula**	**Examples (8x8 NoC)**
Random	$dst_x = rand()\%N_m$	Any possibility
Bitrev	$dst_x = bitrev(src_x)\%N_m$	(1,3) → (4,6)
Transpose	$dst_x = src_y\%N_m$	(4,7) → (7,4)
Neighbour	$dst_x = (src_x+1)\%N_m$	(2,5) → (3,6)
[1]Complement	$dst_x = bxor(src_x,0xf)\%N_m$	(0,5) → (7,2)
Tornado	$dst_x = (src_x+\frac{N_m}{2}-1)\%N_m$	(0,3) → (3,6)
[2]Local	$dst_x = local\_rnd(src_x,\sigma)\%N_m$	*e.g.* $\sigma = 2$, (3,3) → (2,2) to (4,4)

dst_x and dst_y computed in the same fashion
N_m = Maximum number of PEs in X or Y-plane
[1]bxor = bitwise-XOR
[2]local_rnd() = returns random number localized to $\frac{\sigma}{2}$ radius of input

Bulk Synchronous Parallel (BSP) abstraction

TABLE 4.4: Sparse matrix BSP benchmarks used in this study

Benchmark	Domain	Nodes/Edges
`mcca`	Non-LTE (local thermodynamic equilibrium) problem from astrophyics	180/2.7k
`lns511`	Fluid flow modeling benchmark	511/2.8k
`bp1600`	Simplex method basis matrix from the Harwell-Boeing Collection	822/4.8k
`simucaddac`	SPICE circuit simulation benchmark for 90nm process technology	654/5.5k
`jpwh991`	Computer random simulation of a circuit physics model	991/6k
`add20`	Circuit netlist of a 20-bit adder	2.4k/17k

We extract benchmarks from the Bulk Synchronous Parallel (BSP) [42] domain, which is a well-known compute abstraction that represents computation as a graph consisting of nodes (computation) connected by edges (communication). A BSP graph is evaluated in a synchronized lock-step fashion, where computation and communication occur in distinct stages separated by a global synchronization barrier. The NoC is utilized in the communication phase of the BSP workload, where a large number of packets are injected into the network and the system waits for all deliveries before proceeding. Such BSP applications are typically iterative applications that converge to a solution, and communication optimizations greatly influence program runtime. Some popular applications include Pagerank [82, 109] and Sparse Matrix-Vector Multiply [139, 140]. In order to test these benchmarks on DaCO, we made modifications to the PE to support BSP-style injection of traffic into the NoC.

Table 4.4 summarizes the BSP benchmarks used in this study.

Directed Acyclic Graphs (DAGs)

As explained in Chapter 2, token dataflow is a computing model that evaluates explicitly on the directed acyclic graph (DAG) of an application. In Chapter 3, we showed how we can build a custom dataflow-inspired soft-processor and evaluated its performance on a set of benchmarks from the circuit simulation domain. We use the same benchmark set (see Table 3.1 in Chapter 3) in our experiments in this chapter, but instead focus on quantifying the effect of the NoC design on DaCO's performance. Each edge in the DAG is statically assigned a criticality at compile time by using the well-known slack analysis algorithms described in Chapter 3.

We use "criticality" and "priority" interchangeably in this chapter to denote the importance of a communication edge (*i.e.* packet) in the DAG and in the network. The results are detailed in Section 4.5.6, where we isolate and quantify the impact of the Hoplite-Q* NoC when used with either the baseline in-order PE or the novel out-of-order PE-DaCO.

4.5 Results

We quantify the effect of priority-aware routing on BSP and dataflow in this section. Each sub-section here is designed to answer a specific research question, the answer to which is summarized at the end of the sub-section.

4.5.1 Baseline Calibration Tests

Question: Instead of adding a buffer to Hoplite, can we use the FPGA resources in another way to achieve better results, both in terms of resource utilization and speed? If not, what buffer depth should be used?

The extra resources used by Hoplite-Q* could be theoretically reused to create deeper buffers or replicated communication channels. These are simple and obvious alternatives, which have the potential to improve NoC performance significantly and also offer insights into the design for priority-aware Hoplite. We compare the performance and resource utilization trends of baseline Hoplite against 1-deep (Hoplite-B) and 2-deep (Hoplite-2B) buffered-Hoplite, as well as a two-replicated-channel (Hoplite-2×) solution.

In Figure 4.8, we quantify the resulting throughput and latency performance of Hoplite, Hoplite-B, Hoplite-2B and Hoplite-2× designs for `UNIFORM RANDOM` traffic at 64 PEs.

Hoplite-B provides a significant improvement over Hoplite in throughput (1.5×) and latency performance (1.3×). Hoplite-2B improves the throughput performance further but only marginally (1.6× over Hoplite, 1.1× over Hoplite-2B). Curiously, at an offered injection rate of $\geq$0.2, the average packet latency delay is

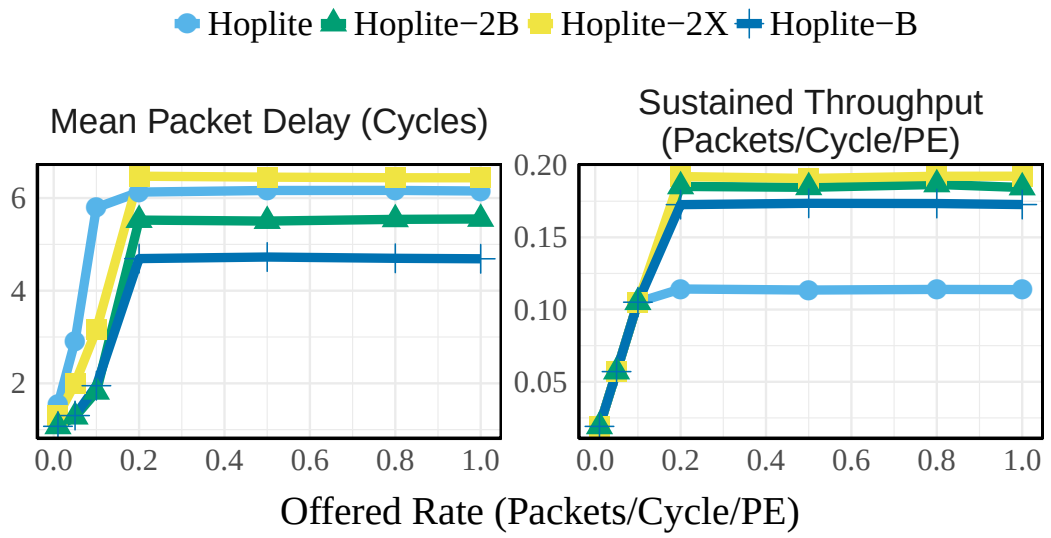


FIGURE 4.8: Average packet latency and sustained throughput vs offered throughput on 8×8 NoC with uniform random traffic

higher in Hoplite-2B than Hoplite-B, despite higher sustained throughput. This effect can be attributed to the slightly deeper buffers in Hoplite-2B and head-of-line blocking. Overall, both Hoplite-B and Hoplite-2B improve NoC communication throughput performance by up to 60% by mitigating the deflection penalty suffered by packets.

The dual-channel NoC, Hoplite-2×, achieves the highest throughput of all designs (1.7× over Hoplite), simply because packets can now be routed over two channels, instead of one, in parallel. The lack of buffers mean that packets still suffer long deflections in each channel, and hence, the average packet latency is close to that observed in single-channel Hoplite NoCs.

The NoC saturation point is also slightly higher in all three Hoplite-B, Hoplite-2B, and Hoplite-2× NoCs (0.2 packets/cycle/PE) when compared to baseline Hoplite NoC (0.15 packets/cycle/PE).

In Figure 4.9, we illustrate the tradeoffs between performance and logic resource utilization for these router designs using post-fitting metrics (ALM cost, and operating frequency). At 16×16 PEs, we observe a 70% improvement in throughput for an additional 40% resource cost when comparing Hoplite-B against baseline Hoplite NoC. Note that, for the target Arria 10 FPGA, the ALM resource utilization is still under 5% for the 16×16 NoC using Hoplite-B routers.

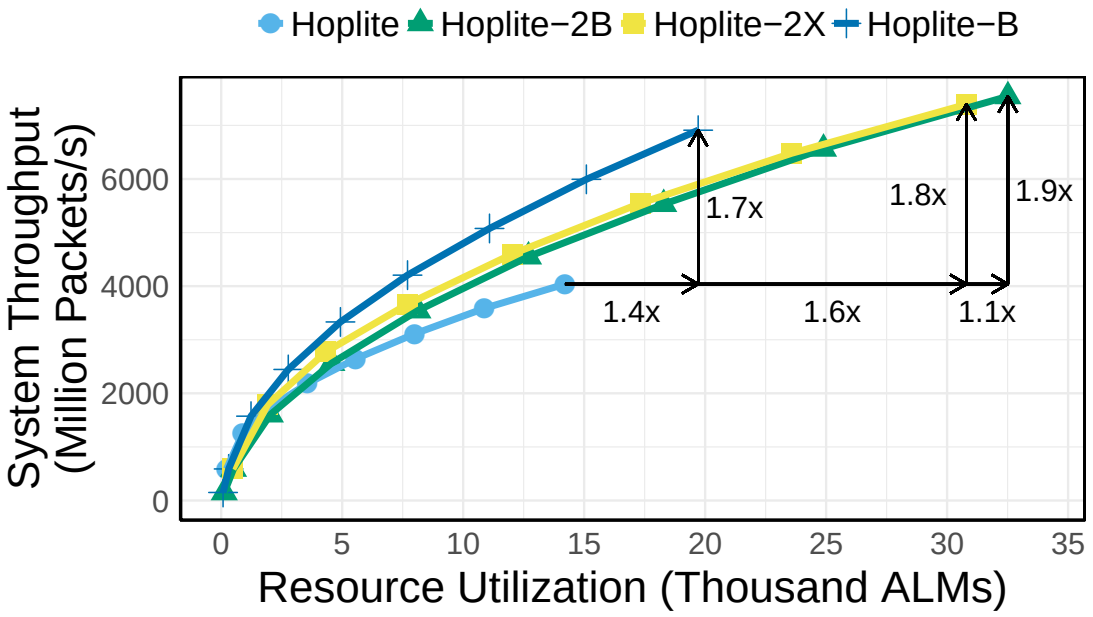


FIGURE 4.9: Sustained throughput (millions of packets per second) vs resource utilization (ALMs)

The same comparison between Hoplite-2B and Hoplite-B highlights the diminishing performance returns – only an additional 10% improvement in throughput performance for 70% increase in resources used on the FPGA. This is supported by trends observed in [68, 38], where deeper buffers do provide improvements, but the most significant improvement comes from just one stage. Unlike [68] however, our solution is cheaper and does not use any BRAM resources and shares a single buffer location for holding deflected packets instead of a per-port buffer.

Hoplite-2× offers a solution in between Hoplite-B and Hoplite-2B, but there is still an inefficient use of logic resources – we observe a 7% improvement in performance for 60% increase in resource utilization over Hoplite-B. Furthermore, increasing the number of communication channels on the NoC requires double the wiring resources that adds complexity to the overall design for limited benefit.

These baseline calibration tests suggest that a single buffer location offers a better return on investment of FPGA resources, when compared to the dual-buffer approach. In addition, minimizing the buffer size also delivers better power efficiency, as static and dynamic power consumption of on-chip buffers is expected to be smaller. Finally, when considering the Hoplite-2× NoC design, we observe a significant increase in ALM cost (and wiring cost) without a commensurate improvement in performance. Overall, the lightweight buffering in Hoplite-B strategy

delivers the most effective balance between resource usage and increased performance. Therefore, these obvious alternative NoC router designs (Hoplite-2B and Hoplite-2×) are not competitive and are eliminated from further consideration.

Answer: Our calibration tests show that adding just a single buffer to Hoplite offers the best tradeoff in terms of resource utilization and throughput performance.

4.5.2 Effect of Buffering (Hoplite-B)

Question: How does adding a single buffer to Hoplite impact the throughput performance of the NoC across different synthetically generated traffic patterns?

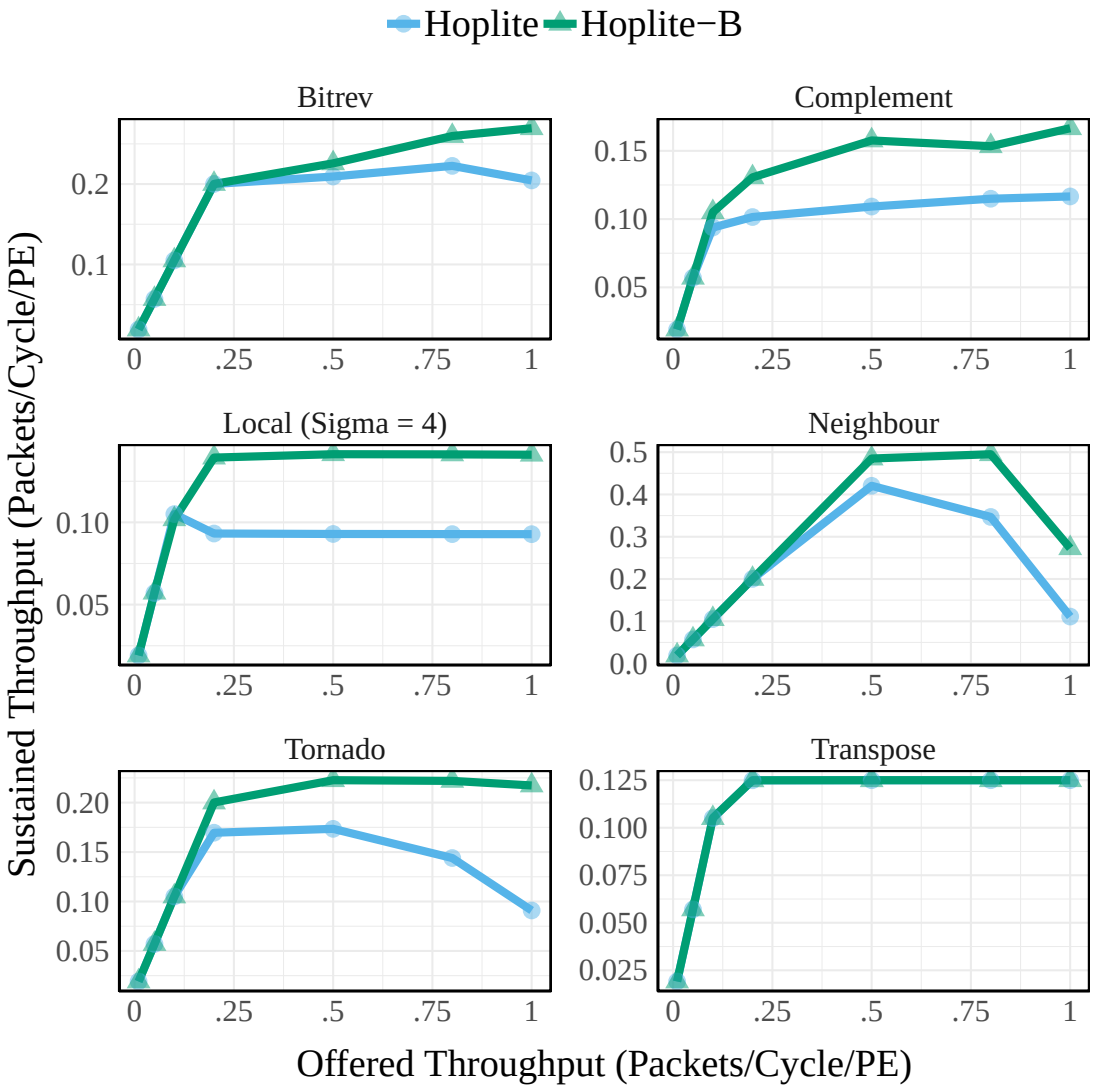


FIGURE 4.10: Sustained vs Offered Throughput for Hoplite and Hoplite-B under various traffic patterns on an 8×8 NoC.

Figure 4.10 compares performance of a Hoplite NoC to a Hoplite-B NoC under various statistical traffic patterns for an overlay system size of 8×8. The sustained throughput of all traffic patterns evaluated by the Hoplite-B router NoC improves by 1.2–2.4×, depending on the pattern type and offered throughput. Some notable observations include:

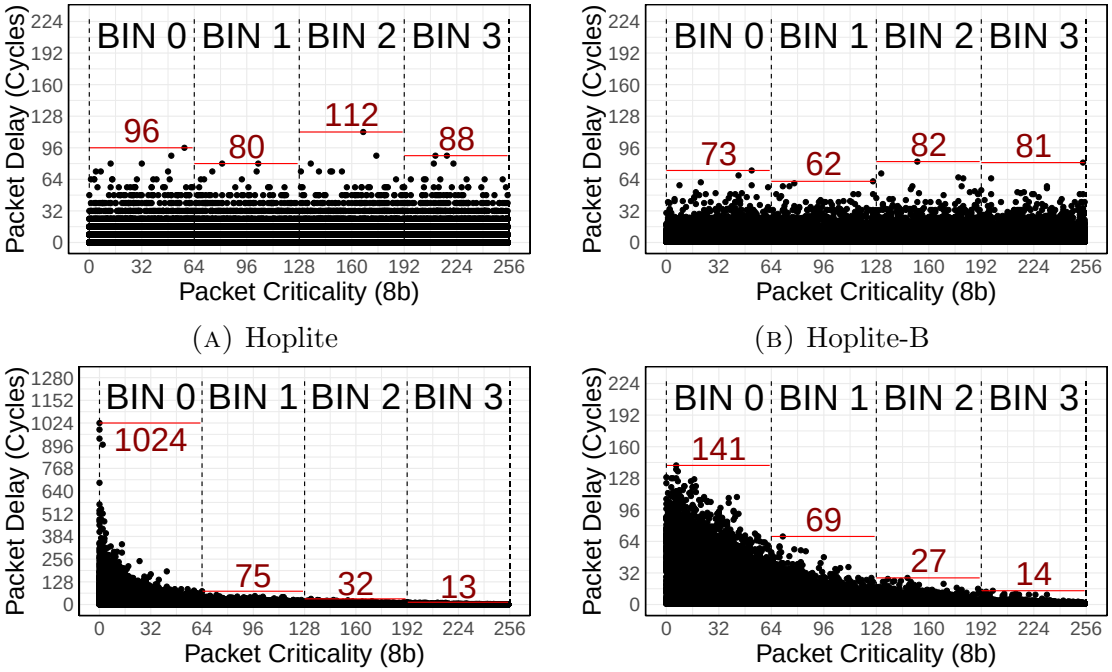
Tornado: This traffic pattern creates communication in clusters that contain PEs from each row in a 2D NoC. By adding the buffer, the Hoplite-B router is capable of absorbing these deflections, and can deliver up to ≈2.4× improvement (1.4× average) in overall sustained throughput.

Neighbour: This traffic pattern creates very few deflections, since the packets have a short source-to-destination distance to travel, and the routing direction of traffic lines up with the unidirectional 2D torus topology of our NoC. The Hoplite-B router NoC, however, is capable of dealing with the reduced deflections particularly well. On average, the sustained throughput improvement for the Neighbour traffic pattern is 1.3× across all injection rates.

Transpose: Both Hoplite and Hoplite-B router NoCs have similar throughputs with this traffic pattern. This is due to the presence of self-communicating PEs, *e.g.* (0,0) → (0,0) or (1,1) → (1,1) along the diagonal. Since both Hoplite and Hoplite-B are agnostic to packet priority, the input PE port in the router is always given the least routing priority by default, hence, packets from these PEs are suppressed for most cycles as they are always contesting for the output S port. Naturally, these diagonal PEs serve as a throughput bottleneck, and as a result, produce very similar sustained throughput trends with both Hoplite and Hoplite-B NoCs.

Bitrev, **Local**, **Complement** : We observe up to 1.3–1.5× improvements in throughput at high injection rates for these traffic patterns. The **Local** pattern is especially common in real-world applications, which motivates us to keep hold of the advantages offered by introducing a buffer in Hoplite.

Finally, since these are statistical traffic models with no inter-packet dependencies, Hoplite-Q/Hoplite-Q* deliver the same performance as Hoplite-B for these single-application synthetic benchmarks.



(A) Hoplite (B) Hoplite-B

(C) Hoplite-Q (D) Hoplite-Q*

FIGURE 4.11: Packet delay distribution on 8×8 NoC for 4-application synthetic workload across various Hoplite NoC designs at 50% injection rate.

Answer: Hoplite-B delivers up to **2.4**× *improvement over Hoplite across the tested traffic patterns, and only fails to improve throughput when there is an inherent network bottleneck resulting from the DOR deflection routing policy. On most synthetic traffic patterns, the addition of the buffer improves throughput performance by 1.2–1.5*×*.*

4.5.3 Effect of Priority (Hoplite-Q*)

Question: What is the effect on packet latency distribution under existing/proposed Hoplite NoCs when priority is introduced?

In this sub-section, we quantify the effect of priority-aware routing on a 4-application synthetic workload (`UNIFORM RANDOM` traffic, 32K packets) by measuring packet delay distributions in each priority class. Each application is assigned its own priority bin with a 2b static tag, used by Hoplite-Q NoC, and a 6b dynamic tag that is incremented per deflection by the Hoplite-Q* NoC. In Figure 4.11, we see the results of this experiment.

We track the number of *extra cycles* spent in the network by each packet due to deflections ($\text{Cycles}_{observed}$ - Cycles_{ideal}). Here, ideal cycles are computed simply as the Manhattan distance in absence of congestion. We then highlight the worst-case latency in each bin to quantify the quality-of-service outcomes observed in each bin.

Below are our observations based on these experiments:

- In Figure 4.11a, the Hoplite router produces quantized levels of delay that are in multiples of 8. This is due to the 8×8 system size, and associated 8-cycle deflection in the X-plane. The worst-case latency delay suffered by a packet is 112 cycles, which indicates 14 deflections. We observe a uniform distribution of packet delay cycles in each priority bin as baseline Hoplite is oblivious to application priority.
- In contrast, Hoplite-B results in Figure 4.11b show a smoother packet delay distribution as packets may wait in the buffer upon deflection. The worst-case latency has improved to 82 cycles, but the packet delay distribution is still uniform across all priority bins.
- The Hoplite-Q NoC delays shown in Figure 4.11c indicate a priority-sensitive delay distribution across priority bins. The higher-priority bin now sees a worst-case delay of only 13 cycles, while the least-priority bin sees a worst-case delay of up to *1024 cycles*. Thus, better performance in the higher priority bin comes at the expense of significantly lower performance in other bins, and the packet delay distribution highlights the need for a dynamic priority-update feature that mitigates the observed aggressive starvation of low-priority packets.
- The results with the Hoplite-Q* NoC (Figure 4.11d) showcases a more balanced solution by accounting for dynamic conditions in the NoC. The worst-case latency suffered by a packet in the highest-priority bin is now 14 cycles while that in the lowest-priority bin is a much more reasonable 141 cycles. This marginally sacrifices the performance of the higher priority bins for balanced outcomes across all other bins.

Answer: Hoplite and Hoplite-B produce a uniform packet latency distribution across varying priority-levels, while Hoplite-Q and Hoplite-Q give a decaying curve that indicates sensitivity to packet priority. Hoplite-Q* delivers the best balance between priority-aware routing and worst-case packet delay.*

4.5.4 Priority-Tag Bitwidth

Question: How many bits (P) should be reserved for the priority-tag in the NoC? How does it impact throughput performance of the top-priority application as number of concurrent applications is varied?

One of the key design considerations is the bitwidth allocation for the priority-tags, especially with dynamic priority-updates in Hoplite-Q*. We now experiment with mixed-priority, multi-application BSP workloads where a varying number of applications are sharing the compute and communication resources available on the NoC. We take a single real-world BSP application (`bp1600`) and replicate it multiple times while assigning a different priority class to each replicated copy. Figure 4.12 shows the throughput improvements observed on the *top-priority application* with Hoplite-B, Hoplite-Q, and Hoplite-Q* NoC when compared to the baseline Hoplite NoC.

In order to interpret Figure 4.12 correctly, we first highlight two factors that impact the experimental observations in unpredictable ways:

1. Figures 4.12a, 4.12b, and 4.12c are speedup comparisons against the Hoplite NoC. Since the Hoplite router is agnostic to any priority-tags attached to packets, there is some randomness in the runtime of each application, which when compared to as a baseline can result in the observed minor fluctuations in speedup at varying configurations.

2. Another source of randomness is the placement of applications on the overlay. Since the NoC has a 2D unidirectional torus topology, placement of the application graphs can have a significant impact on runtime performance, simply because of the long wrap-around enforced by the unidirectional torus layout. For example, PE (0,0) sending packets to PE (0,1) has a very different communication profile than the converse, where the packets from PE (0,1) to PE (0,0) have to traverse the entire length of the overlay to wrap back around to reach their destination. To isolate the effect of priority-aware routing, we use a simple application placement strategy where we try to partition the overlay equally such that each application is allocated a fair share of the compute resources. In Figure 4.12, the compute resources (*i.e.* 64 PEs) are not equally divisible for certain configurations (*e.g.* 5 concurrent applications), which can also result in some of these observed fluctuations.

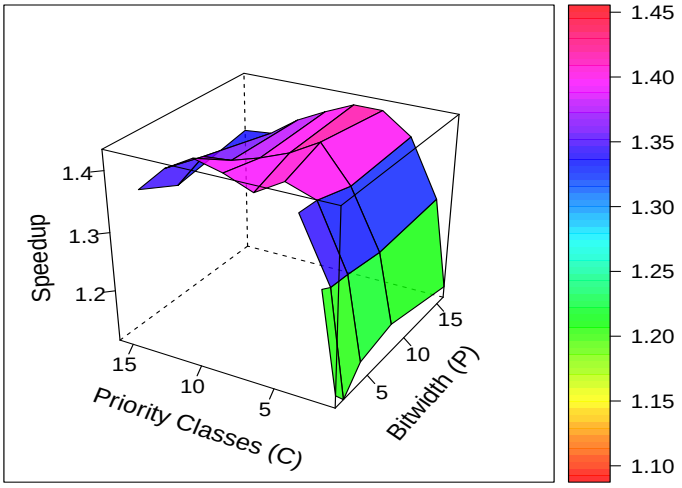
Sometimes, the placement of an application to a block of PEs can just get (un)lucky, which has a higher tendency to occur when a larger number of applications (i.e. larger number of priority classes) are present.

Nevertheless, despite the QoS delivery being susceptible to these random effects, they do not have a significant impact on overall performance, as we obtain a desirable QoS behaviour for top-priority application packets.

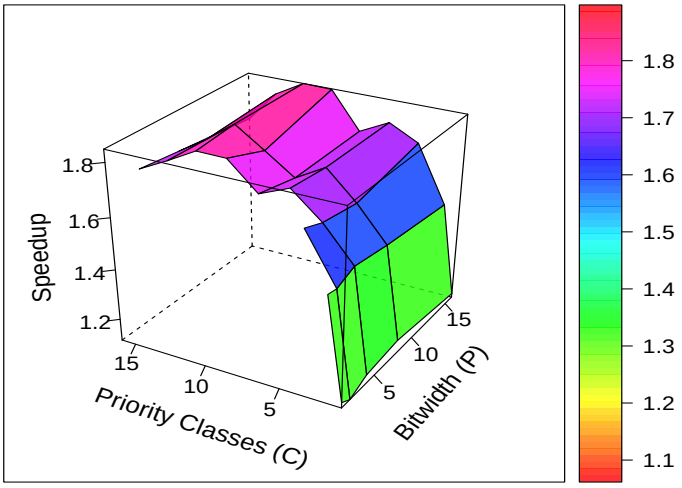
Across all designs in Figures 4.12a, 4.12b, and 4.12c, we see an increase in the speedup as number of applications is increased. This can be attributed to the increase in communication traffic that causes each overlay to achieve its peak sustained throughput. For example, when < 5 applications are instantiated on the overlay, Hoplite-B delivers only about 10–20% improvement over Hoplite (see Figure 4.12a), all of which can be attributed to the usefulness of the buffer in mitigating the effects of deflection. However, as observed in Figure 4.9, Hoplite-B is capable of delivering up to 1.5× improvement in speedup over Hoplite (at an 8x8 overlay configuration). When the number of concurrent applications is increased, we observe the speedups increase to ≈1.45×. Since there is no priority-aware arbitration inside Hoplite-B, the net speedup of the top-priority application is slightly below the expected peak simply due to randomness.

Contrast that to Hoplite-Q and Hoplite-Q* in Figures 4.12b and 4.12c respectively, where speedups to the top-priority application breaks past the 1.5× "buffer-only wall" , and reaches 1.8×–1.9× for the top-priority application.

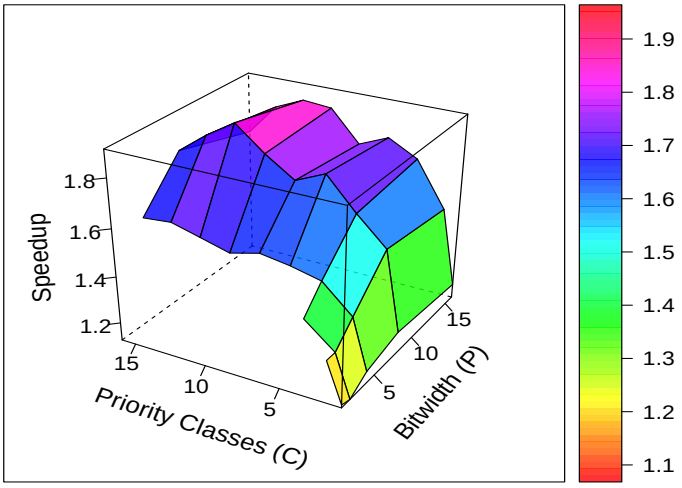
As expected, due to the lack of a priority-aware routing function, increasing the number of bits (P) in the priority-tag has no noticeable impact on performance with a Hoplite-B NoC. While Hoplite-Q does use the priority-tag for making routing decisions, there are no updates made to the packet priorities at runtime, and hence, there is also no noticeable impact on performance by varying P, *i.e.* we can remove the dynamic tag and choose P bits such that number of desired priority classes is $\leq 2^P$. Hoplite-Q*, on the other hand, demonstrates a more noticeable change in performance with varying P. At small P, the numerical range of the dynamic tag is not large enough, such that deflected packets from the lower-priority classes get quickly promoted to higher priority classes, and limit the performance advantage reserved for the top-priority application. At large P, the effect of the dynamic priority-updates is diminished, as it takes many more deflections for packets to get promoted across different priority classes. This results in more low-priority



(A) Hoplite-B vs Hoplite



(B) Hoplite-Q vs Hoplite



(C) Hoplite-Q* vs Hoplite

FIGURE 4.12: Observed throughput improvements over 8×8 Hoplite NoC for the application in the top-priority class, tested over varying number of priority classes (C = 1→16), and total bitwidth of the priority-tag (P = 1→16).

packets getting stuck for long periods in the NoC, which takes up routing resources (*e.g.* buffers) and, on the whole, throttles the performance of the NoC. The design optimum seems to be $P = 8$ for the range of priority classes tested in our experiments, and is likely to be larger as we scale up further beyond 16 priority classes. When compared to Hoplite-B and Hoplite-Q, Hoplite-Q* also delivers the highest peak throughput performance of $> 1.9 \times$ over Hoplite (when $P = 8$).

Answer: From empirical observations, $P = 8$ delivers the optimum throughput for the top-priority application when up to 16 concurrent applications are deployed on an 8x8 NoC overlay. This optimum is likely to shift slightly as number of concurrent applications and/or size of overlay is increased.

4.5.5 Throughput vs average latency

Question: How is throughput and packet latency impacted across different priority classes?

In this subsection, we look more closely at the effect of priority-aware routing on packets from all priority classes. All results detailed in this subsection, unless otherwise stated, are evaluated with the `bp1600` BSP benchmark.

FIGURE 4.13: Average throughput of application in each priority class on 8×8 NoC, where C = 16 and P = 16.

Figure 4.13 shows the effect each router has on the throughput of applications in each priority class (labeled C0 to C15, in increasing order of priority). As expected, Hoplite and Hoplite-B NoCs produce a fairly uniform trend across all priority classes since they do not have any priority-aware routing features. Hoplite-Q produces a stark upward trend in the throughput performance from C0→C15. Hoplite-Q* distributes this effect slightly to produce a more balanced throughput performance across all priority bins (*i.e.* range of average throughput delivered to least priority to highest priority applications is slightly lower in Hoplite-Q* than Hoplite-Q). When comparing throughput performance between C0 and C15, there is a ≈50% improvement in throughput for the top-priority application with the Hoplite-Q/Hoplite-Q* routers.

This throughput performance is supported by Figure 4.14 as well, where we track and compute the average packet latency suffered by packets in each priority class. Here, the quality of service provided to top-priority packets (C15) can be almost 4× better than lowest-priority packets (C0). Note that an overwhelmingly large proportion of the routing latency is due to the injection queue at the PE input, and hence, the eventual throughput improvements across priority bins saturate at smaller values.

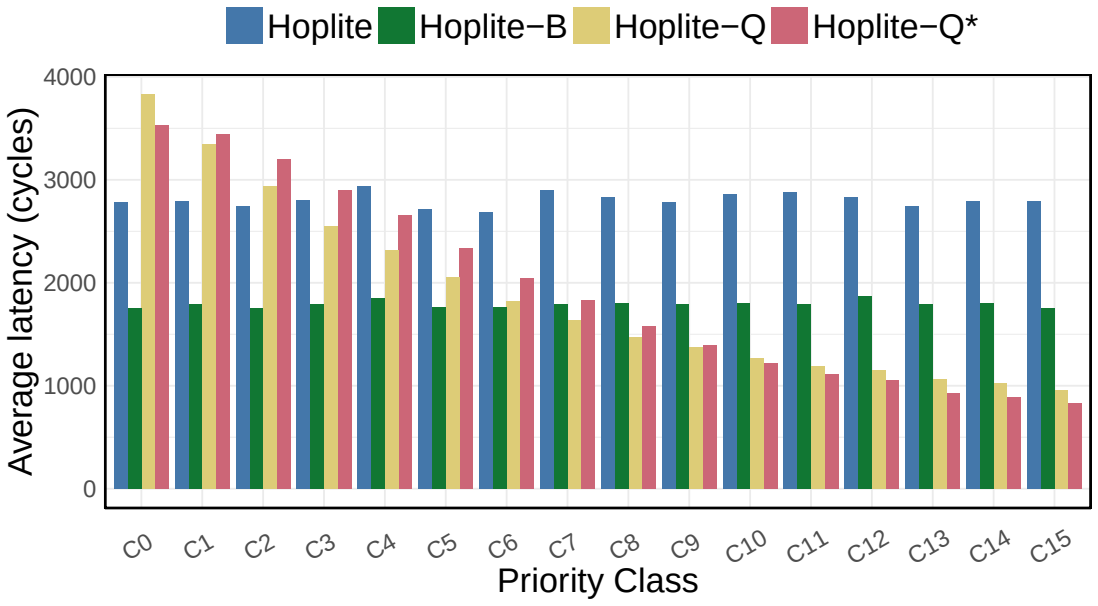


FIGURE 4.14: Average latency suffered by packets in each priority class for 8×8 NoC, where C= 16 and P = 16.

Figure 4.15 shows the throughput and packet-latency performance of the *top-priority* application for different benchmarks. Each benchmark has its own communication pattern, and the sizes vary from 2k – 18k edges (packets).

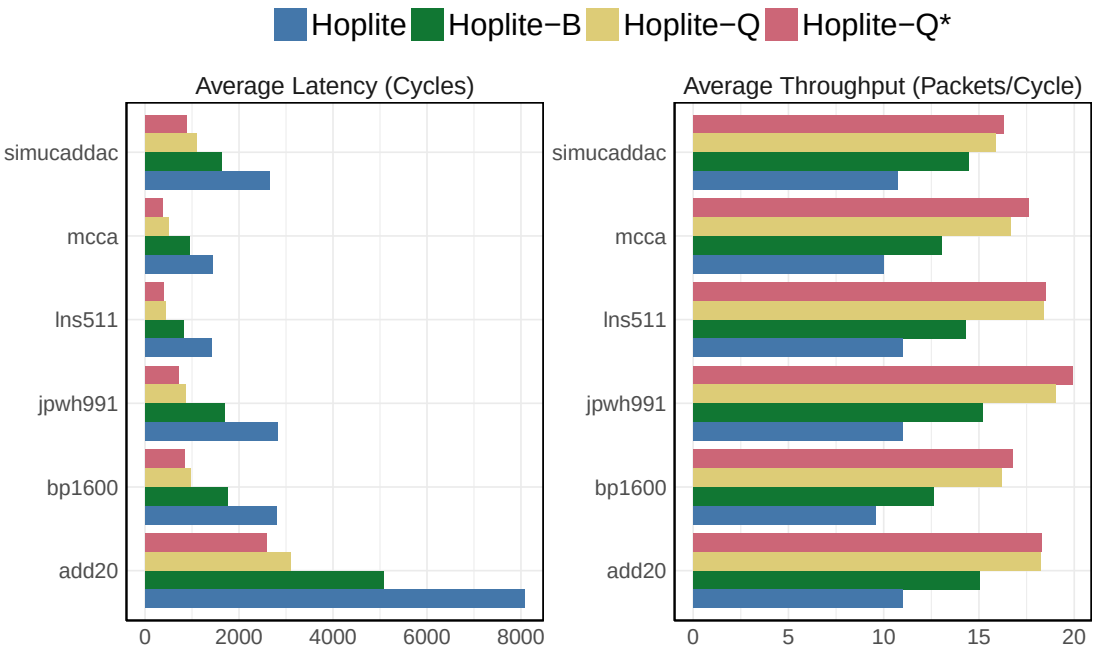


FIGURE 4.15: Average throughput and packet-latency of application packets in the top-priority class across different BSP benchmarks for an 8×8 NoC, where C = 16 and P = 8.

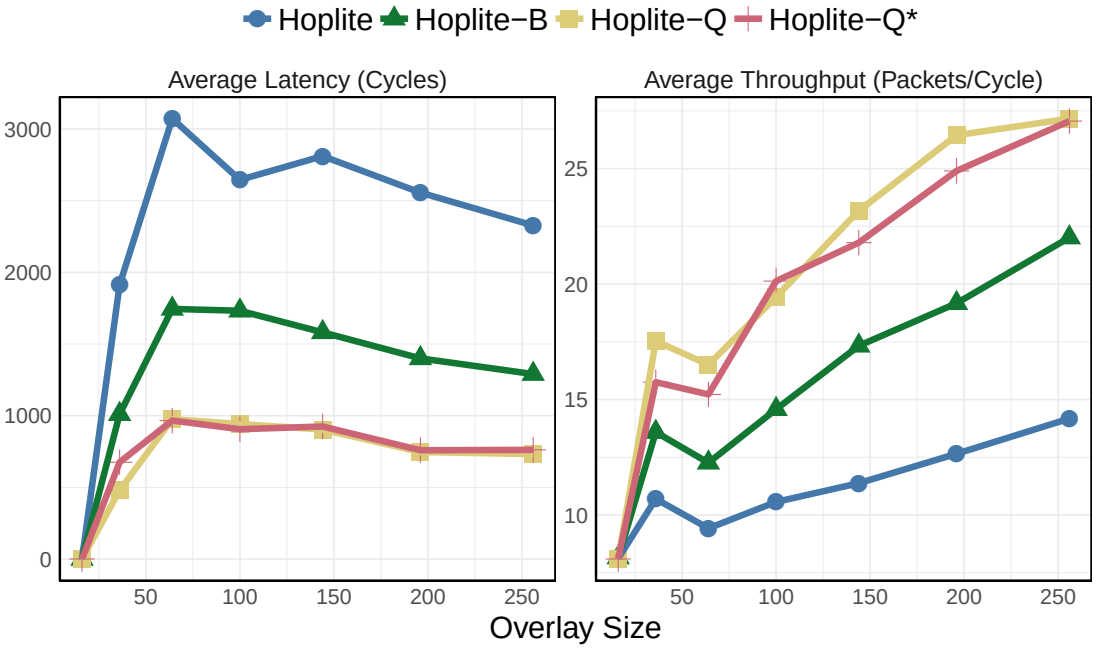


FIGURE 4.16: Average throughput and packet-latency for the top-priority application packets vs overlay size. C = 16, and P = 16.

Finally, in Figure 4.16, we observe that Hoplite-Q and Hoplite-Q* both scale desirably as size of the overlay is increased.

Answer: Top-priority applications observe ≈50% throughput improvement, and ≈4× improved average packet latency. These trends continue desirably as overlay size is increased, and apply to multiple benchmarks.

4.5.6 Token Dataflow

Question: How much impact does the Hoplite-Q router have on the token dataflow benchmarks evaluated on DaCO?*

Dataflow graphs have a distinct critical path that should be prioritized for optimal acceleration. In Chapter 3, we added criticality-aware scheduling inside each PE, and in this section, we explore the benefits of criticality-aware routing in the NoC. To isolate the effects of criticality-aware routing with Hoplite-Q*, we first use the baseline in-order dataflow processor as the PE in our experiments, and turn off clustering in the NoC topology (referred to as `DF Baseline` in Chapter 3). This ensures that any benefits from priority-aware routing is solely due to the features offered by the NoC router. We use priority and criticality interchangeably in this section, *i.e.* each edge criticality in the dataflow graph is quantized to an appropriate priority bin that Hoplite-Q/Hoplite-Q* utilizes to route dataflow communication traffic in a criticality-aware manner.

Figure 4.17 shows the results when the `bomhof3` benchmark is evaluated with varying priority-tag bitwidth (P). Hoplite-B delivers the most significant throughput improvement over baseline Hoplite router of about 13%. As P is increased, Hoplite-Q and Hoplite-Q* are able to exploit the critical path in the dataflow graph by routing traffic by criticality, which delivers a peak throughput of around 16% over Hoplite.

Figure 4.17 also suggests that P = 4 is sufficient for a dataflow graph of this size. At low bitwidths, Hoplite-Q* performs poorer than Hoplite-Q. This is because, at low bitwidths, the numerical range is small, and hence, the +1 added to the priority-tag upon deflection is a significant boost to the priority of a packet. This has a masking effect on the critical path, and hence, the performance from Hoplite-Q* mimics that of Hoplite-B at low bitwidths.

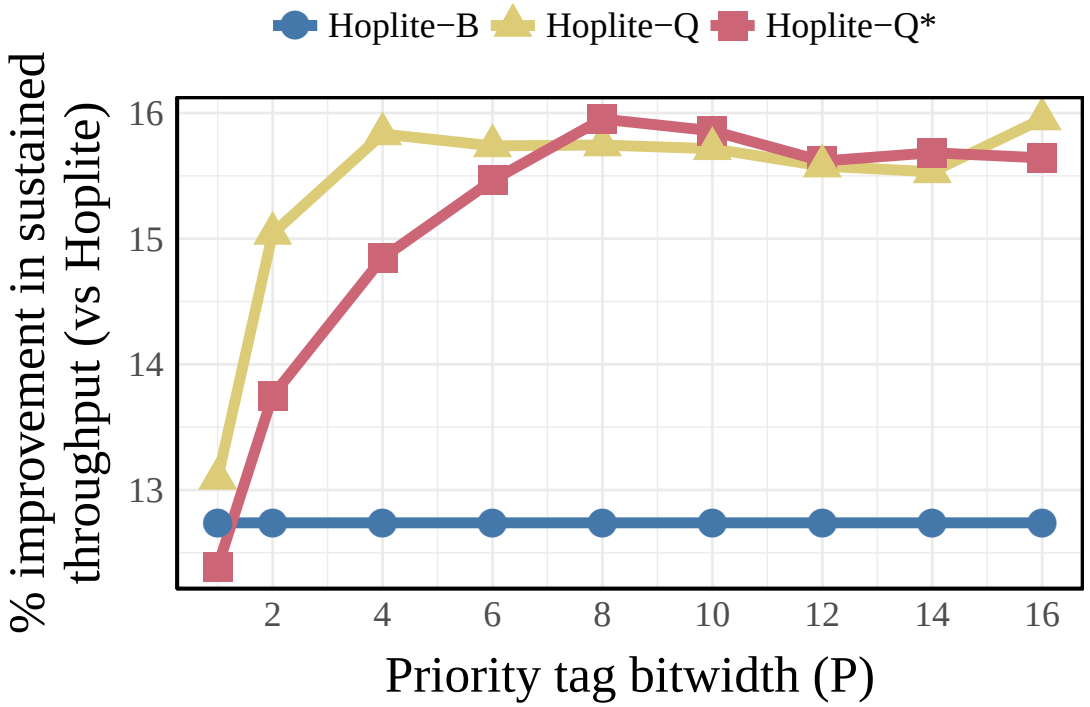


FIGURE 4.17: Average throughput improvement (percentage) vs Hoplite for `bomhof3` with varying P.

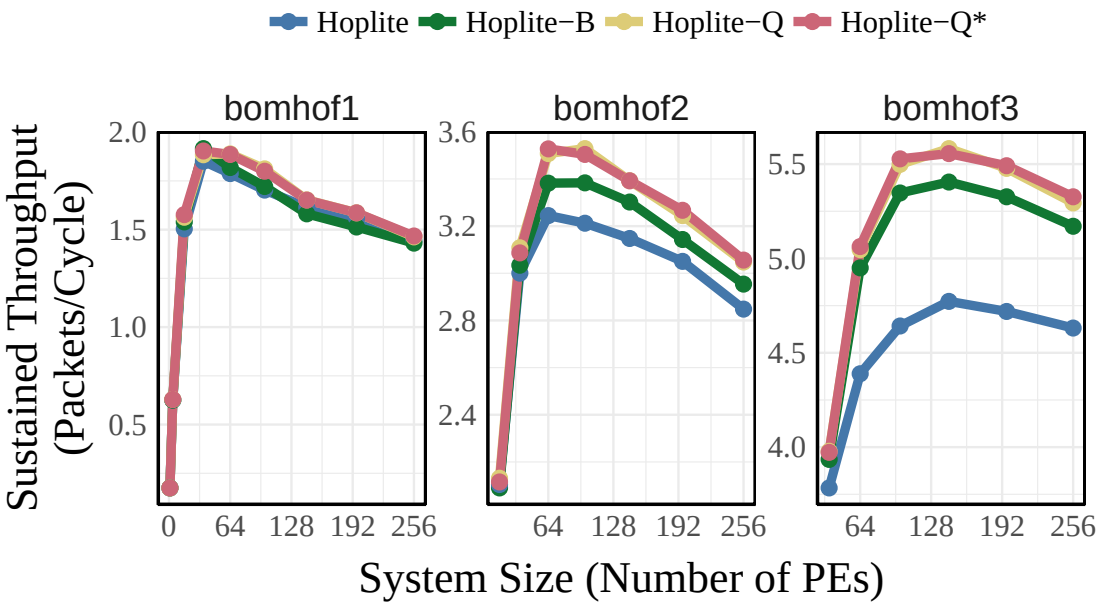


FIGURE 4.18: Throughput performance of three representative dataflow graph benchmarks under different NoC routers.

The number and distribution of nodes/edges along the critical path can result in varying performance when evaluating different dataflow benchmarks. In this

subsection, we look at three representative dataflow graph benchmarks of increasing sizes – `bomhof1` (2k/2.4k nodes/edges), `bomhof2` (36k/46k nodes/edges), and `bomhof3` (75k/90k nodes/edges). Figure 4.18 shows the throughput performance of these three benchmarks when evaluated with different NoC routers and overlay sizes (priority-tag bitwidth is set to 4, only applicable to Hoplite-Q/Hoplite-Q*). As expected, speedup from Hoplite-B, Hoplite-Q, and Hoplite-Q* increases as size of the input dataflow graph increases, since more traffic can take advantage of the buffering and the priority-aware routing features.

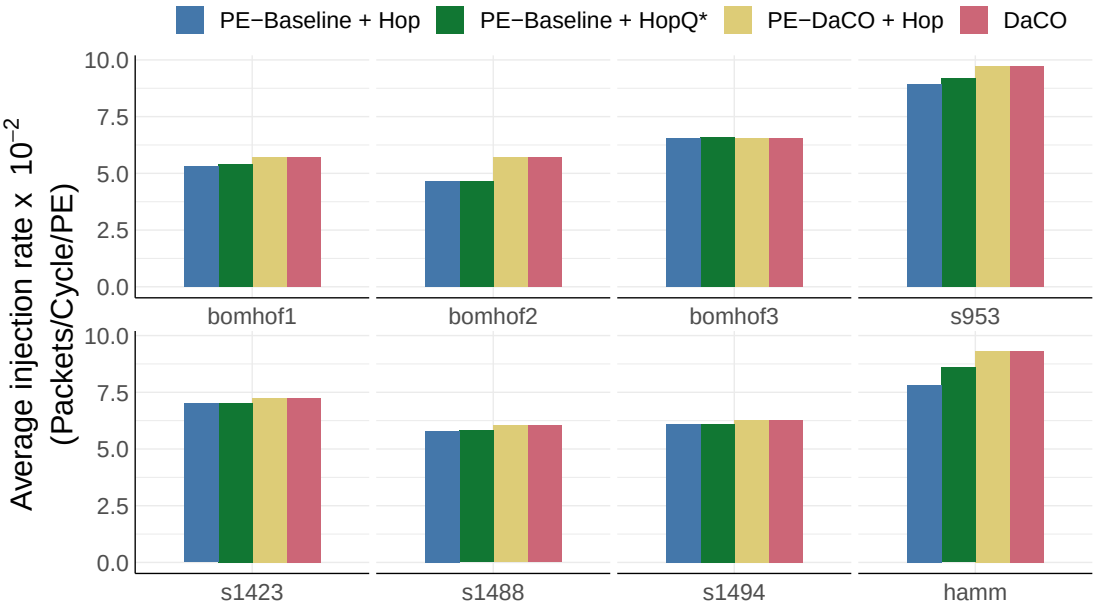


FIGURE 4.19: Observed average injection rate with four different overlay configurations. PE-Baseline is the baseline in-order dataflow PE, PE-DaCO uses LOD-based out-of-order scheduling, and DaCO is PE-DaCO with Hoplite-Q*. Priority-tag bitwidth is set to 8 when Hoplite-Q* is used. The average injection rate is computed across all cluster sizes (1, 2, 4, 8 and 16).

One way to quantify the role of the communication framework in the performance of the overlay is by measuring the injection rate of packets into the network. A high injection rate means that the communication network is relied on heavily, and thus is likely to have a significant impact on overall performance. Figure 4.19 shows how the average injection rate of each benchmark is affected by two main overlay components: (1) the PE, which is either the older dataflow PE that does in-order FIFO-based scheduling (PE-Baseline), or the proposed design in Chapter 3 that introduces out-of-order scheduling (PE-DaCO), and (2) the NoC router, which is either the baseline Hoplite router, or the priority-aware Hoplite-Q* router proposed in this chapter.

Going from {PE-Baseline + Hop} to {PE-Baseline + HopQ*}, we observe a small increase in injection rate, which shows that by routing packets in the network by priority, we improve the execution order of the dataflow graphs slightly. This is corroborated by the marginal improvements observed in Figure 4.18 (<10%).

There is a much more significant jump in injection rate when we switch PE-Baseline with PE-DaCO in the {PE-DaCO + Hop} overlay. This change produces much more favorable speedups of up to 2.6×, as shown earlier in the earlier chapter (see Figure 3.9 in Section 3.6.3 in Chapter 3).

The sole impact of Hoplite-Q* drops even further to <1% when PE-DaCO is in use. Hence, we can conclude that most of the advantages from criticality-aware execution are derived from a better PE instead of the communication framework.

Interestingly, the injection rates for dataflow benchmarks across all configurations are <10%, as only a subset of edges are active in the dataflow graph each cycle. The baseline Hoplite router is capable of handling such low injection rates, which unfortunately diminishes the importance of priority-aware routing offered by Hoplite-Q*, and hence is another reason why we observe very little, if any, improvement in performance when Hoplite-Q* is used with PE-DaCO.

Nevertheless, we believe Hoplite-Q* can be useful to DaCO in the future under two conditions:

1. The benchmarks evaluated in this study so far are fairly small due to simulation tool limitations. As we scale to larger dataflow graphs, we expect more communication traffic that stresses the importance of the NoC further for token dataflow problems.

2. Hoplite-Q* is also likely to play an important role in enabling dynamic dataflow execution on DaCO. In dynamic dataflow, multiple iterations of the dataflow graph can be active in any given cycle. Priority-aware management of traffic across these multiple iterations is likely to deliver better runtime performance, especially in latency-critical workloads (*e.g.* machine learning).

Answer: Hoplite-Q has limited impact on token dataflow execution with DaCO (<1%) on the current set of benchmarks. This can be attributed to the lack of communication traffic in the small benchmarks, since the low injection rate (<10%)*

of packets into the network limit the impact Hoplite-Q can potentially have on overall performance.*

4.6 Future Work

There are several improvements and research extensions that could be made to the body of work in this chapter. We list three potential options that we hope to try in the near future:

1. The arbitration cost inside Hoplite-Q/Hoplite-Q* is unusually large. One of the goals in the future would be to revisit the arbitration strategy in order to condense the design further. A more compact design would enable us to achieve a better resource-balanced DaCO architecture.

2. Instead of a counter-based dynamic priority-aware routing strategy, a RNG-based (random number generator) strategy could potentially give similar priority-aware routing behaviour. Using a cheap hardware RNG, like a linear-feedback shift register (LSFR), packet arbitration could be guided by pseudo-random arbitration, where the probability of acquiring a desired route is correlated to the static priority level of the packet. This has two key advantages: (1) since only static priority bits are needed, the wiring cost drops significantly with this approach, and (2) as size of priority-tag increases, the cost of the counters start to burden the design, which would not be the case with the LSFR-based design.[1]

3. One option that would improve the sustained throughput of the Hoplite-Q* NoC significantly while raising some interesting research questions is to change the unidirectional links between routers to bidirectional links. This strategy essentially changes the topology from a torus to a mesh, which would have several interesting impacts on the design. First of all, it would increase the throughput of the NoC, simply due to the additional bandwidth available from the extra wires. The deflection routing algorithm would also need to change, especially the priority-aware arbitration function. Since packets can be deflected to multiple output ports now, the priority-aware routing function would likely be richer and require more logic resources to implement.

[1] This clever little idea was suggested by Jan Gray at FCCM 2018.

If the associated overheads of wiring and logic are acceptable with this approach, the buffer could also be resized to achieve a new tradeoff balance between throughput and resource utilization (likely that an extra buffer or two would benefit greatly due to extra output ports at each router). How these extra buffers are used in the priority-aware arbiter would also be an interesting research question (*e.g.* buffer0 for only north/south + west/east links, buffer1 for south/north + east/west links).

4.7 Conclusions

This chapter introduces Hoplite-Q*, a lightweight FPGA-friendly priority-aware NoC router that is capable of exploiting priority information embedded in NoC communication workloads. The proposed architecture in this chapter modifies the lean Hoplite FPGA NoC router by adding a single buffer to enhance routing choice. We also design the routing function to use the priority-tags in each packet when determining packet paths. Our proposed design improves throughput of the top-priority workload in a mixed-priority multi-application environment by 1.3–1.8×, and worst-case latency by 1.5–3.9× for BSP applications, while increasing FPGA area cost by 3.8×. Unfortunately, for token dataflow applications, the impact of Hoplite-Q* is minimal due to low injection rates in dataflow systems. In the future, experiments with larger benchmarks and new execution modes could emphasize the importance of a Hoplite-Q* NoC further.

All in all, Hoplite-Q* augments the baseline Hoplite router with priority-aware arbitration that is capable of delivering varying Quality-of-Service to applications running in a shared environment. In the acceleration domain, the Hoplite-Q* NoC complements the criticality-aware execution theme of DaCO, and delivers an efficient dataflow coprocessor overlay for FPGAs. In the next chapter, we look at dataflow-driven software optimizations that call upon hardware-software co-design principles to deliver further improvements.

4.8 Publications

The body of work presented in this chapter has been published in the following peer-reviewed conference proceedings:

Siddhartha, Nachiket Kapre, *Hoplite-Q: Priority-Aware Routing in FPGA Overlay NoCs*, International Symposium on Field Programmable Custom Computing Machines, May 2018 *(Full Paper)*

Chapter 5

Software Optimizations

5.1 Introduction

Computation expressed as dataflow graphs expose all kinds of potential parallelism in an algorithm. As seen in earlier chapters, hardware implementation of explicit dataflow execution using the token dataflow paradigm helps to manage this extra parallelism at low cost. Despite the custom dataflow hardware, dataflow graphs in their raw form have sequential dependencies that can limit runtime performance on DaCO. This chapter identifies three key areas of improvement that can be handled by the compiler in order to boost runtime performance: (1) loop-unrolling, (2) smart criticality-aware reassociation, and (3) fanout decomposition.

One of the most well-known methods for overcoming serialization bottlenecks and exposing parallelism is loop-unrolling. Loop-unrolling exposes additional parallelism and trades off extra work for faster evaluation. Modern compilers are capable of performing loop-unrolling optimizations (*e.g.* `-funroll-loops` in GCC), and our goal is to achieve something similar in the context of dataflow graphs and the underlying DaCO engine. In dataflow graphs, loop-unrolling can be implemented simply by duplicating bottlenecked regions of the graph such that these copies can all be evaluated in parallel. There is however a caveat: unrolling too much could have a detrimental effect on the overall performance, as the excesss computation and communication could result in undesirable effects such as congestion. This can be likened to the side effects of unrolling large loops in C programs, where a large increase in the number of unrolled instructions ends up slowing down the overall evaluation [114]. In this chapter, we explore the limits of loop-unrolling for

the DaCO engine, and demonstrate how unrolling and reorganizing (reassociating) the graph *carefully* at compile time can deliver runtime benefits.

Runtime performance can also be limited by worst case arrival times of fanins to a long chain of associative nodes (*e.g.* summation of a series of terms represented as a chain of add nodes). The naïve strategy would be to restructure these chains into balanced binary trees to achieve a $\log_2$-reduction in the critical path of the computation chain. However, the arrival times of each input into these chains can vary significantly. There are potentially two factors behind this effect:

1. The dataflow graph structure simply creates this effect – *e.g.* both a constant node and an arithmetic node at depth N feeding into another arithmetic node will result in fanin arrival time to differ by at least N cycles.
2. Network effects like deflection and congestion lead to non-deterministic packet latencies.

While it can be difficult, or simply impossible, to predict runtime network effects, fortunately, the two factors listed above are loosely correlated. Intuitively, the latency of a node at a deeper depth accummulates a larger number of prior network delays, and hence, estimating arrival time of fanins using static analysis of the graph at compile time delivers a sufficient approximation. This is corroborated by observations in our experiments, where a small difference in expected arrival time is exacerbated by network effects, thereby improving the benefits of doing arrival-time aware graph structuring even further. To implement this, we design a graph reassociation strategy inspired from Huffman encoding, and demonstrate its benefits over the naïve reassociation methods.

Finally, dataflow graphs can also be plagued by serialization bottlenecks due to nodes with a large number of outgoing edges. This is mostly due to the underlying hardware architecture in the packet generator (see Chapter 3.4.1.4), where packets on all outgoing edges (fanouts) of a node are constructed and injected into the network serially one at a time. Often, these large fanout nodes also lie on the critical path as their result is required by a large number of nodes downstream. While modification to the underlying hardware to support parallel fanout evaluation/injection is potentially a viable strategy, we opt to keep the lightweight processor design and focus on software-based graph transformation optimizations instead. Our solution is to decompose these high fanout nodes into multiple balanced fanout

trees that can then be placed onto different processors. This optimization can be likened to the fanout tree construction carried out by the EDGE compiler [46], since each instruction is only allowed up to two target instructions on their architecture. Our optimizer instead can operate with user-supplied thresholds and fanout-arity hyperparameters. Nodes in these fanout trees can then be placed on different processors, such that multiple processors in the network can share the workload of processing these large number of fanouts in parallel.

The remainder of this chapter is organized as follows: Section 5.2 gives a brief background overview on the benchmarks used in this chapter, and also highlights some related work. Section 5.3 introduces our first loop-unrolling optimization, which we refer to as recursive substitution and reassociation. Section 5.4 goes into detail on the design of a Huffman-inspired criticality-aware reassociation scheme that improves upon the naïve reassociation in Section 5.3. In Section 5.5, we introduce our final compiler optimization on fanout decomposition. Finally, Sections 5.6 and 5.7 contain details on the experimental setup and discussions on the results respectively, with final concluding remarks in Section 5.9.

5.2 Background

5.2.1 Sparse Matrix Factorization

Numerical problems in computing often require solutions to systems of linear equations, represented as $A\vec{x} = \vec{b}$. To solve these equations, we can use either direct or indirect algorithms, but we focus on direct LU factorization in this chapter. We consider factorization in scenarios that permit upfront, one-off static analysis of the computation to extract and exploit parallelism (*e.g.* KLU solver [30] for circuit simulation). The underlying parallelization technique also extends to other numerical routines operating on sparse matrices with irregular dataflow structure. The KLU solver performs a one-time spatial reordering of rows and columns in the matrix which makes it possible for the non-zero structure in the intermediate matrices to remain static/fixed for subsequent iterations. This feature allows us to pre-allocate and optimize the data structures at the start of an iterative simulation phase where thousands of factorizations may subsequently be performed. We adapt the solver to expose raw dataflow parallelism in the resulting unrolled compute graph for a token dataflow implementation.

Algorithm 1: Gilbert-Peierls

```
Data: sparse matrix A
Result: factors L & U
1 L = I;
2 for i=1:N do
3 |   b = A(: , i);
4 |   x = L\b;
5 |   U(1:i , i) = x(1:i);
6 |   L(i+1:N , i) = x(i+1:N) / U(i , i);
7 end
```

Algorithm 2: Front-Solve Loop

```
Data: kth column from sparse matrix A, partially computed factor L upto column
      k-1
Result: kth column for factor L and U
1 for i=1:k-1 do
2 |   for j=i+1:N do
3 |   |   if (x(i)!=0 and L(j,i)!=0) then
4 |   |   |   x(j) = b(j) - L(j,i)*x(i)
5 |   |   end
6 |   end
7 end
```

To understand the parallelization, we investigate the pseudocode of the Gilbert-Peierls (GP) algorithm, shown in Algorithm 1. The GP algorithm is at the heart of the LU decomposition routine in the KLU solver. Runtime is dominated by the repeated call to the front-solve (FS) routine (line 4) in every iteration of the *for* loop over matrix columns. In the FS algorithm, shown in Algorithm 2, the factored column of L and U matrices is generated in-place by iteratively processing the k-th column of A and using the partial entries in the L and U matrices. From the pseudo-code in Listing 1 and 2 it may appear that the matrix solve computation is inherently sequential. However, if we unroll those loops in both algorithms, we can expose the underlying dataflow parallelism in the sparse operations. In the parallel evaluation, we schedule operations purely on the basis of their dataflow dependencies rather than the artificial ordering imposed by sequential loop iterators. The sparsity and natural clustering of non-zeros in matrices plays to our advantage, as the unrolling generates independent subtrees with little or no communication with each other.

5.3 Recursive Substitution (Loop-Unrolling)

5.3.1 Motivating Example

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ L_{2,1} & 1 & 0 & 0 \\ L_{3,1} & L_{3,2} & 1 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

FIGURE 5.1: Toy 4x4 dense matrix example of a front-solve, where the matrix L and vector $\vec{b}$ are known constants, and we are solving for the vector $\vec{x}$

Consider the toy 4x4 dense matrix front-solve example in Figure 5.1. The front-solve algorithm is naturally sequential, as it solves for $\vec{x}$ one row at a time (see Listing 2). Figure 5.2 shows the dataflow graph that represents the front-solve computation for the toy example. Note how x_4 requires the results of x_1, x_2, and x_3 to be computed before its own result can be evaluated.

One way to break these sequential dependencies between x_1 to x_4 is to unroll the for-loops in Listing 2. This essentially creates four independent dataflow graphs that can be evaluated in parallel to compute x_1 to x_4 respectively. We refer to this as *recursive substitution*.

5.3.2 Recursive Substitution

TABLE 5.1: Various optimizations applied to an expression for variable x_4 in the $L\vec{x} = \vec{b}$ Front-Solve computation (Figure 5.1)

Arithmetic Expression for x_4	**Work**	**Latency**
Original Expression $b_4 - L_{43}x_3 - L_{42}x_2 - L_{41}x_1$	6×, 6+	8
Fully Substituted Expression $b_4 - L_{43}b_3 + L_{43}L_{32}b_2 - L_{43}L_{32}L_{21}b_1 + L_{43}L_{31}b_1 - L_{42}b_2 + L_{42}L_{21}b_1 - L_{41}b_1$	12×, 7+	5
Depth-Limited Substitution Expression $b_4 - L_{43}b_3 + L_{43}L_{32}x_2 + L_{43}L_{31}x_1 - L_{42}x_2 - L_{41}x_1$	7×, 5+	5

Recursive substitution is the process of replicating datapaths in a computation in a way that breaks computational dependencies, thereby allowing multiple computations to be carried out in parallel. This is a work-parallelism tradeoff, where

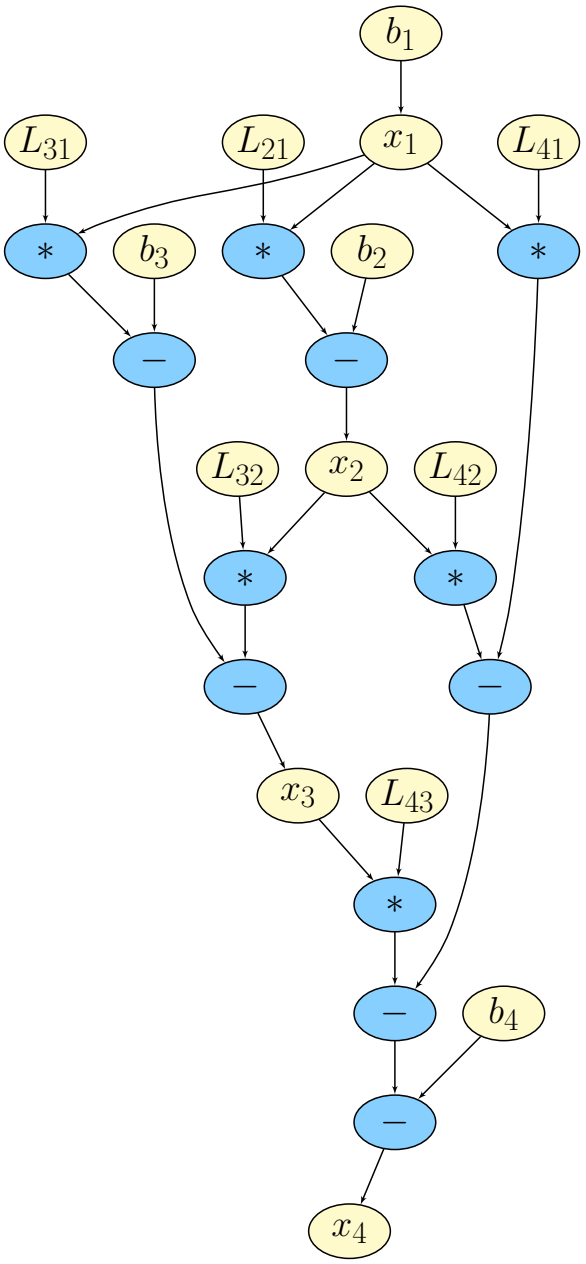


FIGURE 5.2: DAG for solving x_1, x_2, x_3, and x_4 in the toy 4x4 example

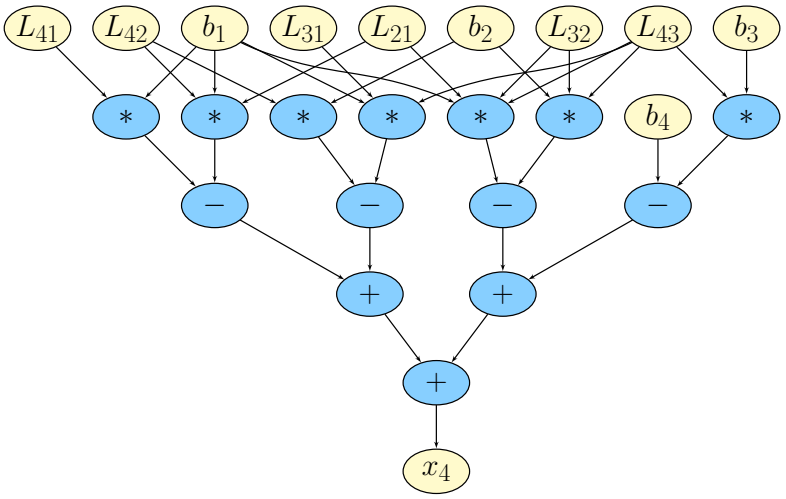
we increase the amount of computational work in order to gain runtime benefits from parallel execution. In sparse matrix factorization, this recursive substitution step can be applied to the front-solve in the Gilbert-Peierls algorithm (Line 4 of Algorithm 2). By doing recursive substitution, we rewrite the expressions for each x_j purely in terms of the vector $\vec{b}$ and partial matrix L_{ij}. We explain this in Table 5.1, where we first note the equations required to resolve elements in x_4 have a sequential order. If we recursively substitute the solutions for earlier x (x_1, x_2 and x_3) in the downstream expression for x_4, we can rewrite the equation such that it can be resolved independently without sequential dependencies.

Unfortunately, a complete recursive substitution for the entire computation gives us an exponentially larger compute graph for the LU factorization, with mostly redundant computations. Hence, our challenge is to create opportunities for asymptotic reductions while controlling growth in redundant work. While *sparse* matrices are expected to return a more manageable growth in work when recursive substitution is applied, unfortunately, however, we routinely observed a $> 30\times$ increase in duplicate work. We address this challenge by doing a depth-limited substitution instead. We show this by rewriting x_4 purely in terms of x_1 and x_2 in Table 5.1 for a depth $= 2$ case. By avoiding total recursive substitution, we reduce the amount of redundant work generated but accept a smaller saving in the reduction of the critical path of the graph. When implementing factorization on real-world dataflow hardware with capacity limits, this is an important engineering tradeoff that makes the idea feasible.

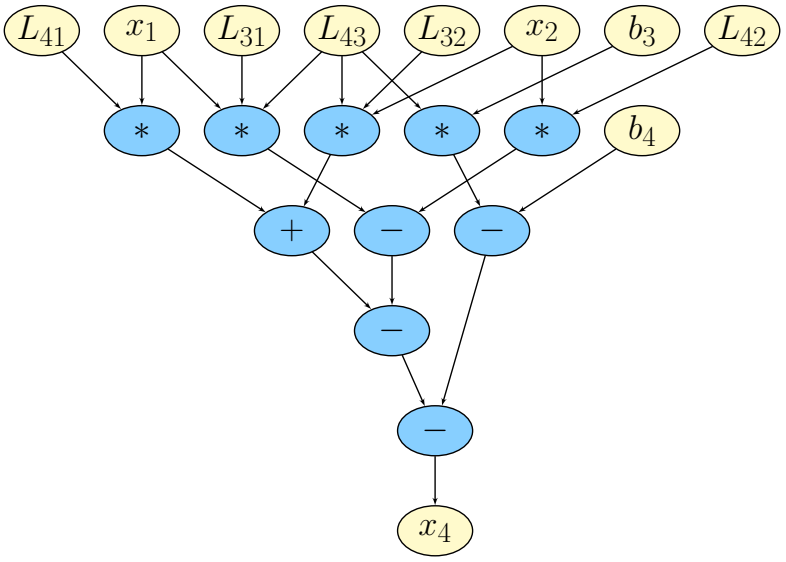
5.3.3 Reassociation

Substitution by itself decouples the computation and removes unnecessary dependencies, but it does not reduce the critical chain of operations. If we restructure the long $O(N)$ multiply and add chains generated from substitution into $\lceil log(N) \rceil$ trees, we can obtain an asymptotic reduction in critical latency. We quantify the improvements for the 4x4 example in Row 2 and 3 of Table 5.1 and associated dataflow graphs in Figure 5.3b and 5.3a. In Figure 5.4, we show the operation count and critical path latency trends for a single front-solve iteration in `bomhof2` benchmark when substituted to varying depths and subsequently reassociated. For most benchmarks, we observe that a substitution depth of 8 generates a good balance between additional work and reduction in critical path delays after reassociation. Note that reassociation results in reordering of the arithmetic operators, which could potentially affect the numerical stability of the algorithm due to finite-precision rounding effects. We discuss this in greater detail later in Section 5.7.4.

In Figure 5.5, we show a preliminary analysis of the utility of these software optimizations on dataflow properties of the `bomhof2` matrix. We plot the number of nodes in the graph at a given level (work) against latency of that node from the input (depth). Here, total depth (critical latency) is a measure of performance; smaller values indicate faster completion with parallel processing. A critical chain of operations that defines this maximum depth is called the critical path in the



(A) Without Depth Limits



(B) Depth-Limited
(depth = 2)

FIGURE 5.3: Substitution and naïve reassociation on x_4 in the toy 4x4 matrix example in Figure 5.1

computation. In the example shown, most of the parallelism is at the head of the graph (depth = 0). Across our benchmark set, we typically observed a similar phenomenon where we can issue as many as 10% of the operations in the first few steps of the graph.

For the example shown in Figure 5.5, we observe that we have a long sequential tail (depth≈75) that defeats parallel scaling. Thus overall performance is ultimately limited by the runtime spent evaluating this long tail as we delve deeper into the dataflow graph. Even when mapped to ideal hardware with no communication delays or scheduling bottlenecks, the resulting speedups will be limited by Amdahl's

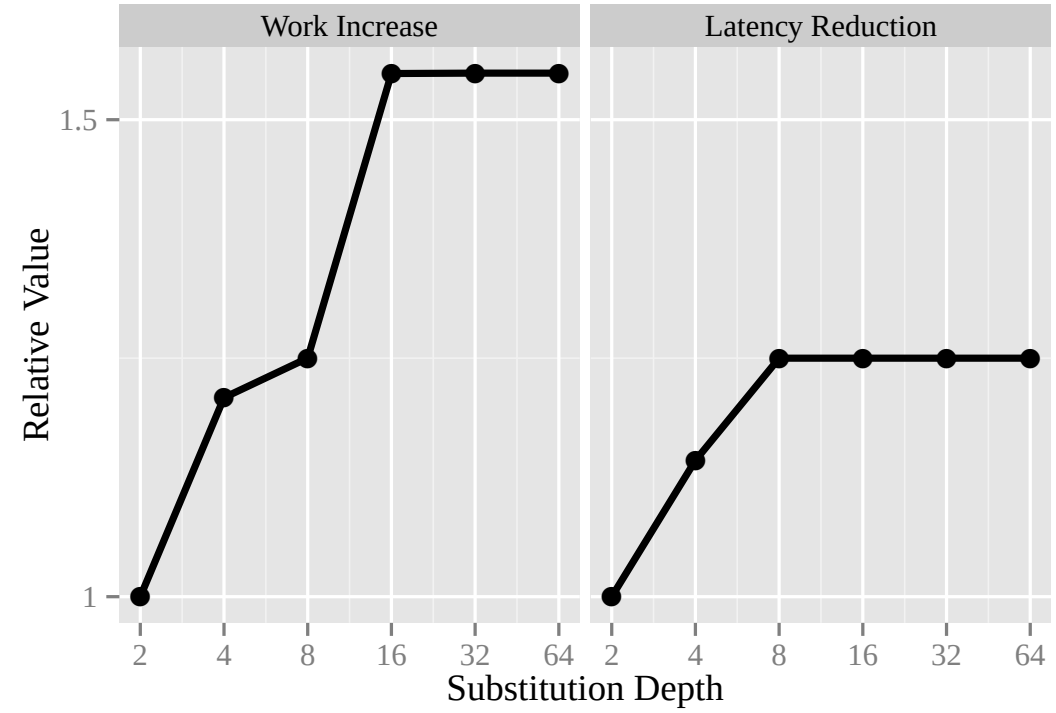


FIGURE 5.4: Work Parallelism Tradeoffs vary with substitution depth for `bomhof2` benchmark's dataflow graph. Large depths increase work excessively without reducing latency sufficiently.

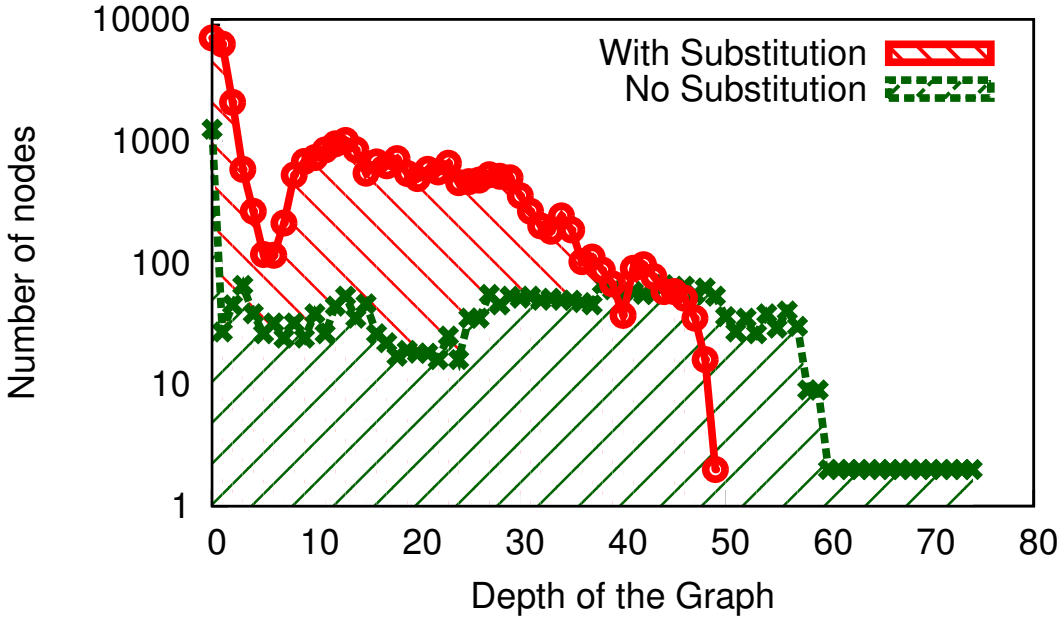


FIGURE 5.5: Work-Parallelism Tradeoff in (`bomhof2`) after applying substitution and reassociation

Law. This observation forms the basis of the software transformations in this chapter that coax additional parallelism from the stubborn sparse LU dataflow graphs.

When we apply our optimizations, we generate substantially more work (≈10×), but achieve critically important reduction in the depth of the graph (depth = 49) which reduces the critical latency in the computation (by 1.5×). This optimization is well-suited for architectures where independent floating-point operators are abundantly available and are tightly-coupled with low-latency interconnect to support irregular dependencies unlike multi-cores and GPU architectures with rigid shared-memory communication paradigms.

5.4 Criticality-Aware Reassociation

5.4.1 Motivating Example

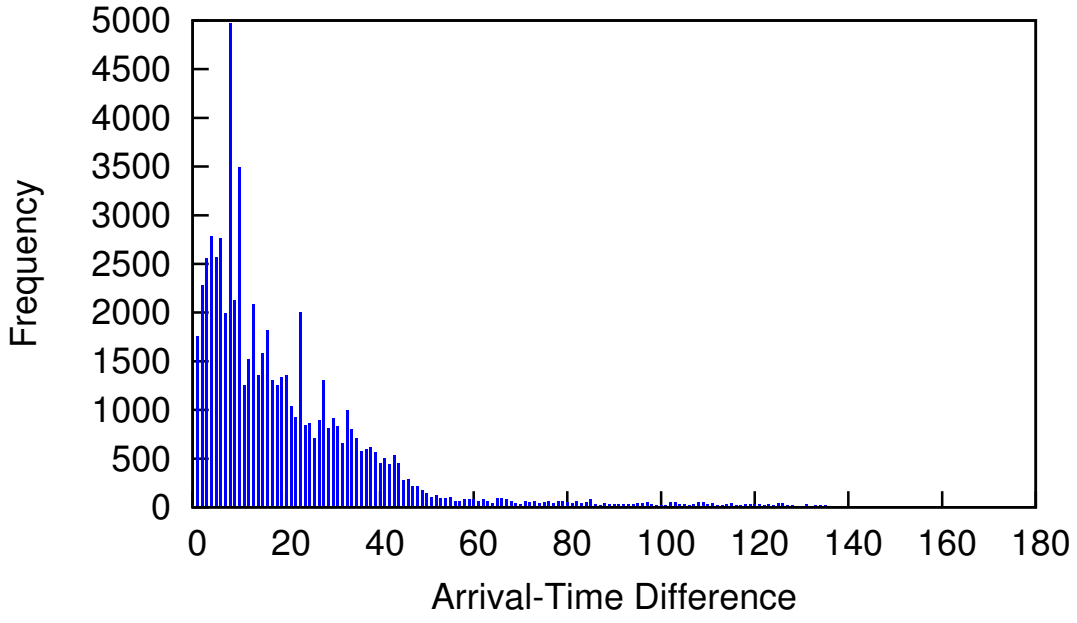


FIGURE 5.6: Arrival time variation at arithmetic operator inputs for `bomhof2` benchmark running on a 144-PE dataflow FPGA architecture. Latest arriving input at cycle ≈ 180 (difficult to see in plot)

Even after substitution and reassociation, poor performance is still possible due to the oblivious processing of high-fanin and high-fanout nodes without awareness of arrival time of the inputs. We could trivially decompose the high fanin nodes into balanced reduction trees using the associative property. Instead, we propose a reordering of operations in a manner that enhances parallelism. As the original order of floating-point operations has already been changed by substitution, this extra reassociation is considered with a focus on parallel performance. For balanced

reduction trees, we are assuming a uniform input arrival time. In hardware, we note significant gaps in arrival times for these fanin tree, as high as 180 cycles in some instances for the 144-PE overlay design (see Figure 5.6). Thus, our goal is to build associative fanin trees that account for this arrival time variation to improve completion times in these subtrees.

5.4.2 Huffman-based criticality-aware repacking

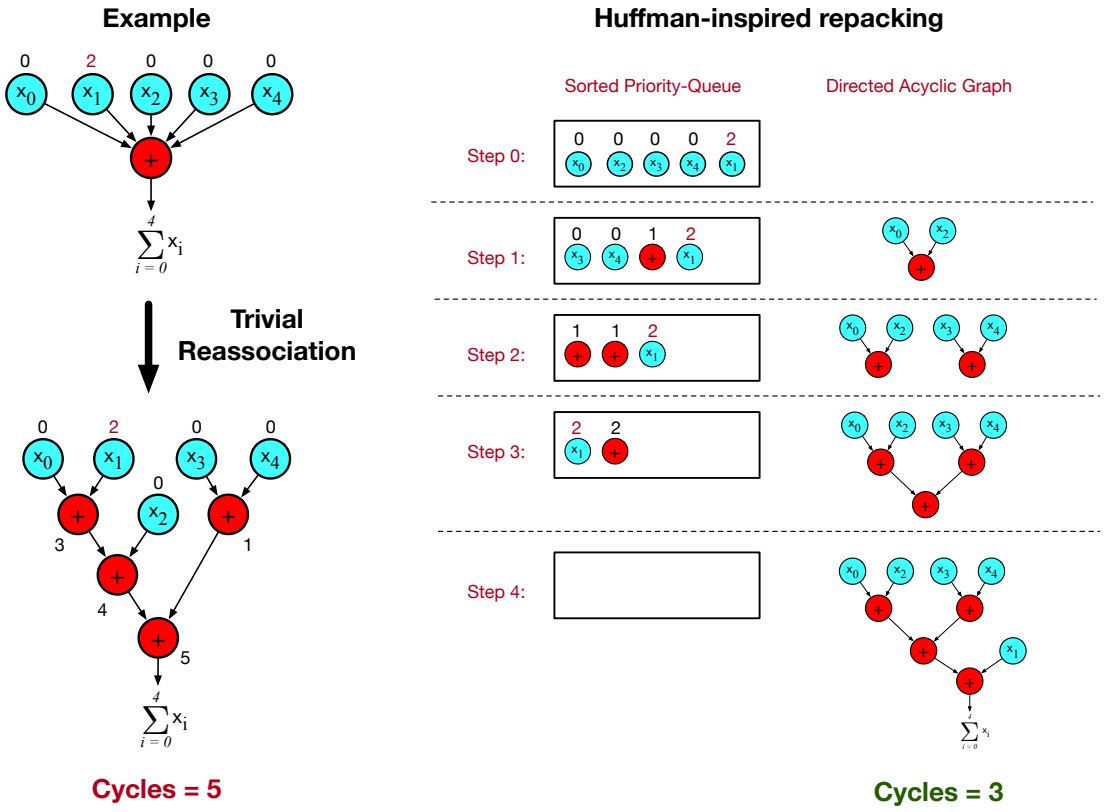


FIGURE 5.7: (1) Top-left: Compact DAG representation of a chain of add operations (top-left) representing a summation of five nodes. All nodes are available at cycle = 0, except x_1, which is delayed by 2 cycles. (2) Bottom-left: Trivial reassociation of add chain into balanced binary tree that takes 5 cycles to evaluate (assuming add instruction has a 1 cycle latency). (3) Right: Visualization of the 4-step process to reassociate smartly with a Huffman-inspired method that uses a sorted priority-queue to iteratively build the dataflow graph. Dataflow graph can be evaluated in 3 cycles now.

To implement criticality-aware repacking of computation represented in a dataflow graph, we draw inspiration from the Huffman-encoding algorithm [57] in a manner similar to the logic synthesis transformations presented in [52].

Algorithm 3: Huffman-styled fanin reassociation

Data: Priority Queue V, Input fanins $f_1, f_2, ..., f_n$ labeled with ASAP timing t_i
Result: Timing-optimized reduction tree

```
1  foreach f_i in f_1,f_2,f_3...,f_n do
2      t_i = f_i.getASAP();
3      f_i.setHuffmanTime(t_i);
4      V.push(f_i);
5  end
6  while V.size() != 1 do
7      inp1 = V.pop();
8      inp2 = V.pop();
9      op_node = createOperator();
10     op_node.connectInputs(inp1,inp2);
11     t1 = inp1.getHuffmanTime();
12     t2 = inp2.getHuffmanTime();
13     t3 = MAX(t1, t2) + 1;
14     op_node.setHuffmanTime(t3);
15     V.push(op_node);
16 end
```

In Huffman encoding, a tree is constructed based on the probability of occurrence of each input symbol being encoded. While constructing the tree, the symbol probabilities are accumulated incrementally and each stage has balanced probabilities. We adapt this algorithm to use arrival times instead of symbol probability when constructing the fanin tree for a high-fanin node (*i.e.* a chain of 2-input associative operations, *e.g.* add/multiply). We compute the arrival times based purely in the static structure of the dataflow graph through a simple as-soon-as-possible (ASAP) analysis. This is a lower-bound estimate of the time when the node will be available for downstream computations as we do not model network congestion costs and queuing of ready nodes in the PEs. In Figure 5.7, we show a contrived example of arity-2 fanin tree construction based on this adapted Huffman scheme. Algorithm 3 shows the pseudo-code for implementing this repacking scheme. This code has an asymptotic complexity of $N \times log(N)$ where N≈100s, thereby enabling rapid execution.

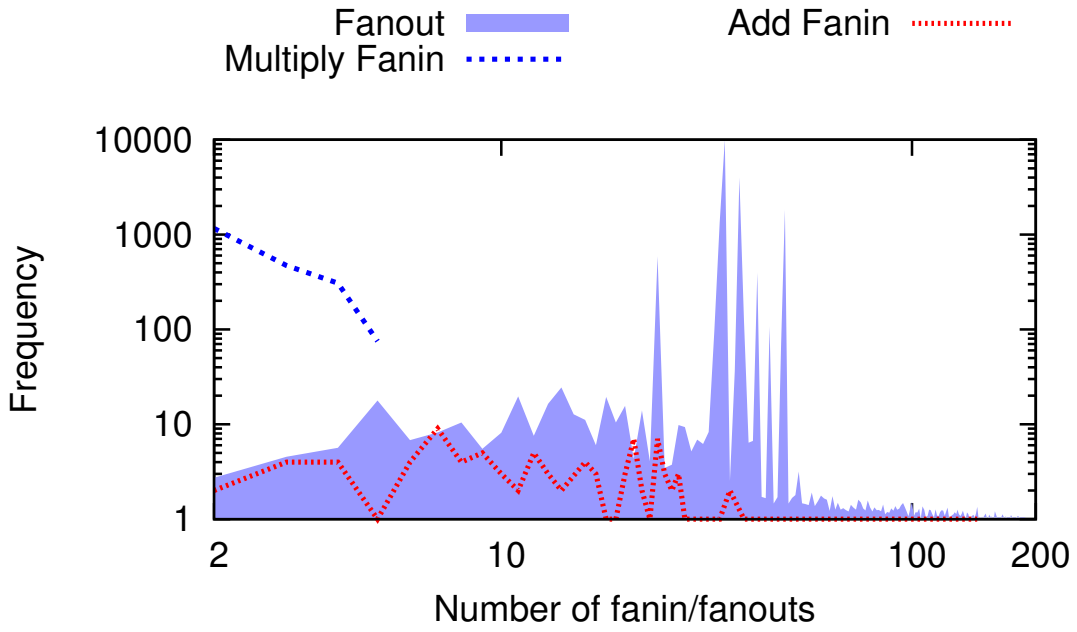


FIGURE 5.8: Fanin/Fanout Distribution of the `bomhof2` benchmark after substitution transformation.

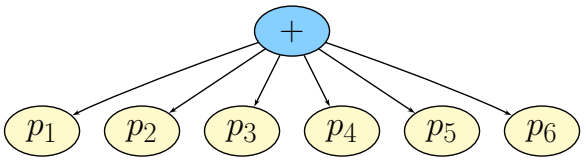
5.5 Fanout Decomposition

5.5.1 Motivating Example

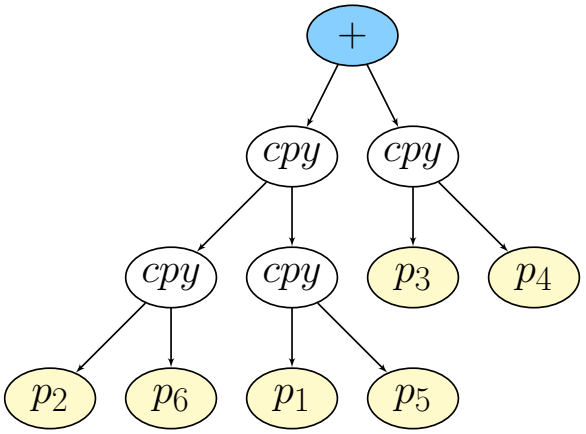
Fanout edges from a node represent dependencies where the result of a floating-point operation is required as input to multiple other operations. This result is explicitly communicated over the packet-switched network in the dataflow hardware. Unlike shared-memory programming, where the shared variables are snooped or explicitly requested from a shared address space, dataflow processing employs the message-passing style of communicating dependencies. Typically, within a PE, fanout edges emerging from a node are processed in sequence one at a time. This results in serialization bottlenecks at nodes with large fanout counts as shown in Figure 5.8 for the `bomhof2` benchmark. We need to reduce this overhead by (1) distributing fanout serialization across multiple PEs, as well as (2) prioritizing evaluation of edges that must travel further.

5.5.2 Implementation

To implement fanout decomposition, we first define two control parameters: threshold (f_t) and arity (f_a). If the fanout size of a node in our input dataflow graph is



(A) Undecomposed



(B) Decomposition $f_a = 2$, $f_t = 4$

FIGURE 5.9: Fanout Decomposition Example (*cpy* is a copy node)

greater than f_t, we perform fanout decomposition on that node. The decomposition is carried out such that arity of decomposed fanout tree is not greater than f_a, *i.e.* each node in the decomposed tree has no more than f_a fanouts. Under these constraints, we decompose the fanouts in the most balanced way possible, such that the fanouts are distributed across the new copy nodes (annotated as "*cpy*"-nodes in Figure 5.9b) as evenly as possible. Both f_t and f_a are tuneable parameters exposed to the programmer by our compiler.

5.6 Methodology

In this section, we describe our hardware design and software setup along with our experimental flow.

5.6.1 Old Hardware Design

We used the Xilinx Virtex-6 LX760 FPGA for our hardware experiments in this chapter. The hardware uses operators generated from Xilinx LogiCORE IP Floating-Point operators v7.0 [135] for double-precision arithmetic. We write VHDL controllers that manage the dataflow trigger logic and memory access logic to support

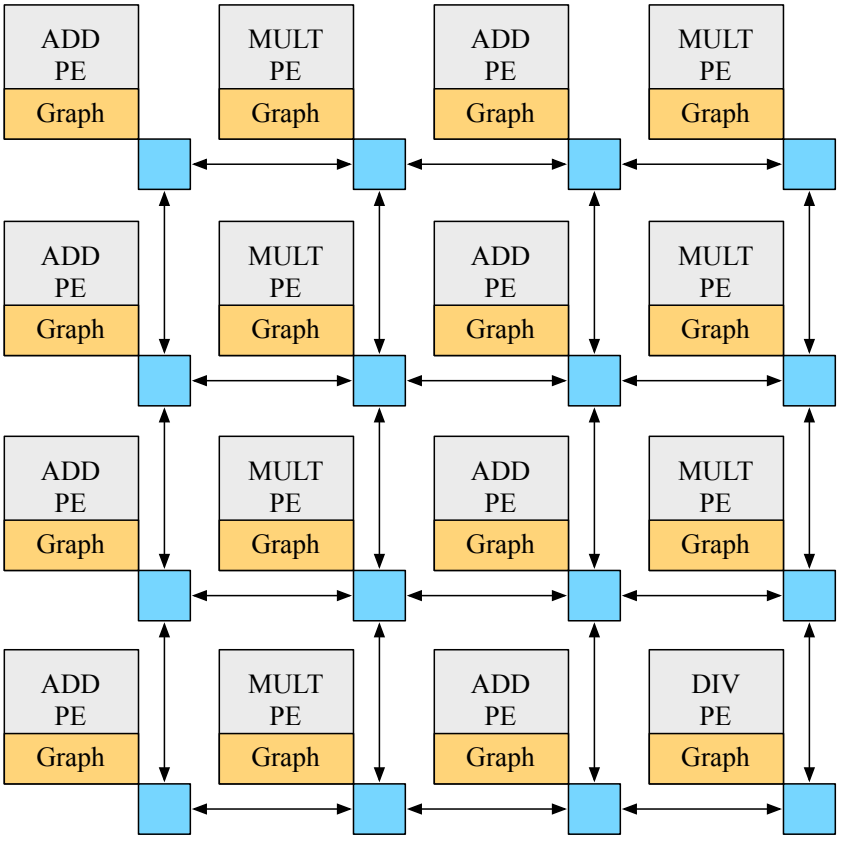


FIGURE 5.10: Heterogenous 2D overlay architecture used in this chapter. Note: We decouple Add/Multiply/Division PE, due to high DSP resource utilization to implement each arithmetic operator.

fully-pipelined high-throughput evaluation of the dataflow nodes. We employ a heterogeneous overlay topology as shown in Figure 5.10. We also developed our own VHDL NoC switch designs that implemented Dimension Ordered Routing (DOR) to support deadlock-free message delivery on the mesh NoC. We pipelined our design to achieve a clock frequency of 250 MHz and calibrate the NoC pipelines to a depth of 2 to meet this timing requirement. The PEs in the overlay are similar to the PE-Baseline in Chapter 3 – *i.e.* they are FIFO-based processors that evaluate active nodes in arrival-order. We compiled FPGA hardware using Xilinx Vivado 2013.4 CAD tool running on a 64b Ubuntu Linux platform. We use Xilinx XPower to estimate power consumption of the FPGA platform at a 15% activity rate on the FFs and LUTs. On the chosen FPGA platform, we are able to accommodate a 12x12 NoC design (*i.e.* 144 PEs) and report a power utilization of 30–40W. For simplicity, we restrict our NoC topology sizes during design space exploration to use square configurations (*e.g.* 1x1, 2x2, 3x3, ...). We show the resource utilization and pipeline depths of the various building blocks of our system

in Table 5.2. For our usage scenario, we need to generate the FPGA bitstream for the dataflow architecture only once across all input matrices. Each sparse matrix will have a unique non-zero structure and a resulting unique dataflow graph that is encoded as a memory image. These images are the only variable component of the hardware design and are loaded once at the start of the factorization. There is no need to reprogram the entire FPGA for each input matrix.

TABLE 5.2: Hardware Resource Utilization of the FPGA design

Block	**Area (LUTs)**	**DSPs**	**Latency**
Multiply PE	1107	11	10
Addition PE	1875	0	8
Division PE	5450	0	55
Switch Box	2254	0	5

5.6.2 Software Setup

We develop a dataflow compiler (pre-processor) that implements all the software optimizations described in Sections 5.3, 5.4, 5.5 as a series of *passes*. We interface the KLU sparse matrix solver with our compilation framework to enable capturing the dataflow graphs as inputs to our flow. The compiler optimizes the graph in multiple stages, applying the various optimization passes in sequence. The recursive substitution optimization is applied first to expose parallelism followed by reassociation (trivial or Huffman-inspired) that only affect the graph node fanins. Once the fanin trees are constructed, we decompose the fanout nodes next. We partition and place our dataflow graphs with MLPart-5.2.14 [92] to allow high-quality distribution of graphs in a manner the exploits communication locality. These pre-processing optimizations are one-time compilation transformations that are done in the same spirit as AMD [6] (approximate minimum degree) re-ordering of the matrices.

5.6.3 Experiments

For our performance evaluation, we compare runtime of the Xilinx Virtex-6 LX760 (40 nm) against a software implementation of KLU solver running on the Intel

Xeon 2407 CPU (32 nm) as our baseline. We report speedups against this baseline, as well as separately evaluate speedups for the various dataflow optimization passes applied cumulatively. We measured the peak power utilization of the Intel Xeon 2407 system at around ≈150W using the Energenie Power meter, which is roughly 3–4× higher than the FPGA platform. We develop a cycle-accurate simulator of our hardware design to enable us to prototype and verify our design performance much more quickly and efficiently across a large design space of tuneable parameters. This is essential to rapidly explore various design configurations as the FPGA CAD toolflow takes hours of compile time for even the smallest design change. In our experiments, we routinely observed that a substitution depth of 4 produces dataflow graphs with a desirable work-parallelism balance. When doing fanout decomposition experiments, we observed that the fanout threshold (f_t) of 16 and a decomposed tree arity (f_t) of 4 yields best results.

We use a range of sparse matrix benchmarks from the UFL Sparse Matrix repository [29] extracted from circuit simulations. We represent the graph properties such as number of floating-point operations (nodes), dependencies (edges), critical path latencies (depth of the graph) and matrix sparsity in Table 5.3. The benchmark sizes range from thousands of nodes up to millions of nodes with similar variations in critical latencies. These matrix factorizations from the circuit simulation domain can run for hours, if not days, due to the thousands/millions of SPICE iterations to reach convergence. The software pre-processing time imposed by our dataflow optimizations at the start of the run are roughly the cost of a few iterations and are easily amortized across the millions of iterative factorizations.

Table 5.3: Benchmark Properties

Benchmark	**Sparsity**	**Nodes**	**Edges**	**Critical Latency**
s526n	0.2%	544k	669k	27k
s526	0.2%	550k	674k	27k
s832	0.2%	938k	1.2m	47k
s953	0.2%	4.0m	5.3m	76k
s1196	0.1%	5.9m	7.5m	142k
bomhof1	0.5%	6.2m	8.8m	22k
bomhof2	0.1%	6.8m	9.4m	48k
simucad	0.3%	12.8m	17.5m	77k
s1238	0.1%	12.8m	16.8m	200k

5.7 Results

In this section, we quantify runtime performance benefits of our hardware-accelerated sparse matrix solver and explain the underlying effects that contribute to speedup. We also consider what-if design scenarios that can help extend the speedups beyond the reported speedups. Finally, we analyze the error properties of the $\vec{b} - A\vec{x}$ residuals.

5.7.1 Notation

To simplify the presentation of results, we use the following terms to refer to different optimization passes:

Dataflow Unrolling : This is the simplest form of optimization, where the computation is unrolled to be expressed as a dataflow graph. Note that no dataflow graph restructuring or further graph-related optimizations have been carried out in this step. This strategy has been introduced before in previous works involving token dataflow architectures (*e.g.* [69]), and is capable of delivering significant speedups on its own.

Substitution + Reassociation : This is the recursive substitution and naïve reassociation as detailed in Section 5.3 above.

Criticality-Aware Repacking : We bundle the two graph criticality-related optimizations discussed in Sections 5.4 (Huffman-inspired repacking) and 5.5 (fanout decomposition) as a one-off graph repacking optimization pass.

5.7.2 Speedups over CPU

Question: How does performance scale as each optimization pass is applied successively one at a time on top of each other?

We summarize the speedup contributions from the different optimizations across our benchmark set in Figure 5.11, where each optimization is applied in order on top of each other:

- Optimization pass 0 : Dataflow Unrolling

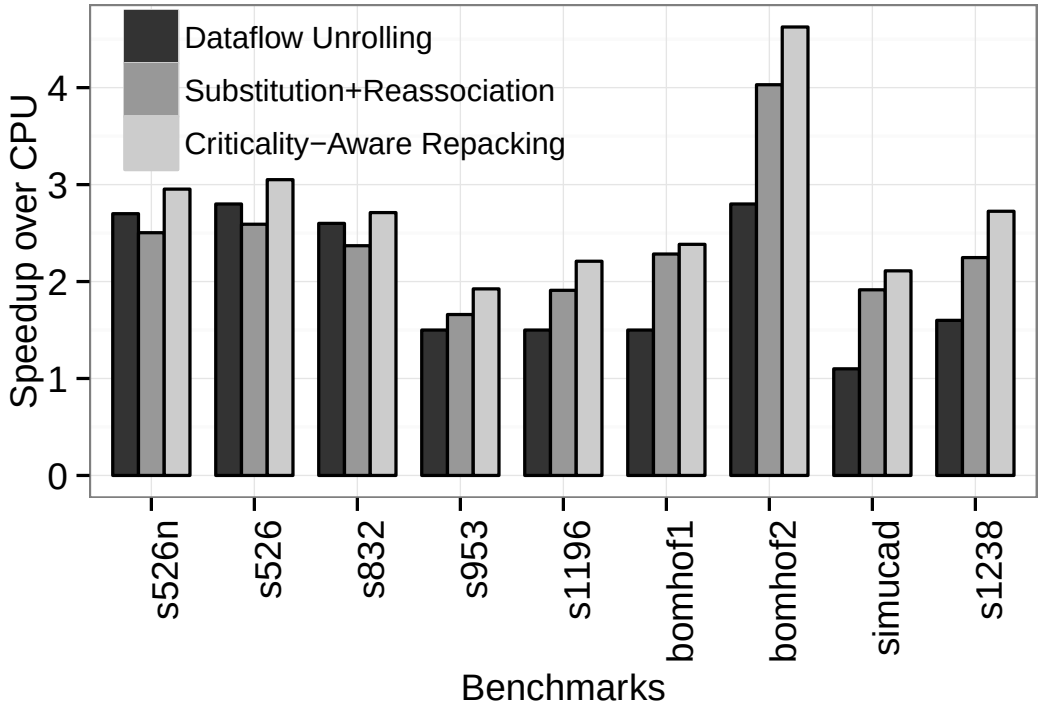


FIGURE 5.11: Overall speedup when different optimizations were applied successively on top of each other. Comparison is against a sequential Intel Xeon 2407 CPU implementation and a 144 PE LX760 FPGA implementation across various matrices.

- Optimization pass 1 : Dataflow Unrolling + Recursive Substitution + Trivial/Naïve Reassociation
- Optimization pass 2 : Dataflow Unrolling + Recursive Substitution + Trivial/Naïve Reassociation + Criticality-Aware Repacking

We observe an overall speedup of 1.9–4.6× over the sparse matrix KLU solver CPU implementation when all dataflow optimizations are activated. The different benchmarks report differing speedups due to a variety of factors such as distribution of chain lengths, sparsity of the bottleneck columns and connectivity in the graph.

The large benchmarks report $> 2\times$ overall speedups when considering all optimizations. Of particular interest is the `simucad` benchmark which offered practically no speedups with simple dataflow unrolling. This was due to the increased density of non-zeroes in the last few columns of L and U resulting in excessively sequential evaluation. We were able to overcome this sequential operation using the substitution, reassociation and repacking optimization to deliver $> 2\times$ speedup for this

stubborn benchmark. The `bomhof2` benchmark offers the highest speedups of 4.6× due to the sparsity structure of the underling matrix. However, for the smallest 3 benchmarks, we observe a slowdown in performance after performing a substitution + reassociation transformation but recovered speedups after applying all transformations (see `s526`, `s526n`, and `s832` in Figure 5.11). These small benchmark graphs have a relatively shorter critical path latency, which suffer longer communication delays (See Section 5.7.6 for ideas for alleviating this limit for small matrices).

Answer: For sufficiently large benchmarks, each optimization pass adds cumulative speedup over the baseline CPU implementation to deliver peak improvements from 1.9–4.6×. For smaller benchmarks, creating extra redundant work via substitution and reassociation does not prove worthwhile, but the criticality-aware repacking optimization pass helps to recover some of the lost performance.

5.7.3 Resource Scaling

*Question: How does performance scale as number of processors (*i.e. *dedicated resources) with different optimization strategies?*

In Figure 5.12, we highlight the performance scaling trends for the `bomhof2` matrix as we increase the number of PEs in our design. Our dataflow optimizations are able to produce a better work-parallelism tradeoff at larger system sizes > 10 PEs, which delivers a desired speedup scaling trend that can continue down for future systems. We discussed the substitution phenomenon that causes this effect earlier in Section 5.3.2 and Figure 5.5. While the FPGA selected limits largest system size to 144 PEs, we note that our parallelism enhancing transformations are continuing to scale even up to the largest system size. Modern FPGAs are 2–3× larger and can easily accomodate bigger 2D NoCs delivering additional improvements beyond those reported in this chapter.

Answer: At small system sizes, dataflow unrolling delivers the best benefits, but as system sizes increase, it becomes increasingly beneficial to apply higher-order dataflow graph optimizations such as recursive substitution, reassociation, and criticality-aware repacking.

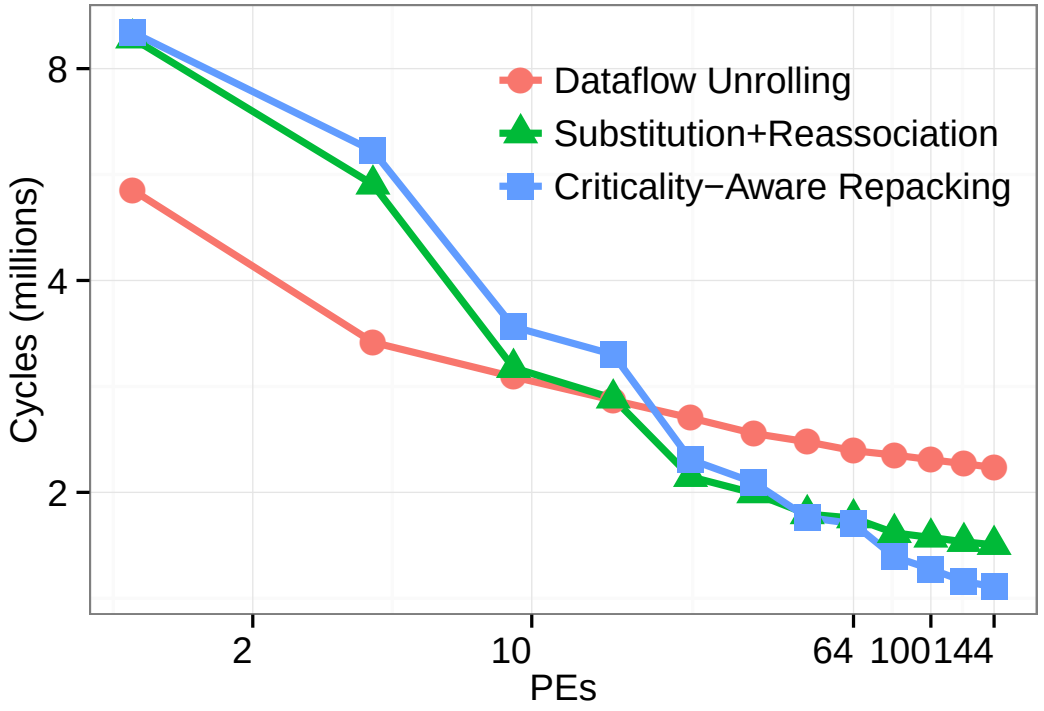


FIGURE 5.12: Performance Scaling trends for `bomhof2` benchmark. Work-Parallelism tradeoffs are visible at crossover systems size of ≈10 PEs.

5.7.4 Empirical Error Analysis

Question: Are these optimizations numerically safe?

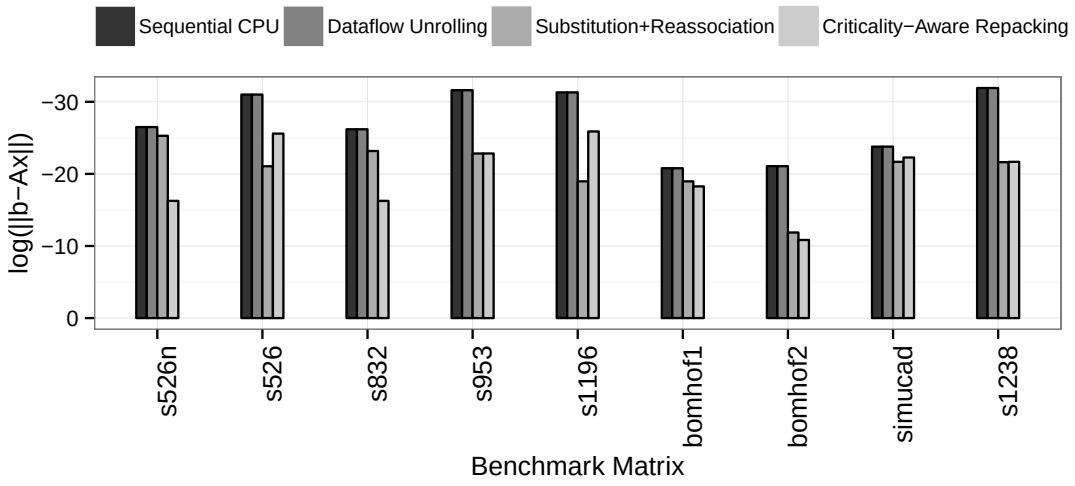


FIGURE 5.13: Impact of parallelizing dataflow optimizations on the residuals of $\vec{b} - A\vec{x}$ of the factored matrix under different optimization groups

We also compute the residual $\vec{b} - A\vec{x}$ of our transformed graphs and compare with the residual of the original sequential evaluation as shown in Figure 5.13. For this experiment, we obtain the $\vec{b}$ from an instance of the factorization evaluation within the SPICE simulation. We observe changes to the resulting residuals as large as 10^{-10} (*e.g.* `bomhof2`) or $< 10^{-20}$ (*e.g.* `simucad`) when applying all optimizations. For many numericallly-sensitive applications such as SPICE, these residues are still small enough to deliver satisfactory convergence. For cases where changes to residual error (and numerical order) are not tolerable, the user can restrict our optimizations to only support dataflow unrolling and still recover ≈2× speedup.

Answer: In the context of SPICE benchmarks, yes, they are. Nevertheless, the user can control the depth of dataflow unrolling and recursive substitution to control this effect, if tighter bounds are required.

5.7.5 Floating-Point Efficiency

Question: What peak floating-point efficiency can be delivered on these benchmarks with these varying software optimizations?

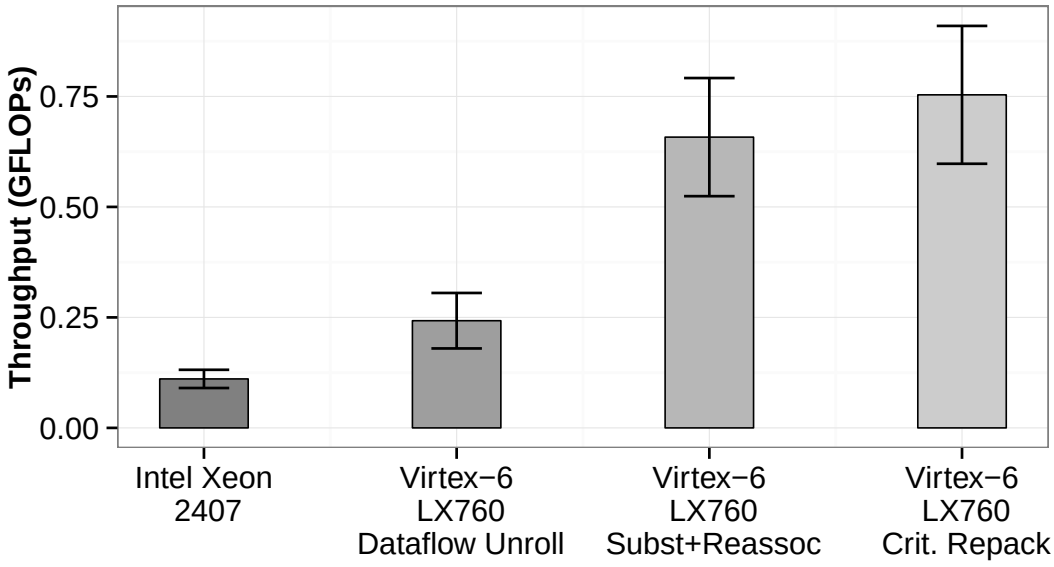


FIGURE 5.14: GFLOPs utilization of different optimizations implemented on CPU and various FPGAs. Error bars represent range of measured GFLOPs across all benchmarks.

Sparse problems are slow and difficult to accelerate due to their irregular memory access patterns. This is depicted in Figure 5.14 for the `bomhof2` benchmark, where

we found that the Intel Xeon 2407 CPU is able to achieve only 0.06 GFLOPs out of a theoretical sequential peak of 8.8 GFLOPs (assuming 2.2 GHz operation × 2-lane double-precision SSE instruction × FMA – 1 Mult + 1 Add × 1 core). Using FPGA accelerators, we are able to push the throughput up to 0.7 GFLOPs, out of a theoretical peak of 36 GFLOPs. This beats the low utilization of the CPU by 11×. It is worth noting that the speedup reported in Figure 5.11 for `bomhof2` is only 4.6× even though the GFLOPs rate is higher. This is due to the extra floating-point operations generated from the work-parallelism tradeoffs of the dataflow optimizations. Ultimately, the floating-point efficiency, even on the FPGA, is poor. A combination of time spent in the communication network, and the initial loading time of the constants contributes to this low utilization. In the rest of this section, we discuss extensions that could push speedups further.

Answer: The FPGA implementation is able to deliver an order of magnitude better GFLOP/s efficiency over an Intel Xeon CPU. This is, however, still far from the achievable peak, and there is still plenty of room at the bottom to optimize performance for these sparse benchmarks. DaCO on Arria 10, as seen in Chapter 3 takes this a step further by delivering up to 4.0 GFLOP/s (≈6× improvement over reported figures in this chapter).

5.7.6 Case for Homogeneous Design

Question: Is there any way to address the performance gap for smaller matrices?

For the smaller benchmarks – `s526`, `s526n` & `s832` – shown in Figure 5.11, we observed a disappointing slowdown in performance when considering recursive substitution and reassociation optimizations. We hypothesize that smaller graphs would prefer more tightly-coupled processors on smaller systems, such that the communication penalty created from substitution is mitigated. We support this hypothesis with an experiment using a different *homogeneous* NoC design that consists of PEs that contain both addition and multiplication hardware primitives. As expected, this comes at a cost of reduced system size. On the same Virtex-6 LX760 FPGA, we can accommodate an 8x8 homogeneous PE NoC design instead of the 12x12 heterogeneous system. We show the recovered and improved performance of the same 3 benchmarks in Figure 5.15 when implemented on the homogeneous design. This homogeneous design, however, limits achievable speedups for the larger benchmarks (not shown) and should only be used for smaller matrices. Current

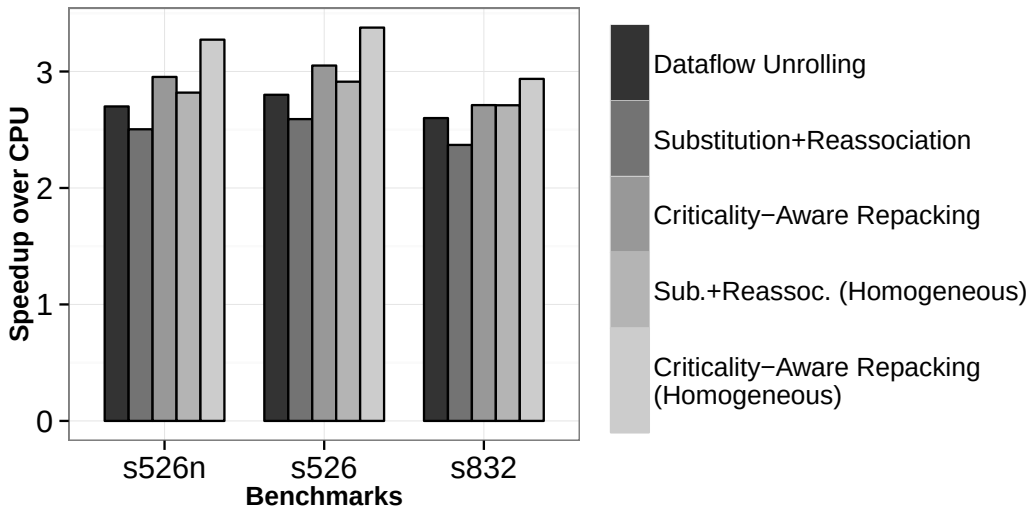


FIGURE 5.15: Performance recovery for small matrix benchmarks when using homogeneous designs with fused MAC units

preliminary results from our experiments suggest graphs with more than a million nodes should be evaluated on heterogeneous designs.

Answer: Smaller matrices, due to shorter critical paths, prefer tightly-coupled overlay architectures to minimize communication delays. By using a homogeneous PE overlay architecture, we can ensure that these small benchmarks do not lose out on any performance gains from these software optimizations.

5.7.7 Case for Selective Optimization

Question: Can we fine-tune the granularity at which each of these optimization passes are applied? Does that give improved performance?

Our current implementation of optimization applies the parallelization transformations uniformly over the entire dataflow graph. We hypothesize that there will be portions within the graph that have short operation chains and small sized subgraphs that will not benefit from our optimization at all. To evaluate this hypothesis, we developed a heuristic to only apply these transformation to subtrees that have chains exceeding a certain threshold. As shown in Figure 5.16, we can enhance the performance of the `bomhof2` matrix by an additional 8% at 144 PEs – some benchmarks produce dataflow graphs with greater degree of variance in parallelization opportunities, thus allowing them to benefit more from such a strategy. An example is the `bomhof1` benchmark, whose performance can be

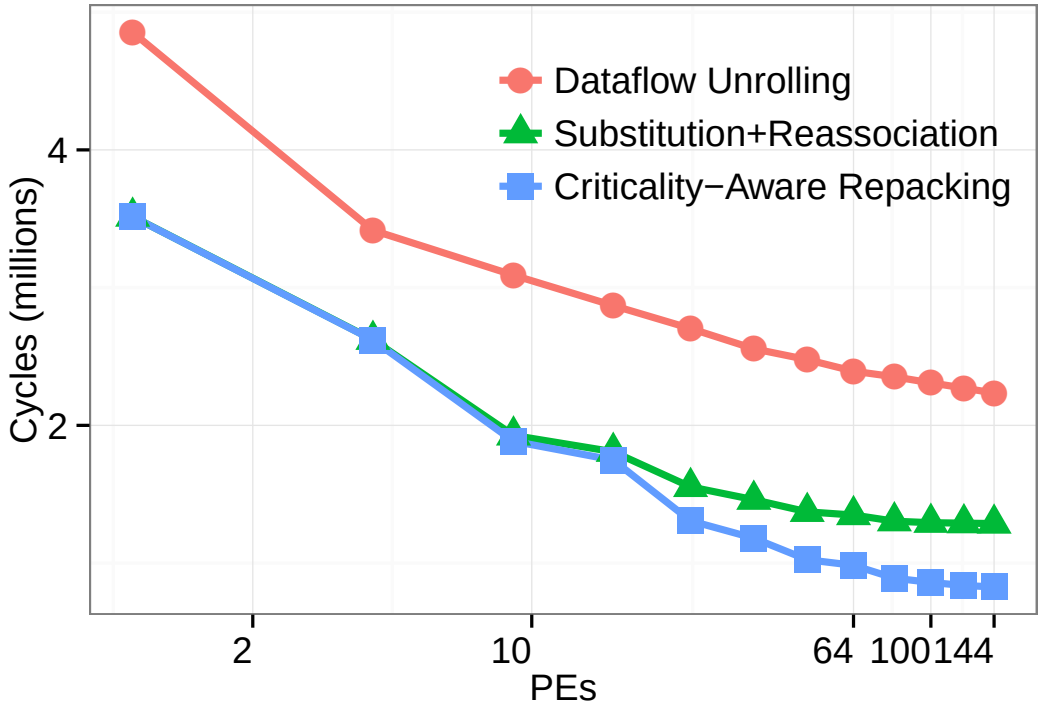


FIGURE 5.16: Effect of Selective optimization on the `bomhof2` benchmark PE scaling trends

scaled by an additional 36% when evaluated with an optimization selection strategy. We also observe a uniformly superior scaling graph that is always better than the design with pure dataflow unrolling. This shows how we can overcome the work-parallelism bottleneck at even smaller system sizes. Presently, we select the threshold for enabling parallelizing dataflow transformations using trial and error, and seek to develop automated machine learning strategies to pick the correct value for the threshold as part of future work.

Answer: Yes, we can, and it is possible to deliver even better performance (8–36% on top of existing reports). However, the methodology behind fine-grained optimization enable/disable is not clear and is difficult to generalize. Potential future work here could leverage the power of neural networks to identify these hidden features.

5.7.8 Related Work

The concept of exploiting the dataflow *task graph* representation of LU factorization is nothing new. In a prescient early work [56], the authors show optimal

Method	Description	Processor	Speedup
Optimal Triangulation [56]	Generate triangulation graph from sparse matrix	Sequential CPUs	N/A
KLU [30]	Static analysis to minimize fillin, no BLAS	Sequential CPUs	N/A
NICSLU [21]	Multi-core parallelization of the KLU solver	Multicore CPUs	2.1–8.6×
GPU-LU [138]	GPU-acceleration of the KLU solver	Tesla K40 GPU	0.5–47.2×
Distributed Memory [134]	Parallel column computation, dynamic analysis	FPGAs	0.4–12.7×
Streaming [62]	Power flow systems, structured sparse	FPGAs	2–10×
Token Dataflow [69]	Dataflow design of the KLU solver	FPGAs	0.6–13×

TABLE 5.4: Related Work

parallelization of LU factorization is possible by extracting the static *triangulation graph* (we call this a *dataflow graph*). In the decades that followed, vector computers and bespoke accelerators for LU decomposition were proposed but they never caught on due the rise of the Intel microprocessors. In this section, we review some of the more recent approaches, revisiting this problem on modern parallel accelerator hardware. Table 5.4 summarizes key characteristics of these related works.

The KLU solver [30] is an improvement over the SuperLU solver[75] and is customized for circuit simulation matrix factorization on sequential CPUs. KLU is able to reduce matrix fillin and also fix the positions of the non-zeros in the factors to optimize execution.

The NICSLU solver [21] enhances the KLU solver for use on shared memory multi-core machines. The multi-core implementation is able to exploit only coarse-grained parallelism at the granularity of column operations in the left-looking Gilbert-Peierls Algorithm used in KLU.

A GPU-driven implementation in [138], showcases similar speedup trends as in [21] but is also restricted to coarse-grained parallelism. In a bulk of the cases for matrices like the ones used in this chapter, the GPU actually *decelerates* the computation due to parallelism mismatch with the rigid data-parallel substrate. In contrast, our FPGA-based dataflow accelerator can exploit additional fine-grained parallelism within each column solve for these problems.

However, for a couple of isolated matrices, the GPU speedups are quite high (45–47×), but they come at the expense of 8–10× more power than the FPGA implementation. We can match those speedups if we partition our dataflow graph across multiple FPGAs at similar total power consumption as a single GPU.

Parallel hardware designs for sparse matrix factorization have been explored in recent past with an emphasis on power-efficient acceleration. [62] uses FPGAs to accelerate LU decomposition but their work is application specific only to symmetric power-flow matrices. [72] uses FPGAs to accelerate the block LU decomposition on dense matrices, while [119] reduces the memory footprint of unrolled dense LU decomposition. Our software optimizations focus on asymmetric and sparse matrices which are harder to accelerate than these designs. In [134], the authors bundle several columns (collectively called a *panel*) using a left-looking technique to evaluate sparse matrix factorization in parallel on an FPGA network. [69] uses complete loop unrolling for sparse factorization to exploit more fine-grained dataflow parallelism with packet-switching networks to deliver highest reported FPGA speedups for sparse LU factorization. The work presented in this thesis expands on this approach by (1) proposing software pre-processing techniques for exposing additional parallelism, and (2) developing a high-performance token dataflow overlay coprocessor hardware that is well-suited for irregular parallelism.

5.8 Future Work

- As mentioned previously, the work presented here has been evaluated on an older design iteration that has been replaced with a much leaner and more performant overlay architecture in DaCO. Our goal in the future is to re-package and re-evaluate these dataflow-centric software optimizations on the newer architecture as soon as possible. We are confident that the software optimizations presented in this chapter would deliver promising results on DaCO.

- Partitioning strategies can have a significant impact on runtime performance. With the introduction of clustered-PE regions in DaCO, we would work on a topology-aware partitioner that is customized for DaCO.

- Machine learning has transformed the software ecosystem in the past few years. We hope to apply some of these machine learning techniques to further improve dataflow graph optimizations – examples could be in better partitioning and placement, or improved optimization selection based on graph characteristics. Machine learning could also help with identification of dataflow-friendly compute regions in an application, that can help the compiler make better scheduling decisions for dynamic HLS flows.

5.9 Conclusions

We show how to accelerate LU factorization by 1.9–4.6× while consuming 3–4× less power across a range of benchmark matrices when comparing a Xilinx Virtex-6 LX760 FPGA with an Intel Xeon 2407 CPU. The magnitude of speedups scales well with the matrix size. Almost half of the speedup comes purely from dataflow formulation of the problem while the rest is due to dataflow graph optimizations such as recursive substitution and criticality-aware repacking. We explore the work-parallelism tradeoffs inherent in our dataflow optimizations and show how these help make our design scale for 100s of parallel processors. Empirical evidence indicates minimal impact of rearranging the floating-point multiplication and addition operations on the $\vec{b} - A\vec{x}$ residual.

5.10 Publications

Subsets of this body of work has been published in the following peer-reviewed conference proceedings:

1. Siddhartha and Nachiket Kapre, *Breaking Sequential Dependencies in FPGA-based Sparse LU Factorization*, International Symposium on Field Programmable Custom Computing Machines, May 2014 *(Short Paper)*

2. Siddhartha and Nachiket Kapre, *Heterogeneous Dataflow Architectures for FPGA-based Sparse LU Factorization*, International Conference on Field Programmable Logic and Applications, September 2014 *(Poster)*

3. Siddhartha and Nachiket Kapre, *Fanout Decomposition Dataflow Optimizations for FPGA-based Sparse LU Factorization*, International Conference on Field-Programmable Technology, December 2014 *(Short Paper)*

4. Siddhartha and Nachiket Kapre, *FPGA Acceleration of Irregular Iterative Computations using Criticality-Aware Dataflow Optimizations*, 22nd IEEE Symposium on Field Programmable Gate Arrays, February 2015 *(Poster)*

Chapter 6

Conclusion

6.1 Final Contributions

This thesis introduces the Dataflow Overlay Coprocessor (DaCO) engine tuned for FPGAs. The work on DaCO is composed of three key bodies of research: (1) the soft processor design, (2) the communication framework design, and (3) the software backend design. We summarize the quantitative benefits observed through each of these works below:

1. **Soft Processor Design** : By introducing out-of-order criticality-aware scheduling inside the token dataflow soft processor, the DaCO soft processor delivers up to 2.5–2.8× improved performance over existing token dataflow architectures and server class CPUs (Intel Xeon E5-2680). The out-of-order scheduler is capable of scheduling on 1000s of active nodes. This is achieved through a lightweight hierarchical leading-ones detector circuit that adds a 70% logic overhead to the existing design, while, more importantly, freeing up precious block RAM resources to allow storage of larger dataflow graphs by up to 20% in the on-chip memory.

2. **Communication Framework** : The communication framework adopts a hierarchical topology where clustering groups of PEs together that are connected by local crossbars improves performance by up to 1.5× at a 15–40% logic resource overhead (for 2–4 cluster sizes). The packet-switching router

that connects each cluster is also augmented with priority-aware routing features, which we call Hoplite-Q*. When evaluated with mixed-priority multi-application workloads, Hoplite-Q* is capable of accelerating top-priority applications by 1.3–1.9× (1.5–3.9× improved worst-case latency), at a resource overhead of 3.8×. On dataflow workloads, Hoplite-Q* has limited impact when used in conjunction with the out-of-order PE-DaCO, but demonstrates potential for the future with larger benchmarks and dynamic dataflow execution model.

3. **Software Backend** : Dataflow-inspired software optimizations that rearrange the graph structure at compile time can deliver runtime speedups of 1.9–4.6× over CPU implementations when evaluated on LU factorization traces extracted from SPICE benchmarks. These optimizations deliver up to 11× better GFLOPs utilization when compared to the Intel Xeon 2407 CPU. Finally, these software optimizations have no noticeable side-effect on the numerical stability of the SPICE kernels, as the benchmarks are still able to achieve convergence. While the software backend was developed and tested on an older iteration of our token dataflow overlay, we are confident that the software optimizations proposed in this body of work would apply favorably to benchmarks evaluated on DaCO as well.

6.2 Lessons

Datapath occupancy : From our experiments, dataflow graphs are typically "top-heavy", where most of the parallelism is concentrated at the top of the dataflow graph, while the tail-end of the dataflow graph usually peters out into a long sequential tail. This results in varying levels of occupancy in the processor pipeline, which results in vast under-utilization of the long FIFO in existing token dataflow processors. Moreover, since the FIFO has to be scaled for the worst-case, the resource utilization efficiency is poor. Our criticality-aware scheduling strategy in DaCO corrects these inefficiences and opens up new optimization possibilities for future designs. For example, age-based reordering at runtime could further improve scheduling performance, similar to the observation when moving from Hoplite-Q to Hoplite-Q* in Chapter 4. The long sequential tail also motivates the design of a tightly-coupled datapath, where sequential chains of instructions are mapped to a single processor and executed in a back-to-back fashion. One

strategy to address this is to adopt the *codelet* model (see Future Work section below).

Dataflow Injection Rates : Average injection rates of token dataflow benchmarks was observed to be fairly low since only a subset of edges are active in each cycle. This motivates the move from a bidirectional buffered mesh router to a much more lightweight deflection router like Hoplite, which can support the observed dataflow injection rates ($\leq$10%). Our results with DaCO demonstrated the benefits of this approach, as both scalability and throughput performance of the benchmarks improved. The low injection rates also, however, allude to limitations of the packet generator: while the packet consumer and scheduler pipeline stages can be theoretically kept busy in all cycles, there is a 4-cycle setup cost associated with each node inside the packet generator due to node/edge memory read latencies. We support back-to-back edge injection from the active node, but unfortuantely, this is not sufficient to keep the communication network busy. In the future, we would look into supporting back-to-back *node* and *edge* evaluation inside the packet generator to raise injection rates. This is a performance-area tradeoff, where we issue and store speculative reads that allow back-to-back packet injection across both nodes and edges.

Criticality-awareness : Throughout this thesis, we have emphasized on criticality-aware optimizations at all design levels, which has been missing in existing FPGA-based token dataflow overlays. In the processor, criticality-aware scheduling not only improves throughput performance by 0.9–2.6×, but also frees up on-chip BRAM resources for better resource efficiency. In the network-on-chip communication framework, Hoplite-Q* currently has limited impact on dataflow problems, with 1–10% improvement in performance depending on the type of PE used (PE-Baseline or PE-DaCO). When considering multi-application workloads, Hoplite-Q* shows capability of accelerating top-priority (*i.e.* most critical) application throughput by up to 1.9×, which could be suitable for a multi-tenant datacenter-type of shared environment. Finally, our software compiler also incorporates criticality-aware modifications that showcase >2× improvements over the baseline token dataflow overlay when evaluated on SPICE circuit simulation benchmarks. These results motivate us to continue exploring criticality-aware evaluation for token dataflow problems, and in the future, we hope to continue refining these ideas and quantify their effects on a wider set of benchmarks.

6.3 Future Work

We believe there is still plenty of room at the bottom to innovate and push the performance of DaCO even further. Here, we list the top three research vectors that we hope to undertake in the near future

1. **Coprocessor coupling** : While SoCs like Intel 6138P or Xilinx Zynq platforms bring FPGA-based coprocessors closer to the microprocesor, this is not the only solution to designing coprocessor-dominated architectures. For example, a small RISC-V [129] processor assigned to *each cluster* on-chip could tighten the coupling even further. In [90], the authors demonstrated the feasibility of a tightly-coupled von Neumann core and explicit dataflow core, and we hope to emulate and extend this branch of research with DaCO.

2. **Codelet model** : The work in this thesis so far has largely focused on fine-grained dataflow acceleration, where each node/vertex in the graph is an arithmetic instruction. While this exposes raw parallelism in the algorithm, it does, however, introduce runtime overheads (*i.e.* large number of nodes and edges) and optimization problems (*e.g.* graph partitioning and placement for distributed architectures). Instead, to derive good performance and alleviate resource impact, one strategy could be to cluster groups of highly-localized computations together into a *codelet* [142]. Codelets, for example, in the case of convolutional neural networks, could represent a single convolution, which is a multi-instruction filter operation that operates on a localized pixel region. Codelets can be likened to the Explicit Token Store [93] model where activation frames bound groups of instructions that evaluate in dataflow-like order. We aim to explore the codelet granularity and extend our software backend and our runtime ISA to support the generation and execution of these codelet instructions.

3. **Benchmarks** : With the above two enhancements, we believe DaCO can be suitable for other application domains that are riddled with irregular forms of parallelism and synchronization overheads. Some promising fields include sparse convolutional neural networks, graph convolutional networks, and molecular dynamics. We hope to demonstrate DaCO's feasibility in these application domains in the near future.

Bibliography

[1] Kaveh Aasaraai and Andreas Moshovos. "Design space exploration of instruction schedulers for out-of-order soft processors". In: *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE. 2010, pp. 385–388.

[2] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente, and J. Á. Gregorio. "TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers". In: *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. May 2012, pp. 99–106. DOI: 10.1109/NOCS.2012.19.

[3] Mohamed S Abdelfattah and Vaughn Betz. "Networks-on-chip for FPGAs: Hard, soft or mixed?" In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.3 (2014), p. 20.

[4] Ankur Agarwal, Cyril Iskander, and Ravi Shankar. "Survey of Network on Chip (NoC) architectures and contributions". In: *Journal of engineering, Computing and Architecture* 3.1 (2009), pp. 21–27.

[5] EC Amazon. *F1 Instance*.

[6] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. "Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm". In: *ACM Trans. Math. Softw.* 30.3 (2004), pp. 381–388.

[7] Federico Angiolini, Paolo Meloni, Salvatore Carta, Luca Benini, and Luigi Raffo. "Contrasting a NoC and a traditional interconnect fabric with layout awareness". In: *Proceedings of the Design Automation & Test in Europe Conference*. Vol. 1. IEEE. 2006, pp. 1–6.

[8] SA Arteris. *From Bus and Crossbar to Network-on-Chip*. 2009.

[9] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. "Tile64-processor: A 64-core soc with mesh interconnect". In: *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*. IEEE. 2008, pp. 88–598.

[10] Luca Benini and Giovanni De Micheli. "Networks on Chips: A New SoC Paradigm". In: *Computer* 35.1 (Jan. 2002), pp. 70–78. ISSN: 0018-9162. DOI: 10.1109/2.976921. URL: https://doi.org/10.1109/2.976921.

[11] Tobias Bjerregaard and Shankar Mahadevan. "A survey of research and practices of network-on-chip". In: *ACM Computing Surveys (CSUR)* 38.1 (2006), p. 1.

[12] S. Borkar. "Achieving energy efficiency by HW/SW co-design". In: *2013 Third Berkeley Symposium on Energy Efficient Electronic Systems (E3S)*. Oct. 2013, pp. 1–1. DOI: 10.1109/E3S.2013.6705856.

[13] Alexander Brant and Guy GF Lemieux. "ZUMA: An open FPGA overlay architecture". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 93–96.

[14] Franc Brglez, David Bryan, and Krzysztof Kozminski. "Combinational profiles of sequential benchmark circuits". In: *Circuits and Systems, 1989., IEEE International Symposium on*. IEEE. 1989, pp. 1929–1934.

[15] Doug Burger, Stephen W Keckler, Kathryn S McKinley, Mike Dahlin, Lizy K John, Calvin Lin, Charles R Moore, James Burrill, Robert G McDonald, and William Yoder. "Scaling to the End of Silicon with EDGE Architectures". In: *Computer* 37.7 (2004), pp. 44–55.

[16] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: high-level synthesis for FPGA-based processor/accelerator systems". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, pp. 33–36.

[17] Juan-Antonio Carballo, Wei-Ting Jonas Chan, Paolo A Gargini, Andrew B Kahng, and Siddhartha Nath. "ITRS 2.0: Toward a re-framing of the Semiconductor Technology Roadmap". In: *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE. 2014, pp. 139–146.

[18] Umit V Catalyürek and Cevdet Aykanat. "PaToH: a multilevel hypergraph partitioning tool, version 3.0". In: *Bilkent University, Department of Computer Engineering, Ankara* 6533 (1999).

[19] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A Cloud-Scale Acceleration Architecture". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press. 2016, p. 7.

[20] Hui Yan Cheah, Suhaib A Fahmy, and Douglas L Maskell. "iDEA: A DSP block based FPGA soft processor". In: *Field-Programmable Technology (FPT), 2012 International Conference on*. IEEE. 2012, pp. 151–158.

[21] Xiaoming Chen, Yu Wang, and Huazhong Yang. "NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 32.2 (2013), pp. 261–274.

[22] Christopher H Chou, Aaron Severance, Alex D Brant, Zhiduo Liu, Saurabh Sant, and Guy GF Lemieux. "VEGAS: Soft vector processor with scratchpad memory". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, pp. 15–24.

[23] James Coole and Greg Stitt. "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing". In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2010, pp. 13–22.

[24] David E Culler. "Dataflow architectures". In: *Annual review of computer science* 1.1 (1986), pp. 225–253.

[25] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[26] R. Das, O. Mutlu, T. Moscibroda, and C. Das. "Aergia: A Network-on-Chip Exploiting Packet Latency Slack". In: *IEEE Micro* 31.1 (Jan. 2011), pp. 29–41. ISSN: 0272-1732. DOI: 10.1109/MM.2010.98.

[27] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R Das. "Application-aware prioritization mechanisms for on-chip networks". In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE. 2009, pp. 280–291.

[28] Jeffrey A Davis, Raguraman Venkatesan, Alain Kaloyeros, Michael Beylansky, Shukri J Souri, Kaustav Banerjee, Krishna C Saraswat, Arifur Rahman, Rafael Reif, and James D Meindl. "Interconnect limits on gigascale integration (GSI) in the 21st century". In: *Proceedings of the IEEE* 89.3 (2001), pp. 305–324.

[29] T Davis. "The University of Florida Sparse Matrix Collection". In: *(unpublished) ACM Transactions on Mathematical Software* (2007).

[30] Timothy A Davis and Ekanathan Palamadai Natarajan. "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems". In: *ACM Transactions on Mathematical Software* 37.3 (2010), 36:1–36:17.

[31] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *Solid-State Circuits, IEEE Journal of* 9.5 (Oct. 1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511.

[32] Jack B Dennis and David P Misunas. "A preliminary architecture for a basic data-flow processor". In: *ACM SIGARCH Computer Architecture News* 3.4 (1975), pp. 126–132.

[33] Xuan Khanh Do, Stephane Louise, and Albert Cohen. "Comparing the StreamIt and ΣC languages for manycore processors". In: *Int. l Workshop on Data-Flow Models (DFM) for Extreme Scale Computing. IEEE*. 2014.

[34] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits". In: *Proceedings of the IEEE* 98.2 (2010), pp. 253–266.

[35] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling". In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 365–376.

[36] Hadi Esmaeilzadeh, Emily Blem, Renée St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Power challenges may end the multicore era". In: *Communications of the ACM* 56.2 (2013), pp. 93–102.

[37] Chris Fallin, Chris Craik, and Onur Mutlu. "CHIPPER: A low-complexity bufferless deflection router". In: *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE. 2011, pp. 144–155.

[38] Chris Fallin, Greg Nazario, Xiangyao Yu, Kevin Chang, Rachata Ausavarungnirun, and Onur Mutlu. "MinBD: Minimally-buffered deflection routing for energy-efficient interconnect". In: *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*. IEEE. 2012, pp. 1–10.

[39] Joel H Ferziger and Milovan Peric. *Computational methods for fluid dynamics*. Springer Science & Business Media, 2012.

[40] David J Frank. "Power-constrained CMOS scaling limits". In: *IBM Journal of Research and Development* 46.2.3 (2002), pp. 235–244.

[41] David J Frank, Robert H Dennard, Edward Nowak, Paul M Solomon, Yuan Taur, and Hon-Sum Philip Wong. "Device scaling limits of Si MOSFETs and their application dependencies". In: *Proceedings of the IEEE* 89.3 (2001), pp. 259–288.

[42] Alexandros V Gerbessiotis and Leslie G Valiant. "Direct bulk-synchronous parallel algorithms". In: *Journal of parallel and distributed computing* 22.2 (1994), pp. 251–267.

[43] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael B Taylor, and Steven Swanson. "GreenDroid: A mobile application processor for a future of dark silicon". In: *2010 IEEE Hot Chips 22 Symposium (HCS)*. IEEE. 2010, pp. 1–39.

[44] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing". In: *IEEE Micro* 32.5 (Sept. 2012), pp. 38–51. ISSN: 0272-1732. DOI: 10.1109/MM.2012.51.

[45] Jan Gray. “GRVI Phalanx: A massively parallel RISC-V FPGA accelerator”. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2016, pp. 17–20.

[46] Jan Gray and Aaron Smith. “Towards an Area-Efficient Implementation of a High ILP EDGE Soft Processor”. In: *CoRR* abs/1803.06617 (2018). arXiv: 1803.06617. URL: http://arxiv.org/abs/1803.06617.

[47] Peter Greenhalgh. “Big. LITTLE Processing with ARM Cortex-A15 and Cortex-A7”. In: *ARM White paper* 17 (2011).

[48] John R Gurd, Chris C Kirkham, and Ian Watson. “The Manchester prototype dataflow computer”. In: *Communications of the ACM* 28.1 (1985), pp. 34–52.

[49] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[50] J. Hicks, D. Chiou, B.S. Ang, and Arvi. “Performance Studies of Id on the Monsoon Dataflow System”. In: *Journal of Parallel and Distributed Computing* 18.3 (1993), pp. 273–300. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1993.1065. URL: http://www.sciencedirect.com/science/article/pii/S0743731583710658.

[51] Clint Hilton and Brent Nelson. “PNoC: a flexible circuit-switched NoC for FPGA-based systems”. In: *IEE Proceedings-Computers and Digital Techniques* 153.3 (2006), pp. 181–188.

[52] H J Hoover, M M Klawe, and N J Pippenger. “Bounding Fan-out in Logical Networks”. In: *Journal of the ACM* 31.1 (Jan. 1984), pp. 13–18.

[53] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. “A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS”. In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE. 2010, pp. 108–109.

[54] Y. Huan and A. DeHon. "FPGA optimized packet-switched NoC using split and merge primitives". In: *2012 International Conference on Field-Programmable Technology*. Dec. 2012, pp. 47–52. DOI: 10.1109/FPT.2012.6412110.

[55] Yutian Huan and A DeHon. "FPGA optimized packet-switched NoC using split and merge primitives". In: *Field-Programmable Technology (FPT), 2012 International Conference on*. Dec. 2012, pp. 47–52.

[56] J Huang and O Wing. "Optimal parallel triangulation of a sparse matrix". In: *IEEE Transactions on Circuits and Systems* 26.9 (1979), pp. 726–732.

[57] D A Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101.

[58] Intel. *Intel FPGAs and Processors - Better Together*. 2018. URL: https://itpeernetwork.intel.com/intel-processors-fpga-better-together.

[59] *Intel shows Xeon Scalable GOld 6138P with Integrated FPGA, Shipping to Vendors*. https://www.anandtech.com/show/12773/intel-shows-xeon-scalable-gold-6138p-with-integrated-fpga-shipping-to-vendors. Accessed: 2018-06-20.

[60] Abhishek Kumar Jain. "Architecture Centric Coarse-Grained FPGA Overlays". PhD thesis. Nanyang Technological University, 2017.

[61] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. "Efficient Overlay architecture based on DSP blocks". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 25–28.

[62] Jeremy Johnson, Tim Chagnon, Petya Vachranukunkiet, Prawat Nagvajara, and Chika Nwankpa. "Sparse LU Decomposition using FPGA". In: *Intl. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*. 2008.

[63] Norman P Jouppi, Cliff Young, Nishant Patil, and David Patterson. "A Domain-Specific Architecture for Deep Neural Networks". In: *Communications of the ACM* 61.9 (2018), pp. 50–59.

[64] N. Kapre and J. Gray. "Hoplite: Building austere overlay NoCs for FPGAs". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7293956.

[65] N. Kapre and J. Gray. “Hoplite: Building austere overlay NoCs for FPGAs”. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7293956.

[66] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. “Packet Switched vs. Time Multiplexed FPGA Overlay Networks”. In: *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Apr. 2006, pp. 205–216. DOI: 10.1109/FCCM.2006.55.

[67] Nachiket Kapre. “Deflection-routed butterfly fat trees on FPGAs”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2017, pp. 1–8.

[68] Nachiket Kapre. “Implementing FPGA Overlay NoCs Using the Xilinx UltraScale Memory Cascades”. In: *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE. 2017, pp. 40–47.

[69] Nachiket Kapre and Andre DeHon. “Parallelizing Sparse Matrix Solve for SPICE circuit simulation using FPGAs”. In: *Field-Programmable Technology*. Jan. 2010.

[70] Nachiket Kapre and Siddhartha. “Limits of Statically-Scheduled Token Dataflow Processing”. In: *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*. IEEE. 2014, pp. 1–8.

[71] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. “An efficient FPGA overlay for portable custom instruction set extensions”. In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE. 2013, pp. 1–8.

[72] M Kumar Jaiswal and N Chandrachoodan. “FPGA-Based High-Performance and Scalable Block LU Decomposition Architecture”. In: *IEEE Transactions on Computers* 61.1 (2012), pp. 60–72.

[73] Charles Eric LaForest and J Gregory Steffan. “Efficient multi-ported memories for FPGAs”. In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2010, pp. 41–50.

[74] Jae W Lee, Man Cheuk Ng, and Krste Asanovic. "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks". In: *ACM SIGARCH Computer Architecture News*. Vol. 36. 3. IEEE Computer Society. 2008, pp. 89–100.

[75] Xiaoye S Li. "An Overview of SuperLU: Algorithms, Implementation, and User Interface". In: *ACM Transactions on Mathematical Software* 31.3 (Sept. 2005), pp. 302–325.

[76] Yuhai Li, Kuizhi Mei, Yuehu Liu, Nanning Zheng, and Yi Xu. "LDBR: Low-deflection bufferless router for cost-sensitive network-on-chip design". In: *Microprocessors and Microsystems* 38.7 (2014), pp. 669–680.

[77] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. "Sparse convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 806–814.

[78] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. "QuickDough: a rapid FPGA loop accelerator design framework using soft CGRA overlay". In: *2015 International Conference on Field Programmable Technology (FPT)*. IEEE. 2015, pp. 56–63.

[79] Zhonghai Lu, Mingchen Zhong, and Axel Jantsch. "Evaluation of on-chip networks using deflection routing". In: *Proceedings of the 16th ACM Great Lakes symposium on VLSI*. ACM. 2006, pp. 296–301.

[80] Roman L Lysecky, Kris Miller, Frank Vahid, and Kees A Vissers. "Firm-core virtual FPGA for just-in-time FPGA compilation". In: *FPGA*. 2005, p. 271.

[81] Roman Lysecky, Frank Vahid, and Sheldon X-D Tan. "Dynamic FPGA routing for just-in-time FPGA compilation". In: *Proceedings of the 41st annual Design Automation Conference*. ACM. 2004, pp. 954–959.

[82] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[83] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[84] G. E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. ISSN: 0018-9219. DOI: 10.1109/jproc.1998.658762. URL: http://dx.doi.org/10.1109/jproc.1998.658762.

[85] Thomas Moscibroda and Onur Mutlu. “A case for bufferless routing in on-chip networks”. In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 196–207.

[86] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. “A survey and evaluation of FPGA high-level synthesis tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604.

[87] II Nios. “Gen2 Processor Reference Guide”. In: *Altera Corporation (Reference Guide). Cited on page* 10 (2015), p. 108.

[88] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. “Stream-dataflow acceleration”. In: *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE. 2017, pp. 416–429.

[89] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. “Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models”. In: *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*. 2015.

[90] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. “Exploring the potential of heterogeneous von neumann/dataflow execution models”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 298–310.

[91] CUDA Nvidia. “Nvidia CUDA C Programming Guide”. In: *Nvidia Corporation* 120.18 (2011), p. 8.

[92] David A Papa and Igor L Markov. *Hypergraph Partitioning and Clustering*. 2007.

[93] G M Papadopoulos and D E Culler. “Monsoon: an explicit token-store architecture”. In: *Proceedings of the Annual International Symposium on Computer Architecture* 18.3a (1990), pp. 82–91.

[94] 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. Monterey, California, USA: ACM, 2012, pp. 37–46. ISBN: 978-1-4503-1155-7. DOI: 10.1145/2145694.2145703. URL: http://doi.acm.org/10.1145/2145694.2145703.

[95] Michael K Papamichael and James C Hoe. “CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs”. In: *the ACM/SIGDA international symposium*. New York, New York, USA: ACM Press, 2012, p. 37.

[96] Raphael Polig, Heiner Giefers, and Walter Stechele. “A soft-core processor array for relational operators”. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2015, pp. 17–24.

[97] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. “A reconfigurable fabric for accelerating large-scale datacenter services”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 13–24.

[98] *Pynq: Python Productivity for Zynq*. https://pynq.io. Accessed: 2018-06-20.

[99] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. “Computational sprinting”. In: *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE. 2012, pp. 1–12.

[100] RM Ramanathan. “Intel® Multi-Core Processors”. In: *Making the Move to Quad-Core and Beyond* (2006).

[101] *Reconfigure.io*. https://reconfigure.io. Accessed: 2018-03-30.

[102] Ling Ren, Xiaoming Chen, Yu Wang, Chenxi Zhang, and Huazhong Yang. “Sparse LU factorization for parallel circuit simulation on GPU”. In: *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. June 2012, pp. 1125–1130.

[103] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. "An architecture of a dataflow single chip processor". In: *ACM SIGARCH Computer Architecture News*. Vol. 17. 3. ACM. 1989, pp. 46–53.

[104] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W Keckler, Robert G McDonald, and Charles R Moore. "TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 1.1 (2004), pp. 62–93.

[105] Aaron Severance and Guy Lemieux. "VENICE: A compact vector processor for FPGA applications". In: *2012 International Conference on Field-Programmable Technology*. IEEE. 2012, pp. 261–268.

[106] Aaron Severance and Guy GF Lemieux. "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor". In: *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press. 2013, p. 6.

[107] Devendra Kumar Sharma, BK Kaushik, and RK Sharma. "VLSI interconnects and their testing: prospects and challenges ahead". In: *Journal of Engineering, Design and Technology* 9.1 (2011), pp. 63–84.

[108] David E. Shaw, J. P. Grossman, Joseph A. Bank, Brannon Batson, J. Adam Butts, Jack C. Chao, Martin M. Deneroff, Ron O. Dror, Amos Even, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Brian Greskamp, C. Richard Ho, Douglas J. Ierardi, Lev Iserovich, Jeffrey S. Kuskin, Richard H. Larson, Timothy Layman, Li-Siang Lee, Adam K. Lerer, Chester Li, Daniel Killebrew, Kenneth M. Mackenzie, Shark Yeuk-Hai Mok, Mark A. Moraes, Rolf Mueller, Lawrence J. Nociolo, Jon L. Peticolas, Terry Quan, Daniel Ramot, John K. Salmon, Daniele P. Scarpazza, U. Ben Schafer, Naseer Siddique, Christopher W. Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley C. Wang, and Cliff Young. "Anton 2: Raising the Bar for Performance and Programmability in a Special-purpose Molecular Dynamics Supercomputer". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 41–53. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.9.

[109] Siddhartha and Nachiket Kapre. "eBSP: Managing NoC traffic for BSP workloads on the 16-core Adapteva Epiphany-III processor". In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2017, pp. 73–78.

[110] Siddhartha and Nachiket Kapre. "Heterogeneous Dataflow Architectures for FPGA-based Sparse LU Factorization". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–4.

[111] Siddhartha and Nachiket Kapre. "Hoplite-Q: Priority-Aware Routing in FPGA Overlay NoCs". In: *2018 26th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. May 2018.

[112] Wilson Snyder. "Verilator and SystemPerl". In: *North American SystemC Users' Group, Design Automation Conference*. 2004.

[113] Hayden Kwok-Hay So and Cheng Liu. "FPGA overlays". In: *FPGAs for Software Programmers*. Springer, 2016, pp. 285–305.

[114] Richard M Stallman. *Using GCC: the GNU compiler collection reference manual*. Gnu Press Boston, 2003.

[115] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J Eggers. "The wavescalar architecture". In: *ACM Transactions on Computer Systems (TOCS)* 25.2 (2007), p. 4.

[116] M.B. Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *Micro, IEEE* 33.5 (Sept. 2013), pp. 8–19. ISSN: 0272-1732. DOI: 10.1109/MM.2013.90.

[117] Michael B Taylor. "A landscape of the new dark silicon design regime". In: *IEEE Micro* 33.5 (2013), pp. 8–19.

[118] Michael B Taylor. "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse". In: *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE. 2012, pp. 1131–1136.

[119] Sanjeev Thiyagarajan. "Reducing memory space for completely unrolled LU factorization of sparse matrices". MA thesis. University of Cincinnati, 2001.

[120] James E. Thornton. “Parallel Operation in the Control Data 6600”. In: *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*. AFIPS ’64 (Fall, part II). San Francisco, California: ACM, 1965, pp. 33–40. DOI: 10.1145/1464039.1464045. URL: http://doi.acm.org/10.1145/1464039.1464045.

[121] Robert M Tomasulo. “An efficient algorithm for exploiting multiple arithmetic units”. In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[122] Stephen M Trimberger. “Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331.

[123] J. Tyhach, M. Hutton, S. Atsatt, A. Rahman, B. Vest, D. Lewis, M. Langhammer, S. Shumarayev, T. Hoang, A. Chan, D. M. Choi, D. Oh, H. C. Lee, J. Chui, K. C. Sia, E. Kok, W. Y. Koay, and B. J. Ang. “Arria 10 device architecture”. In: *2015 IEEE Custom Integrated Circuits Conference (CICC)*. Sept. 2015, pp. 1–8.

[124] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. “A survey on FPGA virtualization”. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 131–1317.

[125] John Von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 4 (1993), pp. 27–75.

[126] Erwei Wang, James J Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter YK Cheung, and George A Constantinides. “Deep Neural Network Approximation for Custom Hardware: Where We’ve Been, Where We’re Going”. In: *arXiv preprint arXiv:1901.06955* (2019).

[127] Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. “HopliteRT: An efficient FPGA NoC for real-time applications”. In: *Field Programmable Technology (ICFPT), 2017 International Conference on*. IEEE. 2017, pp. 64–71.

[128] Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. *Worst Case Latency Analysis for Hoplite FPGA-based NoC*. Tech. rep. 2017.

[129] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Tech. rep. California University Berkeley Department of Electrical Engineering and Computer Sciences, 2014.

[130] Matt Weber. "Arbiters: design ideas and coding styles". In: *SNUG Boston* (2001).

[131] Stephen Weston, James Spooner, Sébastien Racanière, and Oskar Mencer. "Rapid computation of value and risk for derivatives portfolios". In: *Concurrency and Computation: Practice and Experience* 24.8 (2012), pp. 880–894.

[132] Henry Wong, Vaughn Betz, and Jonathan Rose. "High Performance Instruction Scheduling Circuits for Out-of-Order Soft Processors". In: *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE. 2016, pp. 9–16.

[133] Justin M Wozniak, Michael Wilde, and Ian T Foster. "Language features for scalable distributed-memory dataflow computing". In: *Proc. Data-Flow Execution Models for Extreme-Scale Computing at PACT* (2014).

[134] Guiming Wu, Xianghui Xie, Yong Dou, Junqing Sun, Dong Wu, and Yuan Li. "Parallelizing sparse LU decomposition on FPGAs". In: *Field-Programmable Technology (FPT), 2012 International Conference on*. 2012, pp. 352–359.

[135] Xilinx Inc. *LogiCORE IP Floating-Point Operator v7.0 Product Guide*. 2014. URL: http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf.

[136] Xilinx Inc. *Versal, the First Adaptive Compute Acceleration Platform*. 2018. URL: www.xilinx.com/support/documentation/whitepapers/wp505-versal-acap.pdf.

[137] I Xilinx. "Microblaze processor reference guide". In: *reference manual* (2016).

[138] Huazhong Yang, Yu Wang, and Ling Ren. "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling". In: *IEEE Transactions on Parallel and Distributed Systems* 99 (2014), p. 1.

[139] Laurence Tianruo Yang. "The improved CGS method for large and sparse linear systems on bulk synchronous parallel architectures". In: *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*. IEEE. 2002, pp. 232–237.

[140] AN Yzelman and Rob H Bisseling. “An object-oriented bulk synchronous parallel library for multicore programming”. In: *Concurrency and Computation: Practice and Experience* 24.5 (2012), pp. 533–553.

[141] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. “Exploring energy scalability in coprocessor-dominated architectures for dark silicon”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), p. 130.

[142] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R Gao. “Using a codelet program execution model for exascale machines: position paper”. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. ACM. 2011, pp. 64–69.