DaCO: A High-Performance Token Dataflow Coprocessor Overlay for FPGAs

Siddhartha
Nanyang Technological University, Singapore
siddhart005@e.ntu.edu.sg

Nachiket Kapre
University of Waterloo, Canada
nachiket@uwaterloo.ca

***Abstract*—**
Dataflow computing architectures exploit dynamic parallelism at the fine granularity of individual operations and provide a pathway to overcome the performance and energy limits of conventional von Neumann models. In this vein, we present DaCO (Dataflow Coprocessor FPGA Overlay), a high-performance compute organization for FPGAs to deliver up to 2.5× speedup over existing dataflow alternatives. Historically, dataflow-style execution has been viewed as an attractive parallel computing paradigm due to the self-timed, decentralized nature of implementation of dataflow dependencies and an absence of sequential program counters. However, realising high-performance dataflow computers has remained elusive largely due to the complexity of scheduling this parallelism and data communication bottlenecks. DaCO achieves this by (1) supporting large-scale (1000s of nodes) out-of-order scheduling using hierarchical lookup, (2) priority-aware routing of dataflow dependencies using the efficient Hoplite-Q NoC, and (3) clustering techniques to exploit data locality in the communication network organization. Each DaCO processing element is a programmable soft processor and it communicates with others using a packet-switching network-on-chip (PSNoC). We target the Arria 10 AX115S FPGA to take advantage of the hard floating-point DSP blocks, and maximize performance by multipumping the M20K Block RAMs. Overall, we can scale DaCO to 450 processors operating at an f_{max} of 250 MHz on the target platform. Each soft processor consumes 779 ALMs, 4 M20K BRAMs, and 3 hard floating-point DSP blocks for optimum balance, while the on-chip communication framework consumes $< 15\%$ of the on-chip resources.

I. Introduction

FPGAs have assumed an important role in modern computing systems through deployments in cloud environments like Microsoft Azure [1], Amazon F1. New products like the Intel Xeon-FPGA 6138P hybrid SoC, and the Xilinx ACAP 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) through programming languages like C/C++, OpenCL, as well as embedded design ecosystems like Xilinx PYNQ. 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 co-processor. Another way to leverage this capacity is through soft-processor + network-on-chip overlays such as the 1680-core GRVI-Phalanx system [2]. Thousands of tiny, 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.

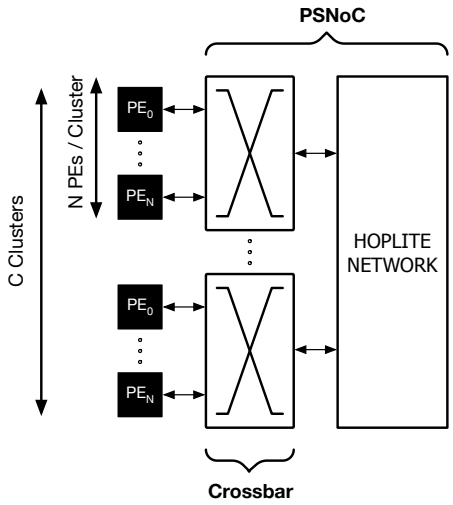


Figure 1: DaCO topology: C clusters of N processing elements (PE) connected by local crossbar arbiters; inter-cluster communication supported by Hoplite NoC

The goal of this paper is to demonstrate an FPGA overlay design of a dataflow coprocessor, called DaCO, that maximizes the resource efficiency (ALMs, M20Ks, DSPs) to deliver a manycore dataflow acceleration engine on an Arria 10 FPGA. DaCO is composed of dataflow-driven soft-processors that communicate over a hierarchical packet-switching network-on-chip [3], [4] (PSNoC). Figure 1 shows a high-level architectural view of DaCO.

We propose the following key techniques to address the challenges of implementing dataflow on FPGAs:

- We devise a hardware-friendly criticality-aware OoO (out-of-order) scheduling technique that uses a bit-vector to capture node readiness and supports that with a hierarchical lookup approach. This technique avoids squander-

ing precious on-chip BRAMs on active-ready queues used by contemporary dataflow systems and instead free 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 schedule across 1000s of active nodes.

- 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 Hoplite-Q NoC to support configurable clustering to determine the right balance of resource cost and dataflow execution time.
- We pay close attention to the ALM, M20K, and 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.

II. Background

A. Dataflow Principles and Limitations

The Good: Unlike conventional soft processors, dataflow processors have *no program counter*. Instead, the token dataflow processor operates directly on a dataflow graph (DFG) 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 DFG 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 DFG 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 DFG 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 where we must choose between many ready instructions to process in the processor. 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 2 shows an example trace of a benchmark (`bomhof2`) on a 4x4 DaCO 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 using on-chip block RAMs (BRAMs), which is an inefficient use of the scarce hard RAM resource.

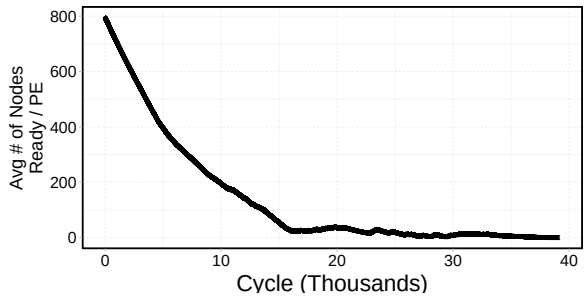


Figure 2: Average number of nodes ready / *per processor* at every cycle. Trace obtained by evaluating benchmark `bomhof2` on a 4x4 DaCO instance (cluster size = 1)

The Ugly: The tail-end of the execution trace in Figure 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 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.

B. Brief History of Dataflow

Token dataflow was originally introduced through the MIT static dataflow machines [5], [6]. The static dataflow model, while promising at its inception, was largely relegated to academic research due to its limited applicability as a general-purpose computing model. In addition, the aggressive frequency/density scaling of the 1990s favored the classic von Neumann single-threaded microprocessor architectures. Nevertheless, the token dataflow overlay has seen interest lately as evidenced by [7]–[9]. The Qualcomm R1 and R2 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 exploiting high instruction-level parallel (ILP) regions in non data-parallel regions, commonly found in sparse workloads [7].

C. OoO in FPGA-based soft processors

Most existing FPGA soft processors are simple in-order processors (*e.g.* MicroBlaze [10], NIOS [11]), including existing token dataflow soft processors [8]. 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). Recent work [12], [13] in this domain took on the challenge of designing out-of-order schedulers for traditional von Neumann soft-processors (>1000 LUTs for a 16–40 scheduling window). EDGE [9] 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.

D. Arria 10 AX115S

Table I: 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

Our target FPGA in this body of work is the Arria 10 AX115S. Table I 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 the resource efficiency of M20K blocks – ideally, M20K blocks should only store the DFG structure in order to maximize the largest application sizes that can fit inside the on-chip memory.

III. Dataflow Coprocessor Overlay (DaCO)

DaCO is a collection of customized dataflow soft-processors that communicate with each other in a hierarchical fashion: 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 network. Figure 1 shows an example N-cluster DaCO engine, where N is the number of processing elements (PE) inside each cluster. In this section, we describe in detail each of the building blocks depicted in Figure 1.

A. Processing Element (PE)

Each PE is a custom dataflow soft-processor composed of five components: on-chip node/edge memory, arithmetic logic unit (ALU), packet consumer, packet generator, and a scheduler for managing computation at runtime. Figure 3 shows these five modules and their layout within the processor design. Each PE communicates by sending/receiving packets to/from the local crossbar arbiter or the Hoplite router directly (if clustering is disabled). 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 NoC architecture, as a packet is always allowed to exit into a PE.

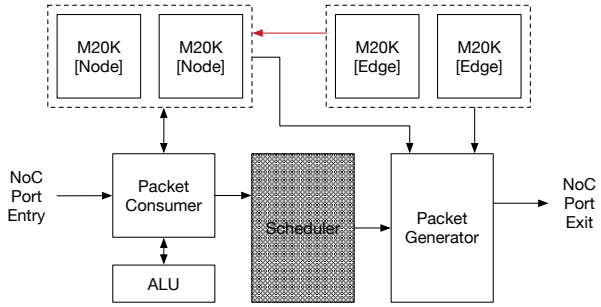


Figure 3: Dataflow soft-processor design

1) **Node and Edge Memory:** We fracture the dataflow graph memory into two distinct node and edge memory structures stored in separate M20K RAMs. This decision is motivated by the different memory access patterns to node and edge state in the processor. Node state is used to store input operands for the dataflow instruction (node) as well the result of the dataflow operation, whereas edge state (dataflow dependency information) is read only by the packet generator module. 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 I). We allocate two BRAMs each to store node and edge state. Due to the fractured graph memory design, there is a small storage overhead for saving addressing information that connects each node to its respective edges in the edge memory (depicted by the red arrow in Figrue 3). Figure 4 shows how the node and edge states are packed into each addressable slice in an M20K BRAM. As a consequence, each M20K BRAM can pack 512 $\times$ 40b node slices or 1024 $\times$ 20b edge slices. There is a $\approx$14% overhead to store node-to-edge addresses described above, 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 characteristics of the dataflow benchmarks in this study, which have $\approx$1.2–1.3$\times$ more edges than nodes.

Multipumping [14] is a well-known overclocking technique that can be used to create 2$\times$ read and 2$\times$ write ports out of the 1$\times$ read and 1$\times$ write port natively supported by the on-chip M20K BRAMs on the FPGA. This gives the advantage of having dedicated read/write ports for each stage, simplifying the design logic and eradicating any non-determinism in the memory operations. Multipumping incurs a small control logic overhead ($\approx$30 ALMs) and achieves the target 250MHz system overlay clock, since the hard M20K blocks can be clocked up to 645MHz.

2) **Arithmetic Logic Unit (ALU):** The hardened floating-point DSP (FPDSP) block is an attractive feature of the Arria 10 FPGAs that enables high-performance IEEE-compliant *single-precision* floating point computation with-

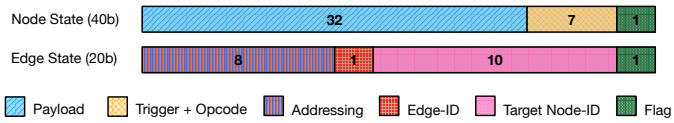


Figure 4: Node state and edge state packed in M20K. Flag guides the control datapath in the processor.

out the need of 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 required for each benchmark type.

3) **Packet Consumer:** The packet consumer implements the key tenet of token dataflow computing – the dataflow firing rule. In this case, the consumer manages the dataflow state of all locally-stored nodes in the node memory. This involves fulfilling 4 key tasks:

- Storing the payload of arriving packets at each processor into the local graph memory,
- Sending instructions, with payload, to the ALU when all operands of a local node have been received
- Storing the ALU result back into the node memory, and
- Notifying the next pipeline stage that a node is ready for edge evaluation to create NoC 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. The node slice contains a trigger bit-vector (see Figure 4), 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, `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, while also supporting writeback from the ALU. The packet consumer datapath is fully-pipelined in order to support an Initiation Interval (II) of 1, *i.e.* the PE can accept back-to-back packets from the network safely. This is achieved by designing a robust data-forwarding circuit that prevents read-after-write data hazards with careful support for back-to-back packets headed to the same destination node.

Finally, the processor notifies the next processor stage that a node is ready for edge processing. This notification is dependent on the scheduling strategy that we discuss in detail in Section III-A5. The flag in the node slice (see Figure 4) indicates whether the dataflow node has any outgoing dependencies (outgoing edges).

4) **Packet Generator:** The packet generator is responsible for iterating over and generating a packet for each of the outgoing edges of all nodes that have finished dataflow firing. These packets are then injected into the PSNoC communication framework and routed to their destination. The packet generator is a finite state machine (FSM) that manages this entire process. With the help of a small fixed-size FIFO, the FSM also executes speculative memory reads to support back-to-back packet injection into the network (assuming there is no congestion). The packet generation logic needs an initial setup cost of 6 cycles for each node to extract its fanout information (due to the fractured node/edge memories described in Section III-A1). This setup phase also masks the cost to reset the LOD ready flags, such that the next ready node is scheduled safely before the FSM has finished processing all the fanouts of the current active node.

5) **Scheduler:** At any given cycle during the graph execution, several nodes may be ready for edge processing to generate new packets (see Section III-A4). We develop a hierarchical, criticality-aware, out-of-order, leading-ones detector (LOD) circuit for scheduling and compare it to a naïve in-order FIFO-based scheduler to demonstrate its effectiveness.

Naïve In-Order baseline: In the simplest implementation, we can connect the packet-consumer and packet-generation logic with sufficiently-deep FIFOs that 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, not all nodes are equal, as some nodes along the critical path of the dataflow graph are more critical than others. If we can schedule nodes along the critical path sooner, we can ensure that the entire dataflow graph executes faster.

Software Pre-processing: In order to implement a node scheduling strategy in hardware, we first identify the critical nodes in the dataflow graph. To achieve this, we run a **one-time** software pass in the dataflow compiler 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 nodes are statically assigned a real number C_n between zero and one that indicates their criticality in the dataflow graph, with one being most critical.

Memory Storage: 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 memory). This is an important step that allows us to pick the most critical ready node in hardware using a simple LOD circuit. The LOD is a well-studied circuit that takes a bit-vector as an input, and evaluates the position of the leading one in the input bit-vector.

Leading Ones Detection : Identifying the leading one in the sorted memory layout implicitly guarantees the most critical node is selected for the next stage of packet gen-

eration. However, the size and speed of the LOD module needed is $\propto$ number of locally-addressable node slices in each processor (*e.g.* 512-wide LOD requires a whopping 800 ALMs and runs at a slow f_{max} of below 200MHz). To address this, we design a multi-cycle hierarchical LOD scheduler that trades-off processor latency for a much leaner design and improved clock performance. Fortunately, the scheduler latency is hidden at runtime since the packet generator is busy for a longer number of cycles servicing the current active node. Figure 5 shows the design of the LOD scheduler.

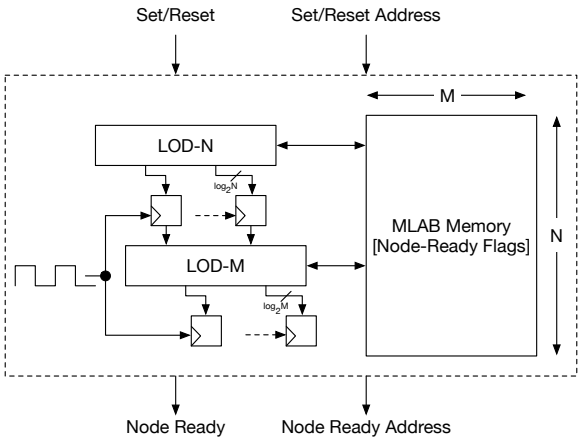


Figure 5: Hierarchical (depth = 2) LOD scheduler design

FPGA-Friendly 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 an 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 ready flags memory can be realized with a small 20 ALM overhead. The ready flags are fractured and packed as an 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 scheduler now has a latency of 3 cycles, but is fully-pipelined to support back-to-back set/reset instructions. Through experiments, 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 III-A1 has a positive effect on the LOD scheduler as well, since now we only need to allocate and track ready flags for addresses of node states. Later in Section V, we compare the performance of the LOD-based scheduler shown in Figure 5 against the baseline FIFO-based in-order scheduler.

B. PSNoC Design

As depicted in Figure 1, we adopt a hierarchical communication framework topology. The PSNoC is composed of crossbars to support intra-cluster communication, and a Hoplite/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 slows down F_{max}. At some point, we expect diminishing returns from moving to larger cluster sizes. 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 network. An arriving packet from the Hoplite network is always given priority over other local packets, if there is a conflict.

Inter-cluster communication is supported by a sparser Hoplite network in a 2D-torus topology. 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.

IV. Methodology

All components of DaCO are designed and written in Verilog and synthesized using Quartus Prime v18.0. We use Verilator [15] 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 Hoplite 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. Tables II and III give a breakdown of the resource utilization for various components in DaCO.

We run each benchmark with varying system and cluster sizes. We vary total system size from 1$\times$1 to 16$\times$16 (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 DFGs and producing criticality-aware optimizations described in this paper. We use PaToH [16] to do cluster-aware graph partitioning.

We extract traces from eight different sparse matrix benchmarks from the circuit simulation domain. `bomhof` and `hamm` matrices are available from the MatrixMarket collection, while the remaining matrices are selected from the ISCAS89 benchmark set [17]. The traces are extracted from the sparse matrix factorization phase, which is evaluated millions of times in an iterative fashion, forming the compute bottleneck.

Table II: Soft-processor resource utilization breakdown

Sub-Module	ALMs	Registers	M20Ks	DSPs	Clock (ns)
ALU	16	17	0	2	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	2	3.7
Total (PE-Baseline)[2]	457	1121	5	2	3.9

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

Table III: Resource Utilization Breakdown (ALMs) and clock performance (ns) of cluster crossbar (CC) and packet-switching (PS) routers that make up the NoC

CC Size	[1]RR Arbiter	Muxes	Total	Clock (ns)
2	4	114	118	3.6
4	27	326	353	3.6
8	242	1015	1257	3.6
16	963	3855	4818	3.7
PS-Router	**DOR Arbiter**	**Muxes**	**Total**	**Clock (ns)**
Hoplite	4	33	56	3.0
Hoplite-Q	127	40	215	3.0

[1]RR = Round-Robin

V. Results

We compare DaCO's performance against a prior state-of-the-art FPGA dataflow processor design as well as an optimized CPU implementation using a well-known linear algebra library (Eigen). We measure runtime in cycles required to process the graphs on the dataflow overlay. Our token dataflow baseline [8] has no clustering and OoO scheduling and we refer to it as *DF Baseline* in this section.

A. Runtime Performance

Table IV summarizes the runtime results across 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, the microprocessor is also able to take advantage of that significantly. We hypothesize that a better data-locality-aware partitioning strategy could further close the performance gap for such benchmarks.

B. Effect of criticality-aware scheduling

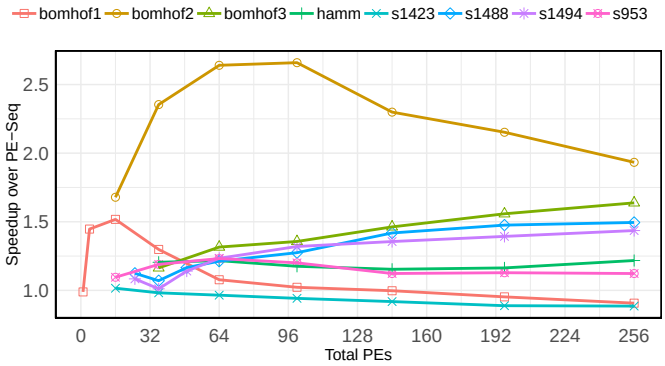
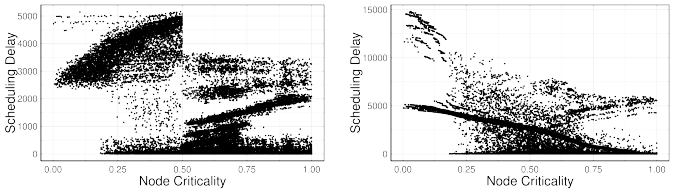


Figure 6: Effect of replacing Baseline PE (in-order) with DaCO PE (out-of-order). Cluster size fixed to 1.

Figure 6 shows the speedup observed when we isolate the effect of criticality-aware scheduling inside the DaCO PEs at varying system sizes. 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%. This is because the DFG of `s1423` has several parallel critical paths, quantified by the high average edge criticality of 0.84 (vs 0.7 in `bomhof2`). Hence, criticality-aware scheduling does not benefit `s1423` as most nodes are of equal importance, and the performance loss is due to unpredictable runtime effects (*e.g.* congestion/deflection).



(a) DF Baseline (b) DaCO

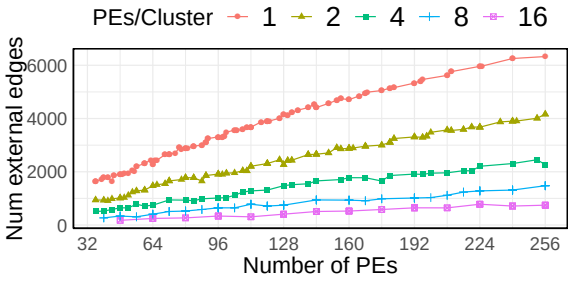
Figure 7: Scheduling delay suffered by node vs node criticality (`bomhof2`)

Figure 7 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, unlike the FIFO implementation. There are, however, some high-criticality nodes in DaCO which still suffer from large scheduling delay (> 5000 cycles). That is due to the long sequential tail of the dataflow graph, which is unavoidable. A rudimentary criticality-delay product heuristic indicates that DaCO schedules nodes by criticality better by up to 87% (15% mean across all benchmarks).

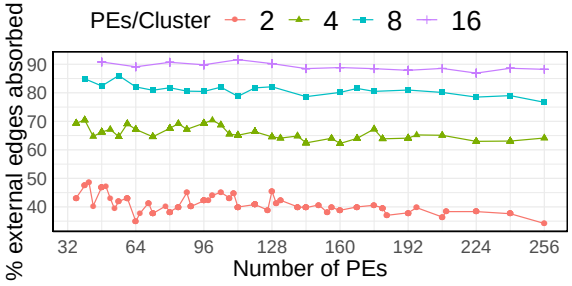
Table IV: Best-case benchmark runtimes with different types of PEs, compared against a baseline CPU implementation

				DF Baseline		DaCO				CPU[1]		
Benchmark	**Nodes**	**Edges**	**Critical Path**	**PEs**	**Time (us)**	**C**	**N**	**PEs**	**Time (us)**	**Time (us)**	**vs Baseline**	**vs DaCO**
bomhof1	1925	2408	57	256	4.2 (1.0×)	16	16	256	3.8 (1.1×)	8.9	0.5×	0.4×
bomhof2	35609	45796	501	256	81.9 (1.0×)	16	16	256	34.4 (2.4×)	23.8	3.4×	1.4×
bomhof3	75305	90264	494	256	79.1 (1.0×)	16	16	256	36.2 (2.2×)	81.1	1.0×	0.4×
s953	37671	44052	171	256	15.8 (1.0×)	16	16	256	12.9 (1.2×)	32.0	0.5×	0.4×
s1423	52310	60852	470	256	30.4 (1.0×)	16	16	256	32.2 (0.9×)	48.0	0.6×	0.7×
s1488	86035	101608	655	256	83.5 (1.0×)	16	16	256	47.4 (1.8×)	70.8	1.2×	0.7×
s1494	86444	102060	628	256	77.1 (1.0×)	16	16	256	45.3 (1.7×)	69.5	1.1×	0.7×
hamm	115616	135416	337	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 (compiled with `-O3`, single-core performance)



(a) External edges at varying system/cluster sizes



(b) % of external edges absorbed at varying system/cluster sizes

Figure 8: Communication trace of `bomhof2` at varying system and cluster sizes

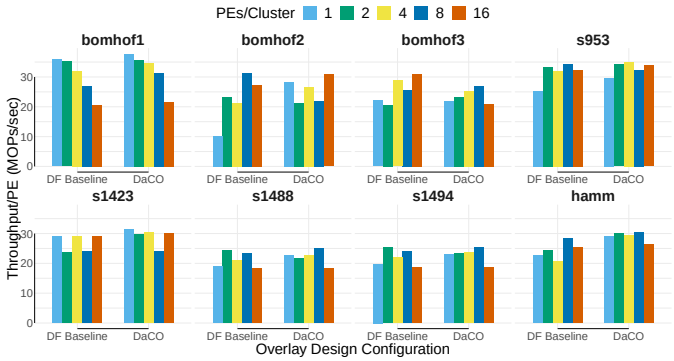


Figure 9: Throughput/PE with varying cluster size across different benchmarks

C. *Effect of clustering*

Figure 8 shows the effect of clustering on `bomhof2`. As cluster size increases, the number of inter-cluster edges decreases as expected. However, we observe diminishing returns when cluster size is increased beyond 4 PEs/cluster. As resource utilization of the cluster crossbar grows by $O(N^2)$ (see Table III), we recommend a cluster size of 2–4 to deliver the most resource-efficient runtime performance.

Figure 9 shows the runtime performance impact of clustering on both DF Baseline and DaCO across all benchmarks. For small benchmarks like `bomhof1`, few PEs in a small cluster configuration delivers 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.

D. *Peak GFLOPs vs. Resource Utilization*

Figures 10 and 11 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. Furthermore, by freeing up the FIFO BRAMs in the DF Baseline, DaCO can accommodate 20% larger graphs in the on-chip memory, hence, offering better scalability and resource balance at the same time.

VI. Conclusions

We present DaCO – Dataflow Coprocessor Overlay – a high-performance token dataflow overlay architecture for Arria 10 FPGAs. DaCO improves over existing token dataflow baseline with three improvements: (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 tradeoff resource for runtime performance, and (3) careful RTL design to maximize resource utilization on the target AX115S FPGA board. Overall, the criticality-aware OoO scheduler delivers up to 2.5× speedup over existing dataflow baseline implementations, and up to 2.8× over a baseline CPU implementation, with a small 15–40% resource overhead from clustering (cluster size 2–4).

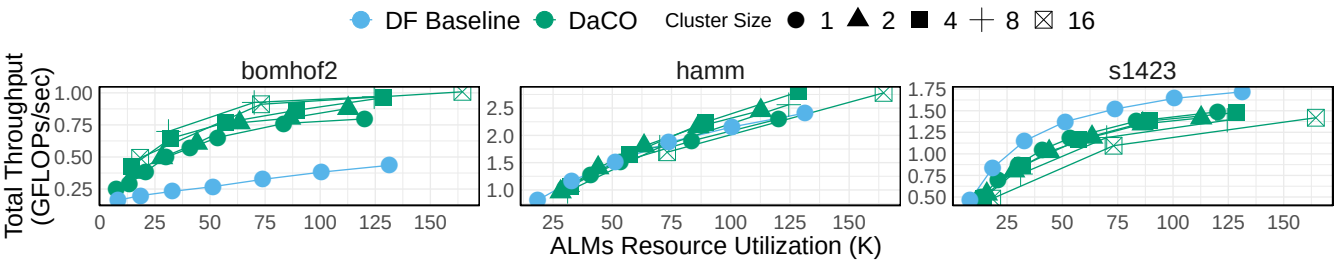


Figure 10: Total through vs ALM utilization observed on 3 representative benchmarks with varying system sizes

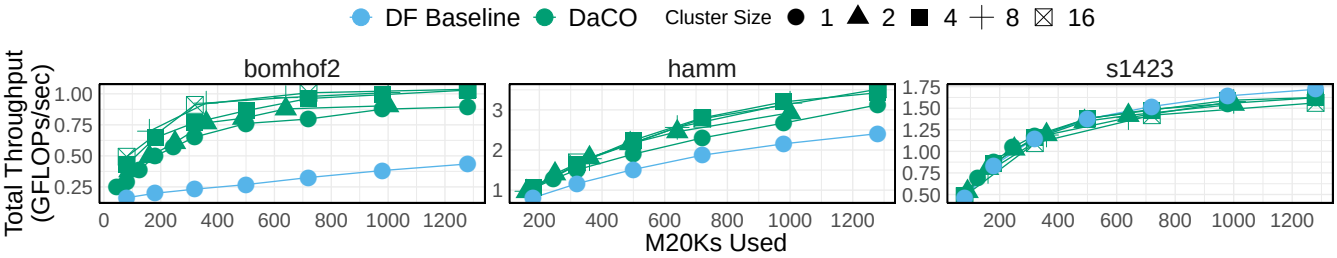


Figure 11: Total throughput vs M20K utilization observed on 3 representative benchmarks with varying system sizes

References

[1] A. P. Adrian Caulfield, Eric Chung and et al., "A Cloud-Scale Acceleration Architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[2] J. Gray, "GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 17–20.

[3] 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.

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

[5] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4, pp. 126–132, 1975.

[6] A. R. Hurson and K. M. Kavi, "Dataflow computers: Their history and future," *Wiley Encyclopedia of Computer Science and Engineering*, 2007.

[7] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[8] N. Kapre and A. DeHon, "Parallelizing Sparse Matrix Solve for SPICE circuit simulation using FPGAs," in *Field-Programmable Technology*, Jan. 2010.

[9] J. Gray and A. Smith, "Towards an area-efficient implementation of a high ILP EDGE soft processor," *CoRR*, vol. abs/1803.06617, 2018. [Online]. Available: http://arxiv.org/abs/1803.06617

[10] I. Xilinx, "Microblaze processor reference guide," *reference manual*, 2016.

[11] I. Nios, "Gen2 processor reference guide," *Altera Corporation (Reference Guide). Cited on page*, vol. 10, p. 108, 2015.

[12] H. Wong, V. Betz, and J. 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.

[13] K. Aasaraai and A. 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.

[14] C. E. LaForest and J. G. 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.

[15] W. Snyder, "Verilator and SystemPerl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[16] U. V. Catalyürek and C. Aykanat, "PaToH: a multilevel hypergraph partitioning tool, version 3.0," *Bilkent University, Department of Computer Engineering, Ankara*, vol. 6533, 1999.

[17] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Circuits and Systems, 1989., IEEE International Symposium on*. IEEE, 1989, pp. 1929–1934.