UNIVERSITY OF CALIFORNIA SAN DIEGO

Accelerating Data Movement at Different Granularities in Datacenters

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Shu-Ting Wang

Committee in charge:

Professor Steven Swanson, Chair
Professor George C. Papen
Professor Geoffrey M. Voelker
Professor Jishen Zhao

2024

Copyright
Shu-Ting Wang, 2024
All rights reserved.

The dissertation of Shu-Ting Wang is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

To Olivia, my mother, and late father

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Table of Contents v

List of Figures viii

List of Tables xi

Acknowledgements xii

Vita xiv

Abstract of the Dissertation xv

Chapter 1 Introduction 1

Chapter 2 Daronpon: Datacenter Load Balancing Across Racks 4
- 2.1 Background 6
  - 2.1.1 Datacenter load balancing 6
  - 2.1.2 Microsecond timescales 7
- 2.2 Challenges 8
  - 2.2.1 Microsecond load balancing at scale 8
  - 2.2.2 Load knowledge 9
  - 2.2.3 Stale information 10
  - 2.2.4 Information traffic reduction 11
  - 2.2.5 Heterogeneous rack configurations 11
- 2.3 Daronpon design 12
  - 2.3.1 Microbursts 13
  - 2.3.2 Packet flow 14
  - 2.3.3 Logarithmic Gossip 15
  - 2.3.4 Piggyback 19
- 2.4 Implementation 21
- 2.5 Evaluation 22
  - 2.5.1 Experimental Setup 23
  - 2.5.2 Workloads 23
  - 2.5.3 End-to-end experiments 25
  - 2.5.4 Heterogeneous server configurations 27
  - 2.5.5 Gossip and Piggyback Overhead 28
- 2.6 Discussion 30
- 2.7 Future Work 31

2.8 Conclusion 32
2.9 Sources for Material Presented in This Chapter 32

Chapter 3 Fianchetto: Accelerating Data Motion Across the Board 33
3.1 Preliminaries and Motivation 36
3.1.1 Data Restructuring Operations 37
3.1.2 Data Motion Overheads 39
3.2 Fianchetto: Accelerating the Data Motion 41
3.3 DRX Placement 42
3.4 Data Restructuring Accelerator (DRX) Design 46
3.4.1 Data Restructuring Characterization 46
3.4.2 DRX Hardware Architecture 47
3.5 System Integration and Programmability 50
3.6 Experimental Methodology 55
3.7 Experimental Results 57
3.7.1 End-to-end Performance Improvement 57
3.7.2 DRX Placement Analysis 61
3.7.3 Sensitivity Studies 64
3.8 Related Work 69
3.9 Conclusion 70
3.10 Sources for Material Presented in This Chapter 71

Chapter 4 Aurelia: Scalable CXL fabric 72
4.1 Motivation and Background 73
4.1.1 What is Different with CXL Fabric? 75
4.1.2 Use Cases of CXL Fabric 75
4.2 Challenges 78
4.2.1 Addressing and Routing Challenges 78
4.2.2 Transport-level Challenges 79
4.3 Design of Aurelia 81
4.3.1 Addressing & Flow 82
4.3.2 Routing 83
4.3.3 End-to-end Congestion Control 85
4.4 Evaluation of Aurelia's Design 88
4.4.1 Packet-level Simulation using ns-3 88
4.4.2 Simulation of Large Model Inference 89
4.4.3 Simulation Results on YCSB Benchmarks 90
4.5 Discussion 92
4.6 Conclusion 94
4.7 Sources for Material Presented in This Chapter 94

Chapter 5 Conclusion 95

Bibliography 97

LIST OF FIGURES

Figure 2.1: High-level overview of Daronpon. Each ToR tracks outstanding request for services running in its rack, and maintains approximate counters for remote ToRs hosting shared replicated services. 13

Figure 2.2: Microburst with no mitigation (top) vs. with logarithmic gossip load balancing enabled (bottom) 15

Figure 2.3: Key functionality and message flow of Daronpon. 16

Figure 2.4: The percentage of gossip messages generated by a logarithmic gossip mechanism as a percentage of overall traffic. Collected from runs of 96 KRPS. . . 17

Figure 2.5: Performance breakdown of gossip and piggyback mechanisms. Lower values are better. 20

Figure 2.6: 99th percentile latency improvements on three common service distributions (Constant, Bimodal, Exponential). Each server is provisioned with homogeneous processing power. 25

Figure 2.7: Throughput and latency improvements with skewed processing capacity in a heterogeneous server configuration. Daronpon scales linearly with with the aggregate processing capacity available. 27

Figure 2.8: Service times across three orders of magnitude (2us, 20us, 200us). Daronpon provides relative improvements with similar overheads in terms of piggyback and gossip messages at each service time. 28

Figure 3.1: Current multi-acceleration systems rely on CPU for accelerator chaining. . 34

Figure 3.2: Data motion stands between two application kernels, i.e., Fast Fourier Transform and Support Vector Machine, of an end-to-end application. 38

Figure 3.3: (a) Runtime breakdown when running applications on CPU or multiple accelerator setup that uses CPU for data motion. (b) Multi-acceleration speedup and scalability are constrained by data motion overhead. 39

Figure 3.4: Integrated DRX. 43

Figure 3.5: Standalone DRX. Number of DRX units in Standalone placement is configurable, and the illustration represents just one possible configuration. . . . 44

Figure 3.6: PCIe-Integrated DRX. 45

Figure 3.7: Bump-in-the-Wire DRX. 45

Figure 3.8: Top-down breakdown of stall cycles for data restructuring operations. . . . 46

Figure 3.9: DRX Hardware Architecture. 49

Figure 3.10: DRX instruction types. 49

Figure 3.11: Sample DRX kernel. 52

Figure 3.12: RX/TX data queue pair architecture in Bump-in-the-Wire DRX. 52

Figure 3.13: Point-to-point DMA workflow involves two accelerators and the sending side DRX. 53

Figure 3.14: Fianchetto speedup over *Multi-Axl* configuration that uses CPU for data motion between accelerators. Fianchetto performance scales with the number of concurrent applications by using Bump-in-the-Wire DRX placement. . 58

Figure 3.15: The latency breakdown of the *Multi-Axl* baseline and Fianchetto. Fianchetto shrinks data restructuring ratio from 64.1% to 14.1% in average. 59

Figure 3.16: Fianchetto throughput improvement over *Multi-Axl*. Fianchetto resolves the throughput bottleneck of data restructuring and shifts the throughput bottleneck to the accelerated kernel. 60

Figure 3.17: Comparison of end-to-end latency speedup with different DRX placements. 62

Figure 3.18: System-wide energy reduction, including host CPU cores, accelerators, and DRXs. 63

Figure 3.19: Fianchetto reduces data motion overhead to less than 5% for Personal Info Redaction benchmark extended with Named Entity Recognition kernel. . . 65

Figure 3.20: Fianchetto eliminates redundant DMA transfers and performs DMA in parallel for broadcast and all-reduce on multi-accelerator setup. 67

Figure 3.21: Data restructuring latency speedup with different numbers of RE lanes on DRX. The increase of speedup is limited after 128 lanes. which is our default configuration. 67

Figure 3.22: Fianchetto speedup across generations of PCIe. PCIe Gen4 and Gen5 result in a slight decrease of speedup because their corresponding *Multi-Axl* baselines improve more than their Fianchetto counterparts. 68

Figure 4.1: CXL fabric abstractive topology. Each solid line connecting to CXL fabric is 16 lanes of PCIe 5 or PCIe 6 with a total bandwidth of 128 GB/s or 256 GB/s. 77

Figure 4.2: Congestion on a shared PCIe switch port causes latency spikes of RDMA writes going through the port. 79

Figure 4.3: Experimental setup for PCIe congestion. 80

Figure 4.4: CXL fabric as a fat-tree using 12-bit FAN-ID address *X:Y:Z*, which X, Y, and Z are hexadecimal values. Dash lines represent the routes from switch *2:2:1*'s routing table. 82

Figure 4.5: Aurelia uses EP Backpressure notification to resolve congestions and overload signal for flow control to avoid overrunning the buffer on the device. . 84

Figure 4.6: YCSB benchmarks with higher ratio of writes demonstrate more improvement. 91

LIST OF TABLES

Table 3.1: End-to-end benchmarks. 54

ACKNOWLEDGEMENTS

I want to thank Olivia, my significant other and soon-to-be lifelong partner, for supporting me through my PhD journey. It is a wild ride and thank you to be always on my side.

I want to thank all faculty members who invested their time mentoring me, Prof. George Porter for the first four years of advising that significantly shapes my taste and capability to do research, Prof. Steve Swanson for chairing my committee and giving me the last push to finish the dissertation, Prof. Geoff Voelker for being on my committee and his wonderful operating system class, Prof. George Papen for being on my committee and his insights on networking and optics, Prof. Alex Snoeren on all the critiques on my figures during weekly meetings, and Prof. Jishen Zhao for being willing to serve on my committee.

To Rajdeep, Yibo, Nishant, Stew, Audrey, Anil, Ariana, Alex, and all other tenants of room 3140, thanks for hanging out with me and being my friends. The camaraderie is invaluable, and I will bear that in mind when I embark on my next journey.

To Rohan, Byung Hoon, and Amin, thanks for all the long talks and chats about research and the time of questioning of our life choices while still working on research.

To Pierre-Louis, Weitao, and Dan, thanks for sharing my research interest on CXL and other topics. I really enjoy all the online chats and meetings and the idea of I am not in this alone pushes me to finish the last chapter of this dissertation.

To Dr. Yiting Xia, Jialong, Yiming, and Federico, thanks for hosting me at MPI-INF in Saarbrücken while I am working on this dissertation.

To *Fujimak*, *SmallGGRen*, *HakkaFish* and *StanfordSneaky*, I really enjoy our time on Mario cart racing and chatting about lives and politics. I will keep your real names in private in case other people scoop you folks from me.

I want to thank Yin Chin Foundation for their scholarship helping me for conference travel and supporting myself during finical instable times.

Coffee as the fuel for any research output is essential. I want to thank the great coffee

shops and roasters in San Diego: Bird Rock coffee, the Art of Espresso, and Finjin.

At last, I want to thank all my collaborators and co-authors, who are listed next. Chapter 2, in part, reprints material as it appears in a draft titled: ”Daronpon: Datacenter-scale Sub-RTT Replica Selection for Low-latency Applications” by Shu-Ting Wang, Stewart Grant, Keerthana Ganesan, George Porter, and Alex C. Snoeren. The dissertation author was the primary researcher and author of this material.

Chapter 3, in part, reprints material as it appears in a paper titled: ”Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators” by Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh [WXM+24]. The dissertation author was the primary researcher and author of this material.

Chapter 4, in part, reprints material as it appears in a published WORD’23 workshoppaper titled: ”Aurelia: CXL Fabric with Tentacle” by Shu-Ting Wang and Weitao Wang [WW23]. The dissertation author was the primary researcher and author of this material.

VITA

2013	B. S. in Computer Science, National Tsing Hua University, Taiwan
2015	M. S. in Computer Science, National Tsing Hua University, Taiwan
2016	Information System Technician, Civil Service Training and Protection Commission, Taiwan
2017	Research Assistant, National Taiwan University, Taiwan
2021	Hardware Systems Foundation Engineer Intern, Meta
2024	Visiting Ph. D. student, Max Planck Institute for Informatics, Germany
2017-2024	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Rohan Mahapatra, Soroush Ghodrati, Byung Hoon Ahn, Sean Kinzer, **Shu-Ting Wang**, Hanyang Xu, Lavanya Karthikeyan, Hardik Sharma, Amir Yazdanbakhsh, Mohammad Alian, Hadi Esmaeilzadeh, “In-Storage Domain-Specific Acceleration for Serverless Computing” in Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS), 2024

Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh, “Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators” in Proceedings of 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2024

Shu-Ting Wang and Weitao Wang, “Aurelia: CXL Fabric with Tentacle,” in Proceedings of the 4th Workshop on Resource Disaggregation and Serverless (WORDS), 2023

Rohan Mahapatra, Byung Hoon Ahn, **Shu-Ting Wang**, Hanyang Xu, and Hadi Esmaeilzadeh, “Exploring Efficient ML-based Scheduler for Microservices in Heterogeneous Clusters,” in Proceedings of 2022 MLArchSys Workshop, 2022

ABSTRACT OF THE DISSERTATION

Accelerating Data Movement at Different Granularities in Datacenters

by

Shu-Ting Wang

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Steven Swanson, Chair

The dissertation investigates redundant communication between servers for large-scale web and cache requests and redundant data movement between accelerators for compute-intensive applications. Redundancy is an impending and critical issue for data centers designed for hardware accelerators and disaggregated resources. The dissertation makes the following three contributions to address this. The first contribution of the dissertation is Daronpon. Daronpon dynamically load-balances and reroutes large-scale requests of web and cache applications on a microsecond timescale. Daronpon prevents these requests, stranded on busy servers with network congestion and long queuing delays, from being processed. Daronpon shows improvement in various service time characterizations of different applications. The second contribution of the

dissertation is Fianchetto. Fianchetto acts as a compute-enabled bypass for inter-accelerator communication. Fianchetto accelerates the data restructuring needed between accelerators and saves the data movement between accelerators and CPUs for compute-intensive applications. Fianchetto shows improvement in a series of benchmarks involving different application domains. The third contribution of the dissertation is Aurelia. Aurelia leverages the emerging interconnect of CXL to investigate the design of a scalable fabric for accelerators and fabric-attached memory expansion. Aurelia improves routing and transport based on the current specification of CXL and shows performance improvement on machine learning and key-value store applications.

Chapter 1

Introduction

Modern datacenters are warehouse-scale computers [BCH13]. These warehouse-scale computers serve as the foundation of cloud computing, e.g. IaaS, SaaS, and serverless. They host commodity servers equipped with many general-purpose CPU cores. The servers are interconnected with 100 Gbps or faster network connections. They are clustered into fleets for different functions. For example, one serve the web requests, another one serve as data storage, and the other one serve as an in-memory cache for frequently accessed objects. Communication and data movement between servers are frequent and demonstrate diverse patterns depending on the mix of traffic from different functions [NFG$^+$, KDH$^+$15, HBB$^+$18, azu].

Moreover, hardware accelerators are introduced into the datacenters for compute-intensive applications because of the effective end of Dennard scaling [DGR$^+$74] and the dark silicon phenomenon [EBA$^+$11, HFFA11]. These accelerators include GPUs, TPUs, video codec accelerators, and other accelerators deployed in the production datacenters nowadays [JYP$^+$17, RSC$^+$21, KDH$^+$15, awsb, awsa]. The introduction of these accelerators demonstrates a monumental shift towards a heterogeneous hardware landscape beyond a massive number of homogenous CPU cores. The accelerators accelerate specific compute-intensive workloads, such as video encoding and decoding, scientific computation, and large-scale machine learning applications. They are

integrated with the system and operate on data moved from the host memory and return the results to it. Thus, efficient data movement ensures that these dedicated accelerators are well-utilized.

In addition to the heterogeneous hardware landscape, resource disaggregation aims at efficient resource provision in the datacenters. Resource disaggregation allocates resources, such as CPU cores, memory capacity, and storage capacity, located on different physical machines in a logically unified manner [SHCZ18, ABAL+20, MC20, RSAB20, ZWL+22, LMC+22]. The rationale behind disaggregation is to satisfy users with diverse requests for different resources. Disaggregation, while aiming at efficient provision of resources, exposes the communication and data movement between CPU and memory/devices through a network fabric connecting the resources [CIP+21, Sha23, MWD+23, LBN+23, WW23]. The externalized communication and data movement motivate the need for a scalable network fabric to serve a resource-disaggregated datacenter.

The current datacenters demonstrate diverse communication and data movement patterns on different scales with their corresponding applications. We are, in particular, interested in the data movement of the following scenarios:

1. RPC communication for web and cache applications. The data movement is on the scale of a few packets to a few MBs in total size.
2. Compute-intensive applications with the use of non cache-coherent accelerators. The data movement is with chunks of data on the scale of up to 100s of MBs.
3. Key-value store on disaggregated memory modules. The data movement operates on cacheline granularity to KBs in total size.

In short, we hypothesize that the conventional design of control and data plane demonstrating inefficient communication and redundant data movement between servers for large scale web/cache requests and between accelerators for compute-intensive applications. We argue that this inefficiency is an impending and critical issue for datacenters designed for hardware accelerators and disaggregated resources. This thesis proposes to intelligently redirect the communication and

data movement to bypass congestion and reduce redundant movement with a minimal addition of control logic.

Chapter 2

Daronpon: Datacenter Load Balancing Across Racks

To improve both performance and fault tolerance, datacenter applications are provisioned to scale horizontally, with replicated instances frequently spread across multiple racks of servers. These replicas must be carefully managed to meet strict service-level objectives (SLOs) for both throughput and latency [BMPR17, DB]. These requirements have birthed an architectural paradigm of highly replicated microservices that can (nearly) arbitrarily fan out to deliver ever-higher throughput, and whose functionality is scoped to provide microsecond-timescale responses with low latency even in the tail.

Realizing this design pattern in practice poses multiple engineering challenges, however. Microsecond-timescale services must be resilient to low-level system and network perturbations and short-lived congestion events to achieve consistent performance [ZLZK]. These events arise within the operating system, runtime, and application software as well as due to network-level incast events, where many clients send to a common destination server, causing in-network buffering and substantial queuing delays [NFG$^+$13]. Moreover, the potential impact of these so-called "microbursts" increases with network bandwidth. Large bursts can lead to packet drops

and necessitate end-to-end retransmissions.

Experience shows that a responsive and effective load-balancing strategy is key to managing overall service latency. Given the ultra-low service times of many modern datacenter services, end-to-end approaches driven by the clients and servers themselves are unlikely to meet demands, as many component services (i.e., computation) times are smaller than datacenter-wide network latencies. Instead, we focus on in-network load balancing techniques carried out by network switches or middle boxes—often generally referred to as *dispatchers*. Recent work has shown the benefits of load balancing with a server rack, for example R2P2 [KPG$^+$] and Racksched [ZKC$^+$]. These approaches take into account server load balancing, building upon prior approaches to core scheduling in individual servers [BPP$^+$16, KCH$^+$19, OFB$^+$19, WCB]. Further, Vargaftik et al. [VKO20] show that, for the multiple-dispatcher environments that we target, a stable and highly efficient load balancing approach is possible through a carefully controlled exchange of status updates between servers and dispatchers.

We present Daronpon, an inter-rack load balancer targeting services with round-trip time dominating the overall response time and service time on microsecond timescales. Daronpon periodically exchanges service-load information between dispatchers, either through explicit gossip messages or piggybacked onto redirected application requests. We employ a logarithmic threshold approach to minimize the network overhead of state-exchange messages while ensuring that application requests are forwarded to replicas with good performance. Our system decreases the 99th-percentile tail latency by up to a factor of two over random replica selection across a variety of workloads, enables scaling across heterogeneous server configurations, and provides performance gains for service times as low as two microseconds. [1]

[1] Services are assumed to be replicated throughout Chapter 2, thus replica and services are used interchangeable.

2.1 Background

Modern datacenter applications are replicated [roc21, Mem21, Mon21] to provide fault tolerance, scalability and flexible response to load fluctuations. Often, individual application components are replicated on the order of three times for fault tolerance. In the case of sharding for scalability, however, the number of replicas can be many orders of magnitude higher and even grow dynamically [fac20, AMH$^+$, KXH$^+$] as needed to service demand.

2.1.1 Datacenter load balancing

Traditional application or *layer-4* load balancers operate across the set of back-end services and maintain connection consistency for long-lived flows. They are generally implemented in software using commodity servers [BTY$^+$, EYC$^+$, OAVR, PBY$^+$], although some have explored using switches with hardware support [GLH$^+$, MZK$^+$]. While effective at responding to end-host-driven load imbalances, they are ill-positioned to address transient variations in service times.

Conversely, conventional *layer-3* load-balancing techniques focus on dispersing traffic load in the network, and do not address server imbalance. For example, equal-cost multi-path routing (ECMP) load balances flows using a hash of their 5-tuple. ECMP is widely deployed due to the benefits it gains from average-case statistical multiplexing. It is well known, however, to cause load imbalance due to hash collisions and when links fail. A variety of improvements to ECMP have been proposed, many using the concept of flowlets [AED$^+$, KGH$^+$, KHK$^+$, VPA$^+$]: a group of packets within the same flow separated from others by a large enough time interval. Other schemes load balance per packet [GYG$^+$, KVHD, ZZB$^+$]. Each of these in-network techniques are complementary to our work, as they load balance exclusively based on the state of the network and not the end-host applications.

2.1.2 Microsecond timescales

Despite practitioners' attempts to spread demand evenly across both servers and network fabrics [RZB+], a certain degree of variability is inevitable. Indeed, as links speeds of 100 and 400-Gbps become the norm, hosts and switches can experience significant traffic bursts over short periods of time. These bursts, while lasting only a few microseconds, can cause congestion, lead to increased jitter, and result in packet drops [ZLZK]. As a concrete example, a switch with 64 MB of packet memory will overflow in 1.9 ms at 100 Gbps [tof20] under incast scenarios. While these events originate in the network, the effect of that queuing cascades at the application. Congestion impacts performance directly as batches of requests are delivered in very short time periods. Existing end-to-end techniques struggle to react to microbursts effectively as the durations of the microbursts are shorter than the network RTT which bounds the response time of any end-host approach.

Daronpon targets microsecond-timescale services which do not involve long-lived connections. (We support any request/response service on top of a connection-less transport or one with migration ability [KIB, LRW+].) Techniques for dispatching such microsecond-duration requests within an end-host operating system have been explored recently [BPP+16, KCH+19, MdKA+19, OFB+19, PKB]. These end-host-based approaches use load-aware schedulers to direct requests to CPU cores and to quickly rebalance those allocations. Extensions to these techniques demonstrate similar scheduling performance at the scope of a entire server racks [KPG+, ZKC+]. These efforts make use of programmable switches to track end-host load at the rate of millions of requests per second [tof20]. Both R2P2 [KPG+] and Racksched [ZKC+] are confined to services which operate entirely within a single rack due to a single control point within the ToR switch. This limits the applicability of these schemes for the majority of replicated applications which are distributed throughout the datacenter.

Researchers have also explored explicitly integrating different replication techniques with programmable switching hardware to avoid RTT delays, including chain replication [JLZ+],

Raft [KB], and Paxos [LMS+]. In-network replication techniques have also been proposed to alleviate read/write conflicts [ZBL+19] and contention [LNM+]. Daronpon operates under the assumption that replicas are interchangeable for all requests; our techniques could be extended to support any of these protocols, however Daronpon would require protocol-specific information to operate.

2.2 Challenges

Theoretically, optimal load balancing is attainable with centralized algorithms [Win77] that maintain full knowledge of the global instantaneous system load. A proven-optimal algorithm, join-the-shortest-queue (JSQ), ensures requests experience minimal queuing delay. Unfortunately, centralization prevents cross-rack sharding and scale-out designs. In contrast, fully decentralized approaches (e.g. client-based power-of-two designs [Mit01]) either require network-wide message round trips to measure load or rely on out-of-date information from previous response. As the service time of requests approaches that of an RTT, both approaches provides little benefit on acquiring more up-to-date information. Further, these approaches struggle to react quickly to congestion given these long network-wide round-trips. Load-balancing decisions made with information which has aged by at least an RTT cannot react to microburst events which occur on sub-RTT timescales.

2.2.1 Microsecond load balancing at scale

An ideal solution for microsecond load balancing at scale will therefore be decentralized to avoid bottlenecks, designed to reduce probing overheads, and designed to react nearly instantaneously to bursts. Vargaftik et al. [VKO20] demonstrated theoretically that distributed load balancing for datacenter scale is possible. Their proposal, LSQ, has many desirable properties such as a bounded measurement difference between true server load and the load observed by a

load balancer, which receives load updates as messages using a variety of mechanisms [VKO20].

We investigated via simulation an implementation of LSQ in which we placed LSQ load balancers on core routers within a fat tree network. Using core switches satisfies LSQ's theoretical assumptions with regard to how each load balancer sees sever load by forcing all packets through the core of the network to ensure that every request is visible. In this core-switch realization of LSQ, we find some practical limitations. The first of which is its (lack of) response to microbursts. When requests target the same server, the available bandwidth on the egress port of that server's associated top-of-rack (ToR) switch is stressed, leading to drops [ZLZK]. These bursts occur due to lack of coordination between core routers; they can be prevented by making load-balancing decisions at the ToR instead.

Hence, we propose to load balance on leaf ToRs directly. This alteration has a variety of implications on LSQ's theoretical approach. First, LSQ has approximate knowledge of all server queues. By moving our load balancer to ToRs we lose instantaneous knowledge of remote queues but gain up-to-date load information for all the servers of a rack beneath a ToR. While this updated knowledge gives us tremendous insight into the state of that single rack as shown in R2P2 and Racksched [KPG+, ZKC+], it leaves the dispatchers unaware to the load of services running on remote racks. A key challenge in this work is designing a ToR-based load-balancing approach for datacenter scale by maintaining accurate estimations of the load of other servers at remote ToRs in the absence of periodic updates.

2.2.2 Load knowledge

At datacenter scale, any centralized load balancer is proned to be a performance bottleneck. This is true not only in terms of traffic, but also in terms of the per-application state that the load balancer needs to track. In the absence of centralized knowledge, load information must be distributed among load balancers. While the potential benefit of making decisions locally with even out-of-date information is substantial, the hazards of load balancing in ignorance are

documented in the literature [Dah00, Mit00]. The question of how to efficiently disseminate timely state information in practice remains open.

Updates issued periodically between load balancers leads to a predictable overhead in terms of bandwidth and number of messages, but admits (potentially unbounded) errors in the estimates at remote load balancers. Assuming that events arrive with a Poisson or exponential distribution, many events can occur between load information message exchanges. Alternatively, state updates could be disseminated in relation to the request rate, with updates being issued for every request or perhaps some fraction of the total number of requests. This too suffers as the number of messages issued during bursts of requests can cause an increase in update traffic at the worst possible time. The choice of how to disseminate load in a way that provides up-to-date information in a non-obstructive way is crucial to Daronpon's design.

2.2.3 Stale information

The following experiments show that the frequency at which load is updated has a significant effect on the quality of the decisions. However, there is a trade-off in terms of traffic overheads to keeping information up to date. As the number of services increases, so too does the number of messages needed to keep the information fresh.

Given imperfect, or stale, information about remote servers, the question becomes: *when is redirecting requests a good course of action*? We submit that given reasonably predictable, but potentially erratic, request patterns, the correct course of action is to act conservatively when conditions are manageable, and to react quickly and decisively when request loads spike, i.e., in microburst scenarios. These constraints dictate a compromise between inflating traffic and making sub-optimal decisions with poor information.

2.2.4 Information traffic reduction

Determining how often and when to send load information requires careful consideration. If an update were to be propagated for every request, that would be ideal from a decisions making perspective, but the overall number of messages sent could inflate by the total replication factor ($2\times$ or $3\times$ in many cases). Such overheads are unacceptable for systems with requests in the thousands or millions of requests per second as the cost of the information quickly surmounts the goodput traffic. A key challenge in designing a distributed in-network load balancer approach is to identify the most critical information to send which ideally only produces a small overhead in terms of state exchange messages.

Network operators generally want predictable behavior within their networks. Given that distributed load balancing requires information to be spread, it raises the risk of adding unpredictability to the network, specifically in terms of overhead. An ideal load-balancing strategy would spread information efficiently while allowing operators to set an overhead budget in terms of bandwidth or messages which would correlate to a comparable increase in performance. Predictability is highly important in heterogeneous systems where multiple applications require guarantees about their apportioned network share.

2.2.5 Heterogeneous rack configurations

At datacenter scale the configuration for any application may vary wildly. Individual service replicas may be co-located with resource-hungry applications. Due to configuration differences applications might have varying processing powers. Many applications are placed in VMs which execute on differently powered hardware. Indeed, in the datacenter there is no guarantee that any set of replicas is equally provisioned. Therefore, any load-balancing strategy must take into account this heterogeneity and apportion requests in response to the real processing rate.

2.3 Daronpon design

The Daronpon design resides within the ToR at each rack and is designed to track server load by counting the number of outstanding requests for each service running on that rack. Figure 2.1 illustrates the role that ToRs play in our load balancing scheme. When a request arrives at a ToR it makes a load balancing decision. It either *Admits* the request, or it *Redirects* it to another replica. Requests are redirected in FIFO without specific priority. The choice to either admit or redirect is subject to the local load the service is currently experiencing under the ToR (as understood by Daronpon), and an estimation of the load each replica has on remote ToRs.

A ToR can only directly keep an up-to-date counter for the services running in its rack. To make good load balancing decisions, fresh knowledge, and more importantly knowledge of bursty behavior, is necessary. Determining the mechanisms for disseminating this load information is non-trivial.

We designed and tested a variety of different options for disseminating load information in simulation and on an Amazon Web Services (AWS) testbed. One design gossiped load information periodically based on wall clock time, and another gossiped on a per-request basis. Our results in simulation and on our test setup demonstrate that both of these techniques require extremely high overheads in terms of messages sent. For example, our best results with periodic message exchanges required updates polled every 25us. This resulted in over a 2x overhead in terms of messages. Most of the information spread in this case was redundant, and does not aid in mitigating bursts. Further, it adds congestion to the network, which reduces the maximum throughput, especially during bursts, which is the opposite of our goal. An ideal load dissemination design would quickly react to bursts while simultaneously generating little overhead during burst events.

Daronpon consists of two distinct but complementary messaging mechanisms. The first mechanism is logarithmic gossip that guarantees the reactive spread of load information

when bursts occur, while generating little overhead otherwise. The second one is opportunistic piggybacking that spreads load information between ToRs when requests are redirected. We find in our evaluation that our log gossip approach prevents large queue build-ups, while our piggyback approach reduces latency.

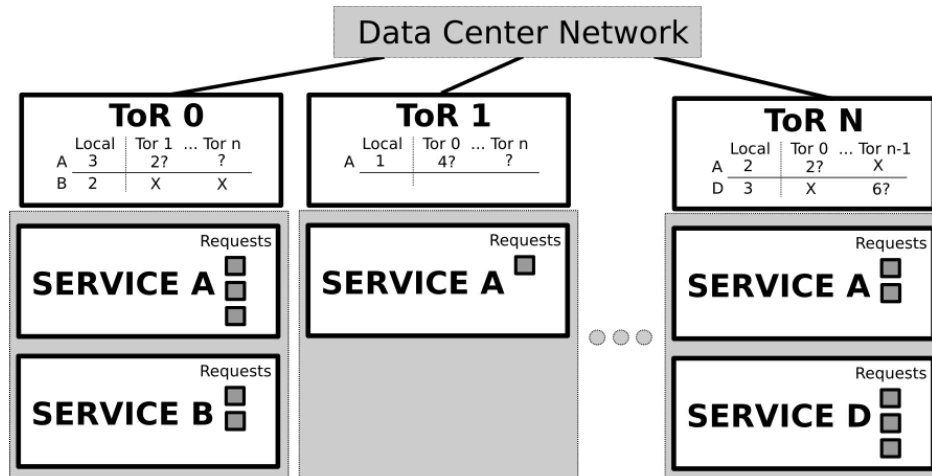


Figure 2.1: High-level overview of Daronpon. Each ToR tracks outstanding request for services running in its rack, and maintains approximate counters for remote ToRs hosting shared replicated services.

2.3.1 Microbursts

Figure 2.2 (top) shows an example of a microburst. In this case, requests are issued at a Poisson arrival rate by multiple clients. The peaks show outstanding requests from the perspective of a ToR instrumented to track request counts. In this case, when requests queue, that queue grows without bound, even though other replicated services are available to process this influx of requests. This has a dramatic impact on the tail latency of the requests in the burst, and also the overall mean request time. A typical request incurs longer wait times due to decreased overall system throughput.

In Figure 2.2 (bottom) the queue builds up with our logarithmic gossip mechanism enabled. Each *red X's* on the chart represents a point at which the load on the server is gossiped. Note that

when peaks occur, and a gossip is sent, the load is quickly spread to other servers. This increases overall system throughput and decreases tail latency. This strategy, however, is not perfect. At low load the benefit of redirecting requests is minimal. For example, when the number of outstanding requests is just one or two above a remote service, and so redirection reduces overall performance.

The age of the gossiped information complicates the act of redirecting. The remote information on remote hosts is at least a few microseconds out of date. Given the few microsecond budget our requests have to begin with, the benefit of redirection quickly evaporates if even a few requests arrive from the point in time at which the load information is sent. This leads to unnecessary redirections and high overheads in terms of gossip messages which do not ultimately deliver useful information. This overhead can be mitigated by adding a threshold which prevents gossip messages from being sent until the number of outstanding requests has exceeded a given threshold. Our proposed log gossip technique for curtailing this overhead is described in the following section.

2.3.2 Packet flow

The Daronpon load balancers are stateful and act per request. Figure 2.3 provides a high level message flow diagram of this system. When requests arrive, Daronpon executes the admission protocol. Requests are admitted only if the local service has the minimum observable global load. A load tracker keeps counters for each global service. Local counters are up-to-date, while remote counters are learned via gossip and piggyback messages. When a request is admitted, the ToR increments its load counter corresponding to that service. When a response passes back through the ToR, that service has its local counter decremented. If, when a request arrives, the local load of a service is not the global minimum, the request is redirected. The redirected request is then sent to the service with the lowest load, based on the ToRs' local load tracker (see Section 2.3.4). Redirected requests have load information attached to them. The attached load consists of request counters for the intersection of services the ToRs share. Therefore, the

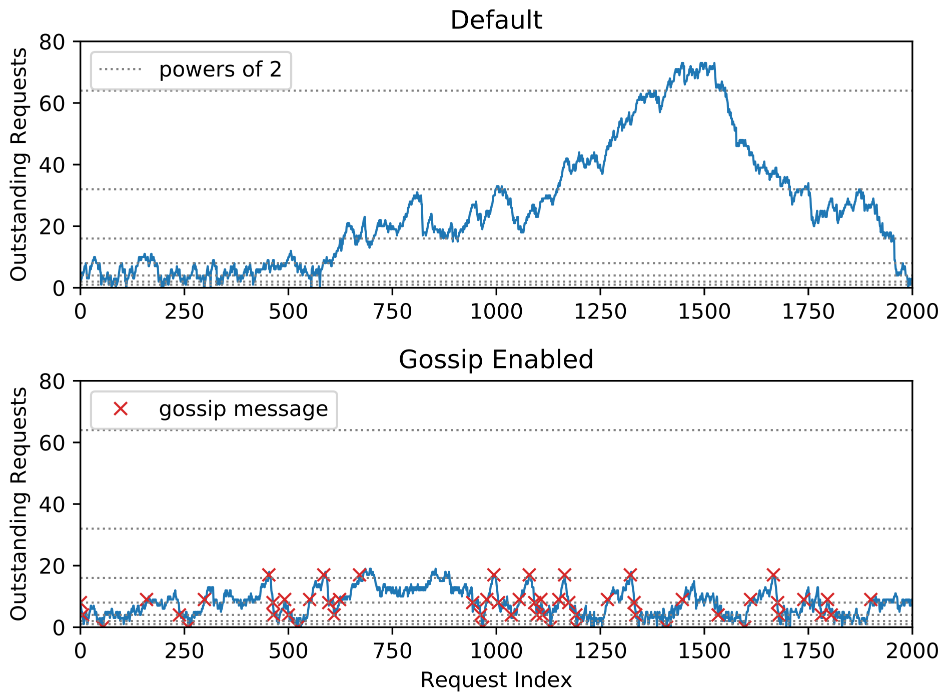


Figure 2.2: Microburst with no mitigation (top) vs. with logarithmic gossip load balancing enabled (bottom)

overhead per redirected request is variable as per the systems' configuration.

Increments and decrements in local load are tracked by a gossip monitor (see Section 2.3.3). The job of the gossip monitor is two-fold. First, it identifies bursts. When load spikes the monitor broadcasts gossip messages to let other ToRs know it is experiencing high load. Second, it identifies valleys. Load balancing schemes which use potentially stale information are known to exhibit herding behavior, a condition which leads to sub-optimal queuing behavior [Dah00, Mit00]. When load drops sharply, gossip messages are also generated to announce that a service has spare processing capacity.

2.3.3 Logarithmic Gossip

Daronpon generates gossip messages aiming to reacting quickly to bursts while consuming little overhead in terms of additional messages. In an earlier design, ToRs gossiped load

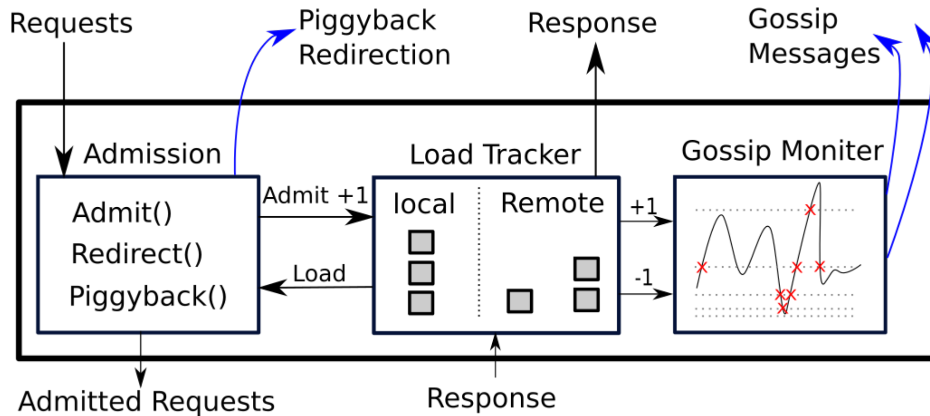


Figure 2.3: Key functionality and message flow of Daronpon. Incoming requests are either admitted or redirected. Redirected requests spread load information via piggyback. Load is updated upon admission, and response. Gossip messages are issues when load breaks a logarithmic threshold.

information every 25 μs. This logarithmic gossip approach has the advantage of providing highly updated load information, but it incurred scalability bottlenecks as the number of ToR increased, since the 25 μs gossip contends with request goodput for link bandwidth.

To compress the number of messages, Daronpon's gossip messages are sent on exponential changes in load. Daronpon uses powers of two as the interval. Daronpon chooses two for its ease of computation requiring only bit shift operations because no commodity programmable switch, to our knowledge, is able to compute floating point arithmetic [GLY$^+$].

Each step increases the bounds in which server load can fluctuate prior to a gossip message being issued. For instance, if the load were to increase from 1 to 2, a gossip message would be broadcast to all other servers with the replicated service that crossed this threshold. In this message, load information for all other shared services is added. When the boundary of two is crossed, and the value of two is sent, this ToR will not issue another gossip message until the load on the service rises by a power of two to four outstanding requests, or falls back down by a power of two to one.

Logarithmic gossip has several benefits. First, it sorts microbursts by magnitude without

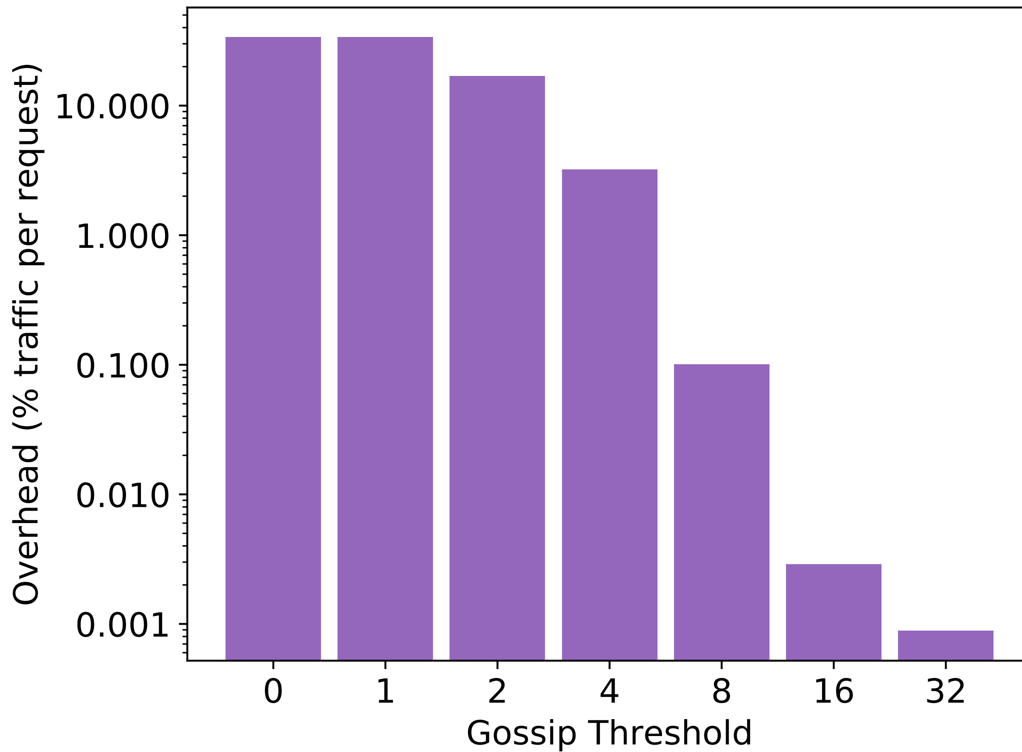


Figure 2.4: The percentage of gossip messages generated by a logarithmic gossip mechanism as a percentage of overall traffic. Collected from runs of 96 KRPS.

adding significant packet overhead. Given that small fluctuations may occur at rapid pace, it is important to give priority to bursts with larger magnitudes.

Second, logarithmic gossip reduces the number of gossip messages sent during bursts, which reduces the overall strain on the system when resources are at their tightest. This is an issue with gossip strategies that operate periodically (e.g. at preordained wall clock times) or are issued at some constant ratio to requests (e.g gossip every 5 requests). When request rates are low, gossip messages are automatically sent with a higher frequency per number of requests which allows for better decision-making at lower request rates. ToRs have the advantage of being an aggregator for the load of an entire rack. Were we to implement our solution on end hosts, each host would need to gossip its load to every other host. Using ToRs, the load of each server in a rack is known explicitly and is gossiped in its entirety, assuming the rack sees both requests and

the associated responses.

Other approaches which use load information collected from end hosts themselves suffer from additional delays in responding to load spikes, as the knowledge of load must be transmitted to the balancer before it can react. Daronpon's logarithmic gossip mechanism reacts to load as soon as a request is admitted. This allows for precise load balancing decisions to be made at sub-RTT time scales. For instance, in a Fat-Tree with two ToRs connected by a single aggregate switch, the distance traveled by load updates is halved in comparisons to a host based solutions (e.g. Tor-Agg-Tor vs Host-Tor-Agg-Tor-Host). This approach is not limited to Fat-trees, as many networks use ToRs, however our ToR-based approach always results in two hops less than an host based approach.

Daronpon's logarithmic gossip estimates server load on the ToR with counters instead of requiring precise and up-to-date measures of load on the host, which we can only get with precise application level knowledge and host control, This imposes lower tracking overhead, and performs nearly as well as highly tuned approaches which report server load directly [ZKC$^+$, Figure. 15 (proactive)]. This algorithm decreases the number of gossip messages significantly, and can be greatly improved by carefully considering a lower bound at which to disable the mechanism entirely. For example, setting a lower threshold such as t implies that below an outstanding request count of t a ToR will not gossip information. The threshold is determined by an exponential weighted moving average and a floor value that the threshold does not go beneath it.

Figure 2.4 shows the percentage of messages gossiped relative to the requests processed for different threshold values. Note that the default values of 0 and 1 have an overhead of around 30% of the request rate. Redirections of requests at these levels of outstanding request sees little benefit in terms of performance as at any reasonably high request rate, the depth of remote queues have changed since the remote data was received making the choice stale. By increasing the threshold to four, the overhead in gossip messages is reduced by a factor of ten down to

around 2% when our system is around 70% saturation. This suggests four is a good floor value for the gossip threshold setting. See Section 2.5.5 for a comprehensive evaluation of gossip overhead. Increasing the gossip threshold beyond four significantly reduces overhead down to around 0.02%, however this comes at the cost of only identifying bursts of size eight and greater which significantly effects our reductions in 99th percentile tail latencies.

Logarithmic gossip, however, provides no liveness guarantees to the freshness of the load information announced despite all the aforementioned benefits. For instance, a server which maintains an outstanding number of requests between 4 and 16 for n requests will not issue a gossip until either threshold is crossed. While this is unlikely in practice due to the small range and short duration of requests, there is no guarantee. This can become an issue for extended periods of load when the number of outstanding requests is deep (e.g. 128 to 512).

2.3.4 Piggyback

Beyond using logarithmic gossip, the redirect requests also able to propagate load information by making load information riding with these requests heading to remote ToRs. With gossip enabled, the percentage of redirected requests is 22%. Each of these redirected requests is sent from the redirected ToR to the receiving one, and therefore has the ability to report the load information on the ToR that performed the redirection. We refer to this method of attaching load information to redirected requests as piggyback, and using redirection to spread information provides significant advantages in the common case. Piggybacking depends on a threshold called load delta indicating the difference between the load on local replica and the load information on remote replica. Load delta determines the triggering of redirection and controls the aggressiveness of redirection. It is determined by an exponential weighted moving average as a threshold to adapt for different workload patterns.

In contrast to the centralized approaches in both Racksched and R2P2 [ZKC$^+$, KPG$^+$] piggybacking information load on requests is not a sufficient mechanism for learning about

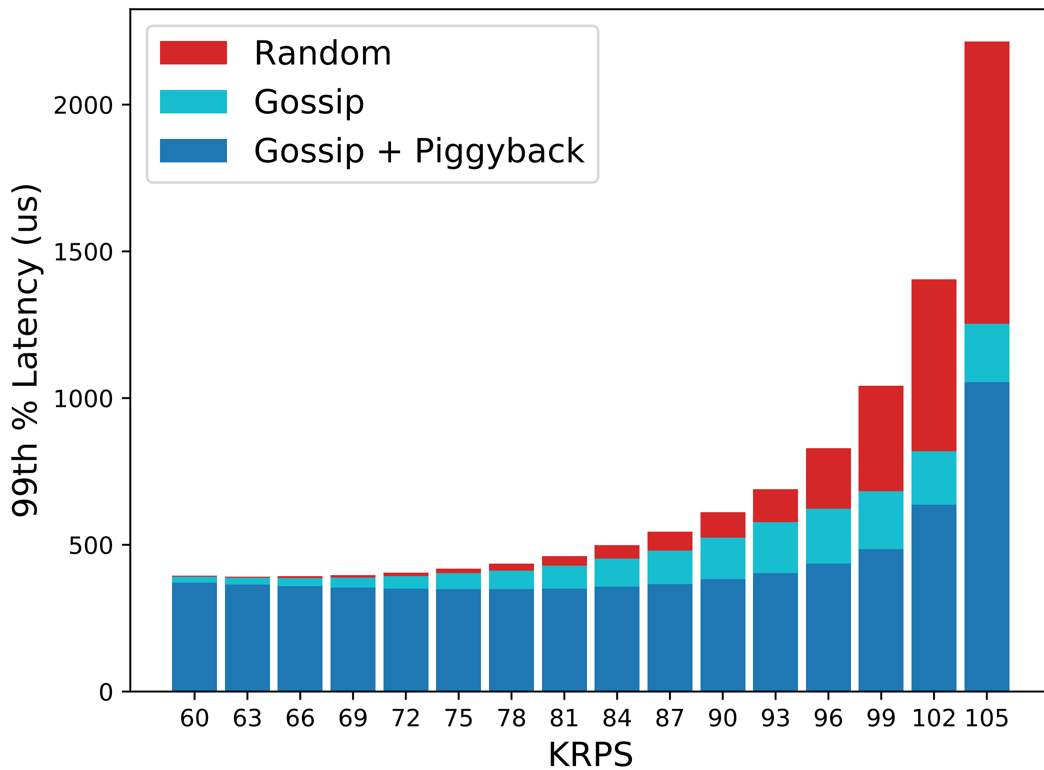


Figure 2.5: Performance breakdown of gossip and piggyback mechanisms. Lower values are better.

remote load. This is because our load balancer is decentralized, which means that our load balancers do not see load information updates from every request. Unlike our logarithmic gossip mechanisms, it provides no guarantees about its operational bounds. Using gossip messages exclusively provides no guarantees that any specific server will have information propagated to it as only the server which is redirected to receive fresh information.

In the case of these centralized solutions, each request returns some information to the scheduler. In the distributed case, there is no liveness guarantee with regard to redirections, and indeed, information can become arbitrarily stale. We therefore consider our piggyback algorithm to opportunistic, only aiding in the common case when load is low, but when making precise redirections will still improve throughput and provide lower latency.

Piggybacking load has the advantage that it is responsive proportional to the request

rate of the system. As the number of requests per second increases so to does the rate at which information is spread between ToRs. Furthermore, it has the advantage of introducing a small amount of overhead. Rather than incurring the cost of an entire load information update, this only adds a few bytes to a custom header injected at the ToR.

Figure 2.5 shows a performance breakdown of the two pillars of Daronpon's design: logarithmic gossip and piggyback compared to a baseline of performing random selection on the client alone. At low request rates the logarithmic gossip does not provide much of a performance benefit in relation to the piggyback method. However, as the request rate, and variability, of the system rises (e.g. to 102 and 105 thousand requests per second), the logarithmic gossip provides the majority of the gains as it detects the peaks which increase tail latencies the most.

2.4 Implementation

We deployed Daronpon on the AWS cloud with instances hosted in VMs connected via Elastic Network Adapter (ENA) virtual NICs. These instances use the Data Plane Development Kit (DPDK).

Components: Our deployment consists of three components: DPDK ToRs, DPDK clients, and servers with default Linux networking stacks relying on UDP for application messaging. DPDK is a kernel bypass networking library which allows for high throughput and low latency packet processing in user space [DPD21]. DPDK ToRs emulate ToR switches with limited latency overhead ($<$ 1 microsecond). Ideally, we would implement our algorithm on P4 switches, however to our knowledge no cloud providers allow for customers to offload custom programs to programmable switches at this time. Considering that programmable P4 switches have not been widely deployed in datacenters, Daronpon could also be offloaded to SmartNICs, reducing packet processing latency through dedicated hardware without interrupting the main CPU cores We implement our clients using DPDK for lower latency and precisely controllable request rates.

These traits are important as AWS's ENA NICs do not have hardware timestamping available to users. Our DPDK clients can generate hundreds of thousands of requests per second with a single virtual core. These UDP-based servers represent services relying on the standard Linux networking stack. We choose to use UDP as the transport because it allows us to redirect request atomically without connecting multiple packets together and redirecting as a group, though this could be supported as future work.

Daronpon DPDK ToRs: DPDK ToRs use an arbitrary number of cores to forward requests/responses and a single core to gossip load information. Redirecting involves header manipulation, tracking load information using hashtable table lookups, and counter increment/decrement operations. The operations we used in these DPDK ToRs are carefully chosen to be simple, and within the capabilities of programmable switches to compute.

Custom packet headers: Daronpon appends a custom header after the IPv4 UDP header. The header consists of a unique request ID for each request, which is used to track lost packets and measure end-to-end latency. A service type field differentiates services, e.g. Memcached and RocksDB. Additionally, the IP and ports describe addresses of replicas which implement other copies of the service. We assume that clients know the server replica addresses by asking the cluster-level replication manager, e.g., Google's Slicer or Facebook's Shard manager [fac20, AMH$^+$, KXH$^+$]. Gossip messages are also based on UDP, and load information is appended after the UDP header. The header contains a list of server addresses, load counters, and its corresponding service types for all servers under a ToR switch.

2.5 Evaluation

We evaluate Daronpon in on the Amazon Web Services cloud (AWS `us-west-2` region) using `c5n` instances. In this section, we describe the experiments and the resulting conclusions.

2.5.1 Experimental Setup

We use 6 instances as servers, 6 as clients, and 3 as software ToRs. All instances are placed in a cluster placement group for predictable low latency. The mean RTT latency between every instance is 50 microseconds. In this setup, we configure 6 servers, 6 clients, and 3 software ToRs. This configuration emulates a datacenter network of 3 ToRs that each ToR has 2 services running underneath it. Services are deployed to servers using the stock Linux networking stack. While this does incur higher latencies as compared to DPDK-based kernel bypass stack, we note that it is representative of many datacenter applications.

We implement the *random* and *Power-of-2 choices* as the baselines. Both of them select the replica of the service on the client side. *Random* baseline selects a replica of a service regardless of any load information. *Power-of-2 choices* for service selection is another baseline that selects a replica of a service based on load information available to the client. The power-of-2 choices [Mit01] randomly chooses two random replicas of services and picks the one with lower load out of the two ones. The load information used for service selection is stale when it reaches the client because the actual load may have changed. The load information is obtained with active probing the servers or from the response of a previous request. We compare Daronpon to random and power-of-2 choice in the experiments.

2.5.2 Workloads

We evaluate our load balancing approach with request-response based applications, in which the time spent on the server is emulated based on different statistical distributions. In our evaluation, we generate requests according to two statistical distributions, one of which generates application-level requests and is run on DPDK-based clients, and another which generates emulated service times. Clients generate requests based on open-loop Poisson arrival using the standard random library. Servers distributions are split into different distribution categories, each

of which has its own separate parameters. These include constant time, bimodal, and exponential distributions.

Each server is loaded with an equal number of requests (in expectation) drawn from common request rate distributions, as described next. Our goal in using these distributions is to demonstrate that using outstanding requests (blind to the underlying service distribution) works in general, without the need to tune our load balancer for each application. Both our gossip and piggyback mechanisms are enabled in each experiment. The lower threshold on the gossip mechanism is initialized as 4.

Constant: In this configuration, all requests compelte in 25 μs on the server. On the server, a work thread busy-polls the time until 25 μs has passed to emulate application-level service times. This type of workload is indicative of many highly tuned key-value stores with strict SLOs [Mem21, roc21]. The servers experience additional latency overheads from the Linux networking stack. Our choice of this constant latency is intended to be representative of a performance-tuned microservice which performs a fixed amount of work per request. To provide a sensitivity analysis to this choice of constant, Section 2.5.5 provides an overview of Daronpon's performance across constant service times

Exponential: To generate an exponential distribution we use the standard C++ random library. We set the mean to 25 μs with a standard distribution of 40,000. These parameters generate tails of up to 400 μs which is indicative of many applications that may be subject to blocking, such as occasional writes to disk or the invocation of a blocking RPC to another machine.

Bimodal: Our bimodal service times are distributed into two categories. 90% of the requests take 13 μs and 10% are 130 μs. We choose this distribution as the mean value is close to 25 μs. This distribution is aimed at emulating longer and less frequent tasks such as writes and scans in certain key-value store workloads or even garbage collection events in the runtime.

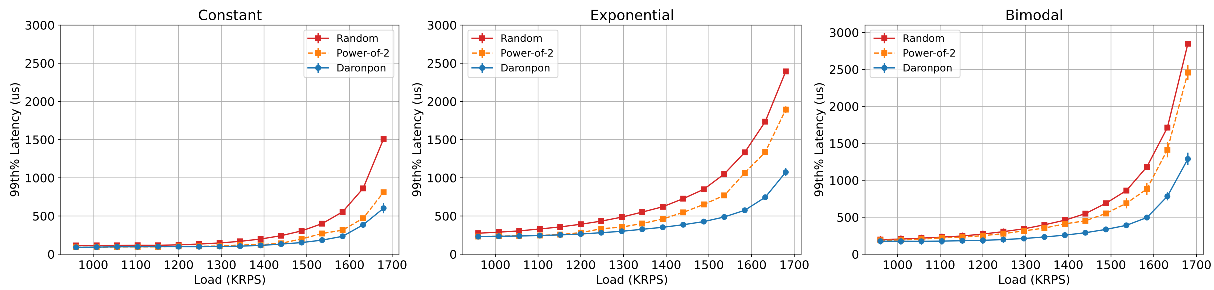


Figure 2.6: 99th percentile latency improvements on three common service distributions (Constant, Bimodal, Exponential). Each server is provisioned with homogeneous processing power.

2.5.3 End-to-end experiments

We test the effectiveness of our load balancing technique by running it against random replica selection alternatives on the aforementioned workloads. Figure 2.6 shows the relative performance gains across these workloads at the 99th percentile latency. In this configuration, each of the servers has identical processing capacity for each service. We consider this idealized and homogeneous configuration because of its simplicity.

Daronpon demonstrates the most relative gain over random when skew in the workloads is common. At high loads, request variation occurs at higher degrees for multiple reasons. First, the Poisson arrival process on our clients has a higher probability of generating bursty sequences of events at higher load. Second, hypervisor and NIC hardware on AWS contributes to the bursty arrival of requests because of the underlying batching behavior. These forms of burstiness is largely out of our control as we do not have direct access to AWS's hardware configuration. Finally, as rates increase, more batching happens in the Linux kernel networking stack. This leads to sharp decreases in the number of outstanding requests.

Daronpon demonstrates observable benefits in the bimodal and exponential distributions and the long service times cause significant and frequent queuing on the end hosts. Daronpon's logarithmic gossip design reacts quickly to these changes under load and steers requests away from the servers which are running long average request times. In our homogeneous experiments,

the average value for number of outstanding requests is four in our constant distribution, with frequent peaks of up to 80 outstanding requests when Daronpon is disabled.

Constant: Figure 2.6-left shows our approach provide benefit on tail latency starting at 1248 Krps (kilo-requests per second). This constant workload gives Daronpon the fewest opportunities to load balance effectively as the random distribution of requests, each with a static service time, should be approximately even. In this distribution, the benefit is found in the load fluctuations. At high request rates, other mechanisms such as Linux's request batching, have more of an effect on queuing. At the highest request rate that a single vCPU core can handle, the latency improvements of Daronpon over the random and power-of-2 choice baselines are 0.60 $\times$ and 0.25 $\times$ at the 99th percentile with the highest request rate. The improvement over power-of-2 choices is less because the evenly distributed requests given the same constant service times. This remediates the stale information and makes the performance of power-of-2 choices relatively close to Daronpon.

Exponential: Figure 2.6-middle shows the throughput and latency gains on an exponential server distribution. The benefits are most noticeable here as the exponential distribution leads to the fastest disparity in load. In this case, a single request on the exponential distribution can lead to significant queuing on a given server. Daronpon's latency improvements over the random and power-of-2 choice baselines are 0.54 $\times$ and 0.43 $\times$ at the 99th percentile.

Bimodal: In Figure 2.6-right, we see the disparity between random service selection and Daronpon. The service time dispersion provided by bimodal distribution causes noticeable request queuing when a long service time request occupying a service. Logarithmic gossip messages are ideal in this case as they quickly react to the skew in load. Further, in this distribution the probability of finding an under-utilized replica is relativity high. The difference of 99th percentile latency between random and power-of-2 choices shrinks compared to the case of the constant and exponential service time. The 99th percentile latency of power-of-2 choices moves closer to random as the queuing penalty of selecting the congested replica could be as large as 130 μs.

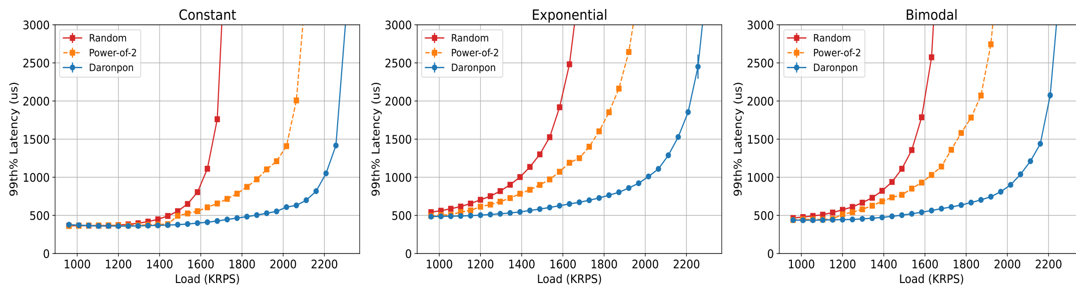


Figure 2.7: Throughput and latency improvements with skewed processing capacity in a heterogeneous server configuration. Daronpon scales linearly with with the aggregate processing capacity available.

2.5.4 Heterogeneous server configurations

The placement of replicated services is subject to the cluster scheduler [fac20, AMH$^+$, KXH$^+$]. The placement of individual services may be tightly coupled to a single rack, or distributed to multiple racks. Additionally, not all replicated services may be configured using identically powered instances. Some may be provisioned with different core counts, memory, and potentially different OS versions. Finally, should rack-level scheduling be utilized such as Racksched or R2P2, the individual rack level throughput may differ below the operating domain of Daronpon.

To show the generality of our approach in situations where replicated applications differ in their throughput capabilities, we configure one out of our three servers to run using three times the processing capacity, i.e., 3 $\times$ number of CPU cores rather than one. In this setup, clients otherwise operate identically to the homogeneous configuration.

Figure 2.7 shows the performance gains from enabling Daronpon on our heterogeneous testbed. In this test, the server with twice the processing power of the other processes requests twice as quickly. Our random client takes no measure of queue depth, and therefore does not adjust to this excess compute power. Its performance is only marginally better in this case, as the request which it probabilistically sends to the doubly provisioned server are processed more quickly.

Daronpon apportions load to servers in precise relation to their processing power. Our results from this test show that Daronpon exhibits good scaling with more computation in this configuration. Daronpon is able to process 1.4 $\times$ in the case of the constant service time. When running in a heterogeneous configuration, the proportion of gossip and piggyback request remains approximately the same as in the homogeneous case. The gossip mechanism is triggered periodically as the request on the faster servers drains below its current threshold, however this periodic variance occurs on the same order as the natural fluctuations in load. Redirections also occur at approximately the same rate, however they are almost entirely directed at the over provisioned server.

Ideally, we would demonstrate the scalability of Daronpon across many racks with more services. We see this heterogeneous result as a proof of concept that our approach can scale approximately linearly with available processing power. We expect this result to hold as our approach is similar in style to theoretical approaches for distributed load balancing which are proven to provide linear scaling while using incomplete local information to balance load [VKO20].

2.5.5 Gossip and Piggyback Overhead

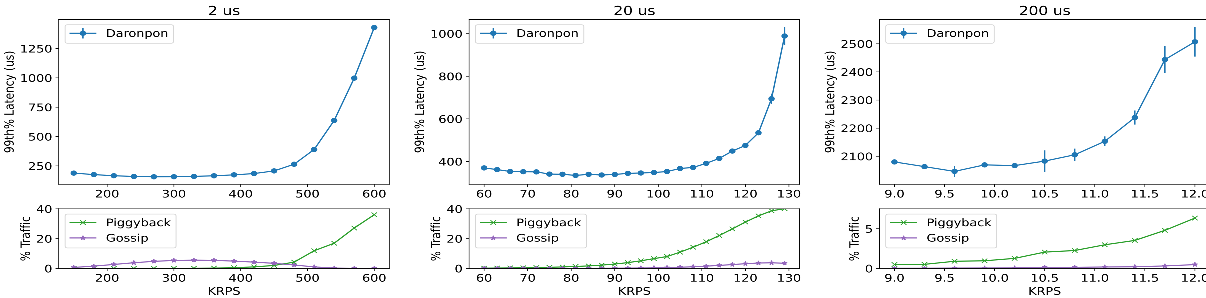


Figure 2.8: Service times across three orders of magnitude (2us, 20us, 200us). Daronpon provides relative improvements with similar overheads in terms of piggyback and gossip messages at each service time.

To investigate Daronpon's sensitivity on service time and the related overhead, we vary

constant service times by three orders of magnitude across three experiments: 2 μs, 20 μs, and 200 μs. This helps us to understand the overhead of our logarithmic gossip mechanism, the redirected proportion of messages with piggybacked load information under different service times. First, constant service times stress the servers more because it makes the servers to process more requests per second. With more incoming requests, the number of outstanding requests on the switches is also higher. Second, we set up aggressive redirection that does not require the load of a remote replica to be lower than that of the local one. In this experiment, the load delta between remote and local replica is set to 0, which means that the minimum replica is always selected regardless of the performance impact. Choosing delta above 0 resulted in a lower number of redirections, with a delta of 8 resulting in piggyback messages being generated for only 6% of all requests. The aggressiveness of redirection is a trade-off of performance, resulting in lower tail latencies at the cost of the bandwidth.

Figure 2.8 shows the tail latency and the corresponding overhead of Daronpon. At 2 μs (Figure 2.8-left), the 99th percentile latency increases wit higher request rate as expected. Interestingly, Daronpon gossips more frequently between 200 and 400 Krps and shows a bump peaks at 330 Krps. The frequent trigger of gossip is because gossiping thresholds are based on power of two numbers, e.g. 2, 4, 8, 16, etc. Statically, gossiping thresholds are more frequently met when the number of outstanding requests is lower. For example, an increase of outstanding requests from 1 to 8 triggers 3 gossip messages while an increase from 8 to 17 triggers a single message. At ranges of 64 and above, Daronpon gossips when encountering bursts of incoming requests. Daronpon gossips less frequently during higher request rates when piggyback takes over gossip as the main mechanism to propagate load information.

At 20 μs (Figure 2.8-middle), Daronpon operate with the maximum gossip overhead reaching no more than 3% at peak system load. On the other hand, the number of piggyback messages grows with the request rate. This is because at higher rates more bursts occur, and thus the opportunities to load balance increase. Near peak load, piggyback messages reach

approximately 40% with the highest request rate, the redirected packets do incur additional bandwidth usage. The additional uplink bandwidth usage, as mentioned in Google's Jupiter-Rising paper [SOA+], does not stress the most bottleneck links that are all downlink to servers and to ToR switches.

At 200 μs service times, Daronpon operates with lower traffic on piggyback as the service time is longer. With longer service time, it provides a larger window for load information to propagate. The overhead from gossip remains low.

2.6 Discussion

Scaling: In production cluster, managers determine the application service replication factor. This factor is determined dynamically by monitoring system load, which can cause applications to scale up and down significantly on the order of hours. Daronpon's gossip broadcast could be inflated by the replication factor, and therefore large replication counts (e.g. in the 100s) present a potential bottleneck. Proper placement of services is possible to constrain the overhead of gossip within a specific sets of racks. The placement can also enable piggybacking to carry load information for multiple services on a single redirected request.

Emerging Topologies: Daronpon works on any datacenter topology which uses ToRs, or virtual ToR-like abstractions (such as our DPDK software middlebox switches). Emerging network designs, e.g. Jellyfish [SHPG] and Xpander [VSDS], are supported under our architectural assumptions. Some network designs have asymmetric latencies between servers. While this may cause some racks to propagate information which is more stale, we do not see this as a limitation of our techniques as our AWS testbed has latency variations on the order of a few microseconds. Daronpon's redirection piggyback and load gossip can take a variable amount of time to propagate load information to other ToRs, but our load balancing design is similar to theoretical techniques prevent to be effective even with stale information [VKO20].

Multi-packet requests: Daronpon operates on IPv4 UDP packets. It does not bake reliable transport into the design and assumes that retransmissions of lost requests are handled by the application. To support reliable transmission such as TCP for Daronpon, it is required to track flow-level specific state in the network and ensure that once a replica is selected for a request, no further selections occur on that flow. This is interesting but also introduces additional complexity. It is left for future work.

Failures: Our approach does not explicitly handle failures. If a server fails, Daronpon will automatically load balance around it, as requests issued to the failed server will not respond, and thus the queue will grow indefinitely. If a Daronpon ToR were to fail, that rack becomes partitioned. We leave the detection of ToR failure in this case to future work.

Application Heterogeneity: Daronpon assume that all replicas are created equal, in that any request can be sent to any replica. In the case of replication systems with various roles, such as leaders and followers [OO], additional application-level information would be required to only perform replica selection on requests which do not have a specially configured destination.

2.7 Future Work

We've used DPDK as a software ToR for simplicity. The latency between our software switches is approximately 25 μs on AWS. This overhead is caused by the underlying network at AWS. These overhead may have reduced the benefits of Daronpon's gossip and piggyback based mechanism as every microsecond of delay diminish the value of the propagated load information. In the future, we would like to implement Daronpon on a programmable switch such as the Barefoot Tofino 2 [tof20]. We predict that with inter-ToR one-way latencies between 1 and 3 μs, Daronpon's load balancing decisions is very likely to be further improved. Also, thus far we've explored microservices which we expect to have service times on the order of tens of

microseconds. Recent work in persistent memory has demonstrated that remote storage is now accessible on these same time scales, and so we believe that replicated storage might benefit from Daronpon's approach of distributed load-balancing.

2.8 Conclusion

In this work we present Daronpon, a microsecond timescale load balancer for replicated data center-wide applications. We have developed a novel hybrid gossip and piggyback scheme to keep ToR switches up to date with load information on each server with low overhead, and have demonstrated the effectiveness of this technique by comparing to random load balancing and demonstrating up to 2.1 $\times$ lower 99th percentile latency.

2.9 Sources for Material Presented in This Chapter

Chapter 2, in part, reprints material as it appears in a draft titled: "Daronpon: Datacenter-scale Sub-RTT Replica Selection for Low-latency Applications" by Shu-Ting Wang, Stewart Grant, Keerthana Ganesan, George Porter, and Alex C. Snoeren. The dissertation author was the primary researcher and author of this material.

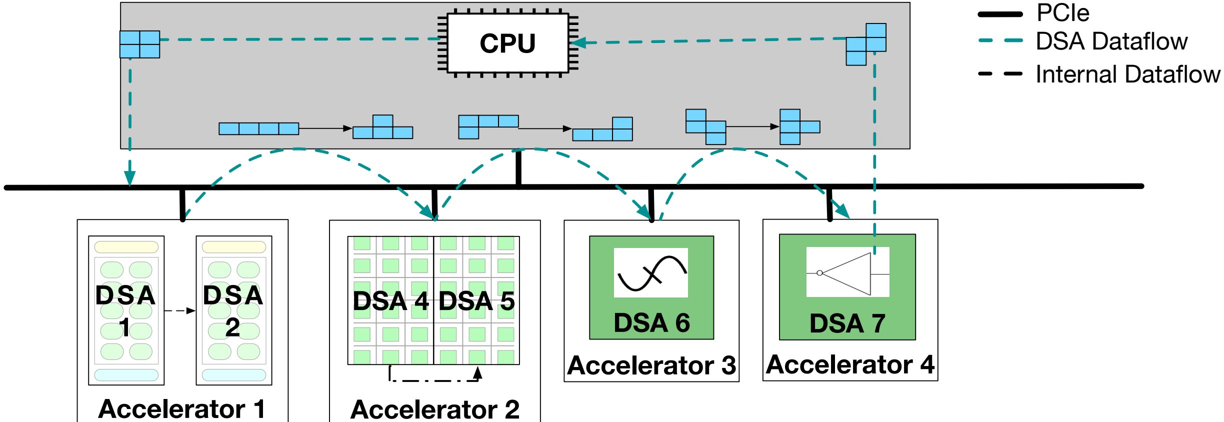
Chapter 3

Fianchetto: Accelerating Data Motion Across the Board

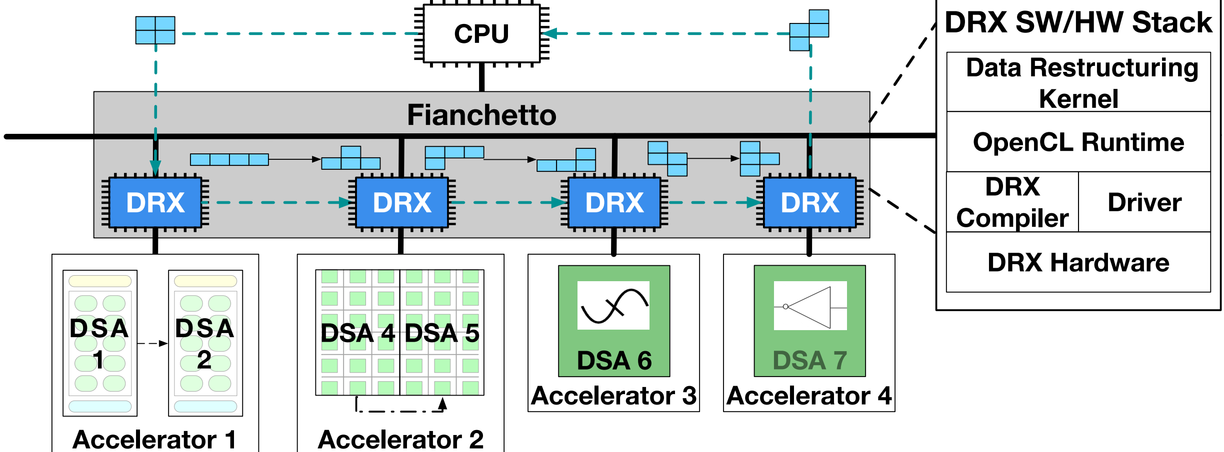
With the effective end of Dennard Scaling [DGR^{+}74], the dark silicon [EBA^{+}11, HFFA11] phenomenon has led to the development and adoption of Domain-Specific Architectures (DSA) or accelerators. With the Cambrian explosion of accelerators [CDS^{+}14, ZLS^{+}15, LCL^{+}15, DFC^{+}15, LDT^{+}16, ZDZ^{+}16, SWH^{+}16, SPM^{+}16, ACFM16, MPA^{+}16, SFM17a, SQLC17, PRM^{+}17, JYP^{+}17, SFM17b, KSK18, LKK^{+}18, CYES19, SCV^{+}19, GYP^{+}19, GLS^{+}20, YDH^{+}20, GAK^{+}20, QSK^{+}20, LWL^{+}21, LLKB21, HWS^{+}16, LKY^{+}17, YZL^{+}18, MBA^{+}18, ZZW^{+}18, SZQ^{+}18, ZMTC18, BGP^{+}19, MBS19, RAGG20, ZLJ^{+}21, RAAGG21, ZZL^{+}21, TBD18, FSZ^{+}18, CGH^{+}18, BEHL^{+}19, NRB^{+}19, HLL^{+}19, CKB^{+}20, KIMR20, HML^{+}20, LGY^{+}20, FWO^{+}20, HMSA^{+}21, MGPM^{+}22, CKL^{+}22, KGP^{+}13, MFJQ^{+}16, SMLE18, LZT^{+}21, NPB^{+}21], it is fitting to consider the current cadence of the architecture design as the golden age of accelerators. Amazon Web Service (AWS) [awsa, awsb], Microsoft Azure [PCC^{+}14, CCP^{+}16, FOP^{+}18, mica], and Google Cloud Platform (GCP) [JYP^{+}17, JHYA^{+}21, RSC^{+}21] as the three providers of public cloud recently started offering accelerator equipped instances.

The offering of accelerator equipped instances is the result of market push toward

hardware-accelerated compute-intensive applications such as genomics, content streaming, recommendation systems, virtual reality, data analytics, etc. Such applications often cross the boundary of multiple domains, each of which can be potentially accelerated with its own domain-specific architecture (DSA). These applications would maximally benefit from the DSAs in the cloud only if all the domains are accelerated and not just one.



(a) Current multi-accelerator systems.



(b) Multi-accelerator systems with Fianchetto.

Figure 3.1: Current multi-acceleration systems rely on CPU for accelerator chaining. (a) shows a system with four heterogeneous accelerator cards. The CPU needs to intervene in the communication between accelerator cards. This involves data copies from system memory to accelerator memory and non-trivial data transformations. (b) The proposed Fianchetto framework removes the CPU from the data path of multi-acceleration. Fianchetto delivers the performance of a monolithic accelerator while offering the composability and programmablity of the baseline system.

To enable heterogeneous cross-domain multi-acceleration, there is an essential need for cross-stack solutions for accelerator chaining to enable intimate communication between different DSAs, each of which is responsible for accelerating a part of a single application. This chapter sets out to explore a heterogeneous cross-domain multi-acceleration datacenter that harvests the recent initiative towards democratizing hardware design and enables the vision of a *sea of accelerators* [JPOB20, Tay18, PGW+20, CQS+22, GKK+23].

In a cross-domain multi-acceleration system, a chain of DSAs is created, where each DSA accepts inputs in a specific data structure and produces outputs in another data structure. The accelerator chaining currently needs to involve the system CPU (Figure 3.1(a)) for restructuring and then exchanging data between different DSAs to run a single application. This restructuring usually involves reshaping and reformatting the output of one DSA to match the input of the next.

We refer to the data restructuring and communication overhead of executing a single application using a number of different DSA as *data motion* overhead. Using the CPU for data motion requires frequent copies between the host and the DSA memory. Moreover, because the overhead of data restructuring between DSAs exacerbates with the number of accelerators, the CPU quickly becomes the performance bottleneck at scale.

To address these challenges, we propose Fianchetto to accelerate data motion by integrating a programmable Data Restructuring Accelerator (DRX) with each DSA. DRX offloads data restructuring computation from the CPU back to a specialized engine near the DSAs. Fianchetto illustrated in Figure 3.1(b) removes the CPU from the data path of accelerator chaining and gives the illusion of a monolithic but composable accelerator to the user application.

Fianchetto offloads the data restructuring operations to a scale-out programmable accelerator (DRX) while running the control plane on the CPU. DRX acts as a compute-enabled interface through which data moves between DSAs while DRX itself encapsulates a domain-specific accelerator. Although there have been efforts in offloading ser/des protocols to hardware [PGK+20, KLK+21], prior work has not considered acceleration and offloading of cross-

domain DSA communication, which enables efficient and seamless accelerator chaining.

We evaluate Fianchetto using five end-to-end applications, each of which was composed of kernels from different domains weaved together using data restructuring kernels. We evaluate the scalability, performance, and energy of various Fianchetto configurations with a baseline that uses the same accelerator but still executes data restructuring on the host CPU. Fianchetto provides on average 3.4× to 8.2× speedup on end-to-end latency, 3.0× to 13.6× improvements on throughput, and 3.8× to 5.2× improvements on energy consumption. The significant additional improvements over a baseline that itself maximally speedups an application using multiple DSAs show the emerging importance of data motion and restructuring as accelerators take the stage in datacenters.

3.1 Preliminaries and Motivation

Future datacenter computing landscape will deploy an ocean of accelerators, each purpose-built for accelerating different application domains. A mixture of CPU, GPU, with FPGA- and ASIC- based accelerator cards is already employed in today's cloud services [PCC^+14, CCP^+16, JYP^+17, FOP^+18, awsa, $JHYA^+21$, RSC^+21, awsb]. Such an accelerator-heavy datacenter [MKGT16, TVK^+20] breaks applications into several domains, each running on a domain-specific accelerator (DSA), possibly implemented on different accelerator cards. [1]

The current accelerator cards, unlike GPUs, do not have a well-supported system around proprietary interconnection and programming interfaces such as NVLINK and CUDA. Accelerator cards are developed by individual vendors using standard interconnection technologies (i.e., PCIe) and lack a standard interface to inter-operate with each other. The vendors implicitly assume that their accelerator is the only accelerator in the system. Therefore, as illustrated in Figure 3.1(a), two DSAs implemented on different accelerator cards rely on a CPU to communicate with each

[1] DSA and accelerator are used interchangeable throughout Chapter 3.

other following these steps: (S1) the CPU copies the output of the first accelerator to the system memory, (S2) the CPU transforms the output to the second accelerator's input format, (S3) the CPU copies the transformed data to the second accelerator's memory, and (S4) the CPU fires up the computation on the second accelerator. Note that often the CPU configures a DMA device to copy data from accelerator memory to system memory. The lack of an inter-accelerator communication standard necessitates excessive *data movement* and *data restructuring overhead* for performing non-trivial data restructuring operations on general-purpose cores.

We call the data movement and restructuring, *data motion* and develop Fianchetto framework to maximize the end-to-end performance of heterogeneous multi-accelerator systems. Figure 3.1(b) illustrates a high-level overview of Fianchetto. Fianchetto removes CPU from the data plane of multi-accelerator communication by offloading data restructuring to a programmable Data Restructuring Accelerator (DRX) integrated into the I/O periphery of each accelerator card. In the rest of this section, we motivate Fianchetto design by studying representative data restructuring operations and the overhead and scalability issues of performing data motion operations on a CPU in a multi-accelerator system.

3.1.1 Data Restructuring Operations

In this work we use five end-to-end applications that span multiple domains [APR20, SJB14, KdCL+20, micc, SOI+17, CMM+22] to quantify the inefficiencies of cross-domain acceleration in current multi-accelerator systems. Each of them has different domain-specific kernels and data restructuring requirements between the kernels. Specifically, Video Surveillance decodes input video streams into video frames and passes them to an object detection kernel [RF18]. Brain Stimulation receives electromagnetic signal input generated from a brain simulation model, processes it with FFT and data restructuring operations before outputs the data to reinforcement learning kernel [KdCL+20]. Personal Information Redaction decrypts privacy-sensitive text and uses a regular expression kernel to detect personally identifiable information and redact them

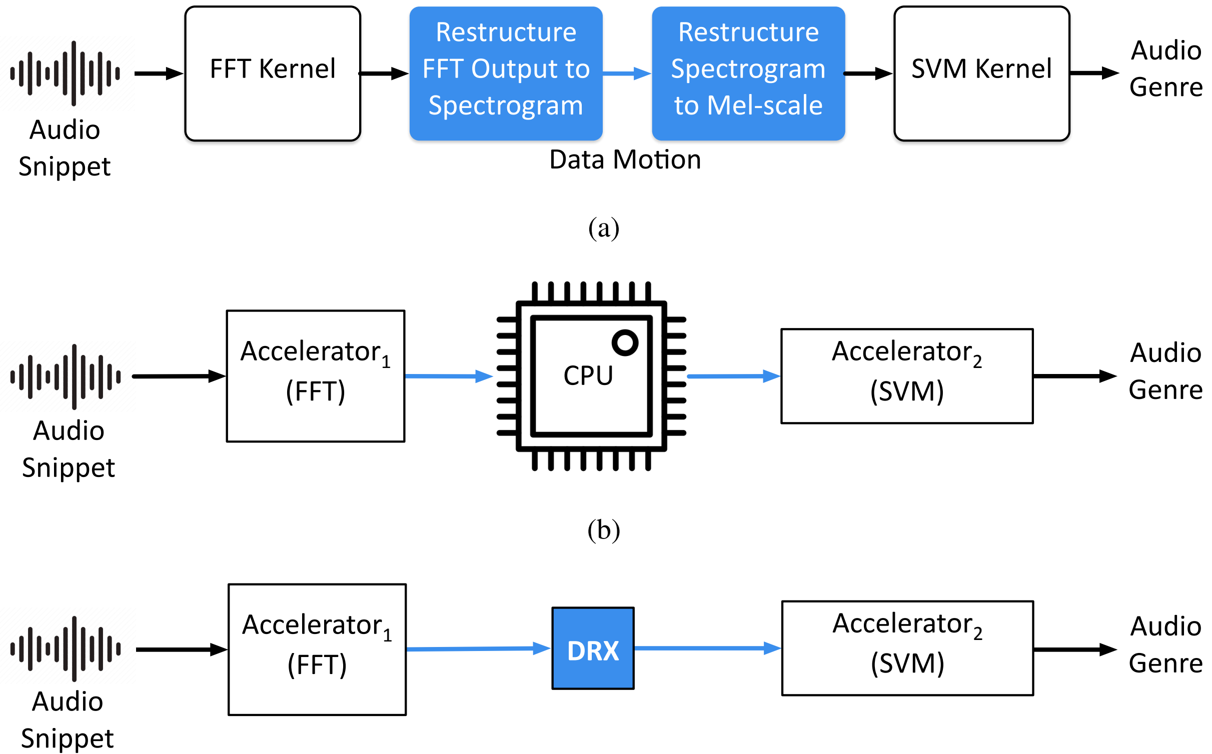
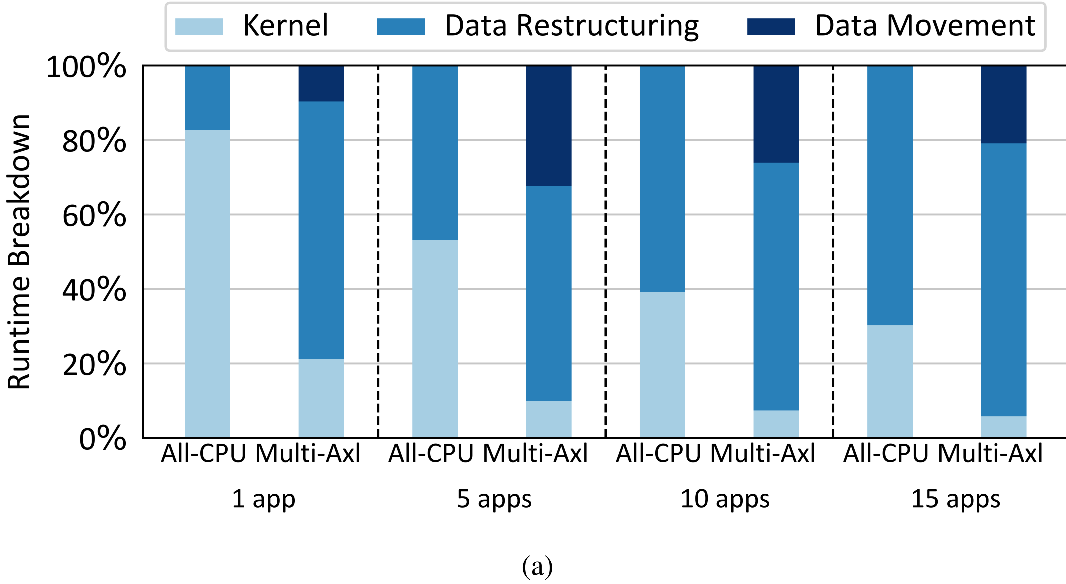


Figure 3.2: (a) Data motion stands between two application kernels, i.e., Fast Fourier Transform and Support Vector Machine, of an end-to-end application. (b) Data motion is on CPU and application kernels are on their corresponding accelerators (c) For Fianchetto, data motion is accelerated on DRX and application kernels are on their corresponding accelerators.

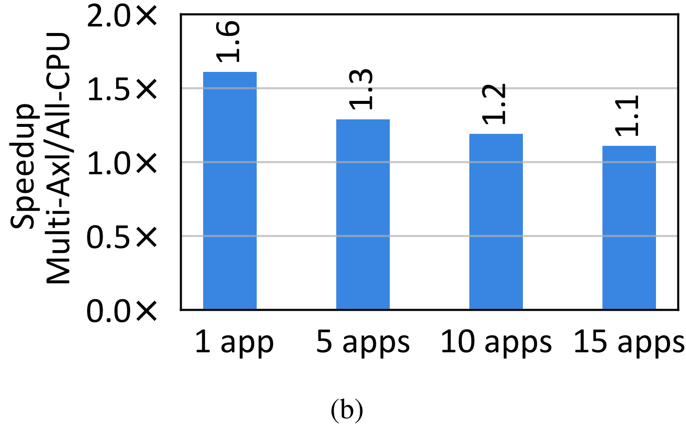
from the text with blanks [micc]. Database Hash Join decompresses database tables and hash joins the tables [SOI+17, CMM+22].

Figure 3.2(a) illustrates the end-to-end application pipeline of the Sound Detection application. As shown, Sound Detection is composed of two domain-specific kernels: (1) FFT kernel running short-time Fourier transformation for the input audio snippet, and (2) support vector machine kernel to decide the genre of the audio snippet. An intermediate *data motion* step is required for restructuring the output of the FFT kernel to the input format of the support vector machine kernel while copying the data from the output buffer to the input buffer. In this example, data restructuring requires generating a spectrogram from the output of FFT kernels and applying mel scale transformation to the spectrogram. The mel scale transformation maps the spectrogram

into mel-frequency bins which are closer to the human-perceivable scale.



(a)



(b)

Figure 3.3: (a) Runtime breakdown when running applications on CPU or multiple accelerator setup that uses CPU for data motion. (b) Multi-acceleration speedup and scalability are constrained by data motion overhead.

3.1.2 Data Motion Overheads

Figure 3.3(a) shows the geometric mean of the runtime breakdown for the five applications explained in Sec.3.1.1. We show the results for co-running up to 15 applications on the server while data restructuring is performed on the CPU. *All-CPU* configuration runs application kernels

on the CPU while *Multi-Axl* runs the application kernels on the DSAs. Because each application consists of 2 domain-specific kernels, 15 application setup runs on 30 DSAs. As Figure 3.3(a) shows, in the *All-CPU* setup, the execution of domain-specific kernels accounts for up to 78.5% and on average 49.1% of the total runtime. However, the *Multi-Axl* setup reduces the runtime of domain-specific kernels, but at the same time amplifies the ratio of data motion within the end-to-end runtime. The ratios range from 71.3% to 97.1%, showing that data motion becomes the performance bottleneck under multi-acceleration.

Another important observation from Figure 3.3 is the poor scalability of current multi-accelerator systems when concurrently running applications on multiple accelerators. From a single application to 5 applications, data movement emerges as a bottleneck. The limited PCIe bandwidth of CPUs creates a bottleneck for data moving in and out of CPU for data restructuring operations as they cannot directly connect all accelerators concurrently. As the number of applications grows further to 10 and 15 applications, the CPU demonstrates its incapability to keep up with the increased concurrency of data restructuring operations though using 16 Xeon cores. Such a bottleneck in data movement and data restructuring stifles the end-to-end speedup achieved by multi-acceleration at scale. As shown in Figure 3.3(b), accelerating application kernel while relying on the CPU for data restructuring achieves 1.4$\times$ and 1.1$\times$ end-to-end speed up for 1 and 10 applications, while the geometric mean of per DSA speedup is 6.5$\times$.

The above results demonstrate the untapped potential of multi-acceleration with ideally accelerated data motion. This significant performance difference between end-to-end and per-kernel speed-up stems from the following **I**nsights: **(I1)** Using specialized accelerators reduces the runtime of kernels significantly, shifting Amdahl's bottleneck towards data motion. **(I2)** Host CPU engagement imposes inevitable data communication with accelerators, adding the cost of data movement on top of data restructuring. **(I3)** Heterogeneity in the architecture of both accelerators and CPU demands additional data type conversions and layout transformation on top of discussed data restructuring, further amplifying the cost of data motion. Heeding these

insights, this work makes a case for accelerating the data motion.

3.2 Fianchetto: Accelerating the Data Motion

Multi-acceleration in Figure 3.2(b) represents the current system design using CPU for data motion. This design requires data to move through the CPU for restructuring the output of one accelerator before the data can be used by the next accelerator. In this chapter, we propose Data Motion Acceleration as illustrated in Figure 3.2(c) to facilitate data motion between heterogeneous accelerators. Fianchetto accelerates data restructuring and bypasses CPU for data movement between accelerators via integrating the purposefully-built Data Restructuring Accelerator (DRX) into the system. Realizing Fianchetto requires synergistic design considerations at the following levels:

- **DRX Placement.** An important design decision in Fianchetto is the location of the DRX. The placement of DRX impacts the data movement and the overall system design. We consider three different placements for DRX: integration on the CPU, standalone PCIe-attached card, and per accelerator bump-in-the-wire placement.

- **Specialized Hardware Acceleration.** We need to design DRX to be programmable and support a range of data restructuring operations. As Figure 3.3(a) shows, data restructuring accounts for 57.7%$\sim$73.2% of end-to-end runtime, therefore efficient execution of data restructuring is critical for multi-acceleration.

- **System Integration and Programmability.** To minimize data movement, the CPU should be removed from the data path of accelerator-to-accelerator communication. However, the control plane should run on the CPU, otherwise, it requires a completely new programming interface that stifles interoperability of Fianchetto across arbitrary accelerators. In Sec.3.5 we explain the current programming interface of multi-accelerator systems and how Fi-

anchetto only offloads the data plane to the hardware without changing the current control plane.

In Sec.3.3 we explore various placements for DRX and show that tight integration of DRX and accelerators in a bump-in-the-wire fashion minimizes the data movement and delivers the best performance and energy efficiency at scale. Next, we demystify the data restructuring operations in Sec.3.4 and introduce a programmable accelerator specialized for the data restructuring domain. Lastly in Sec.3.5, we discuss the runtime and kernel drivers that coordinate the offload of data restructuring operations to bump-in-the-wire DRX while still running the control plane on the CPU.

3.3 DRX Placement

The key design considerations in designing Fianchetto are the placement of DRX and interconnection between DRX, accelerator, and CPU in the system. Since Fianchetto is to enable interoperability between accelerators designed by different vendors, DRX's interconnect should be standard and well adopted. As such, the current incarnation of Fianchetto considers PCIe as the standard interconnect to connect accelerators to CPU and DRX. PCIe is a well-established standard of interconnect and serves as the basis for future interconnects such as CXL [cxl].

The placement of DRX ideally should (1) scale with the capacity of associated accelerators, (2) avoid being the bandwidth bottleneck when accelerators transfer/receive data from it, and (3) minimize data movement as data movement is the main performance and energy bottleneck in today and future system [Hor14].

Integrated DRX into CPU. This configuration considers integrating DRX with the CPU as illustrated in Figure 3.4. The integrated accelerators become more common recently as Intel Sapphire Rapids, IBM z15, POWER9, and Telum offer them in their CPU products [Bis21, ABR^{+}20, LBB^{+}22]. Integrated accelerators are efficient in performing computation on the

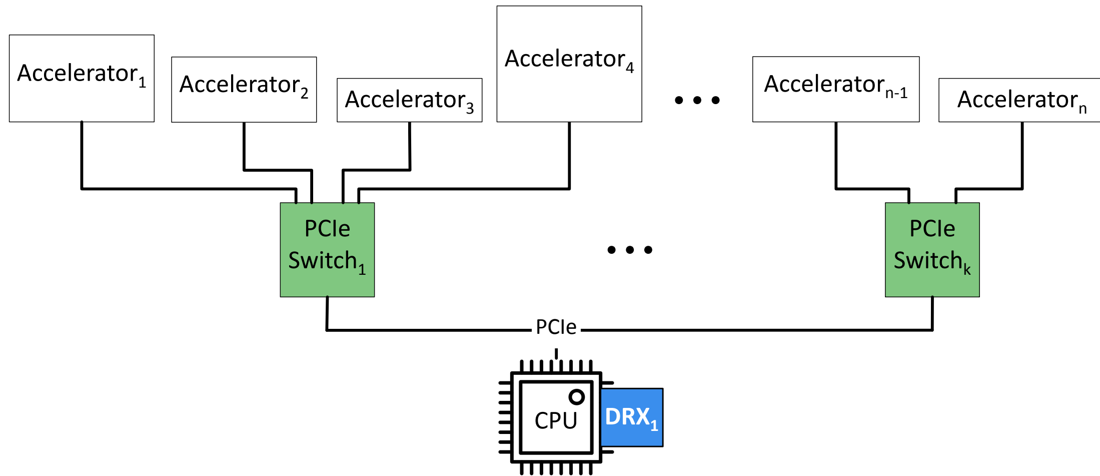


Figure 3.4: Integrated DRX.

data that is on the CPU chip. However, integrated accelerators are going to eat up the already limited CPU power budget [EBA+11, HFFA11]. Such power and thermal constraints limit the performance of integrated accelerators on the CPU.

Fianchetto considers a fixed power budget for an integrated accelerator and design an Integrated DRX to operate within this power limit [sup, ABR+20]. This fixed power budget limits the performance of DRX. As we will show in Sec.3.7, Integrated DRX becomes the performance bottleneck when scaling the number of accelerators to more than 8. Although integrating DRX using die-to-die interconnects like UCIe could alleviate the affect, integrated DRX still becomes the performance bottleneck with excessive data movement [NBB+21, ods, uci]. Moreover, Integrated DRX has the same data movement as the baseline CPU without DRX. Such design requires all accelerators to send their data to the CPU which makes the PCIe link connecting the CPU to the accelerators the bandwidth bottleneck when multiple accelerators use DRX at the same time. Such data movement is also the main source of system energy consumption.

Standalone DRX as a PCIe card. This configuration considers implementing DRX as a standalone PCIe card that is installed just like any other accelerator on a PCIe slot. Without using an external power supply cable, the performance of a single Standalone DRX PCIe card is limited by the PCIe power supply standard, which is 25 Watts, and its bus bandwidth. Nevertheless, as

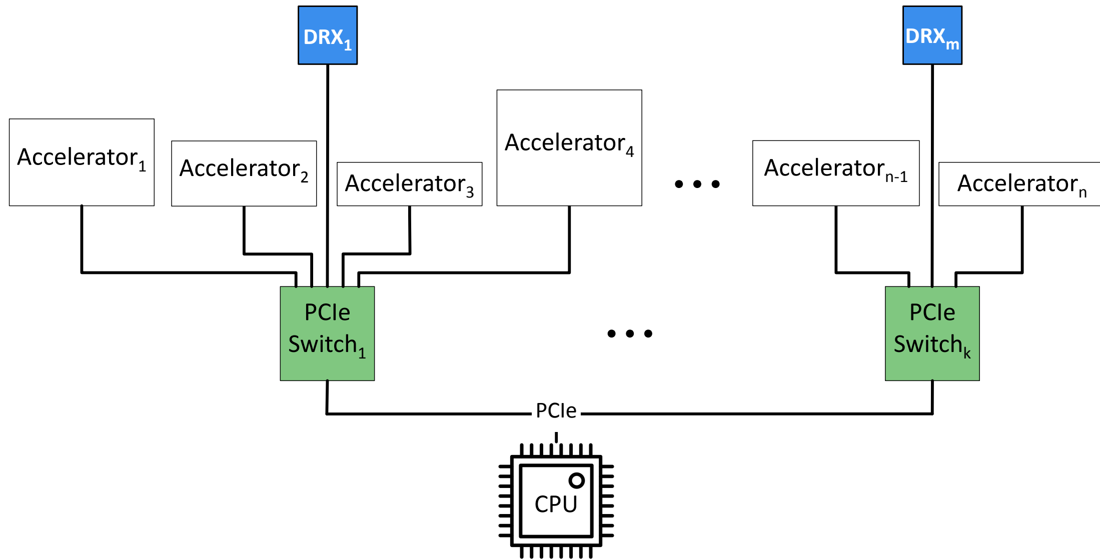


Figure 3.5: Standalone DRX. Number of DRX units in Standalone placement is configurable, and the illustration represents just one possible configuration.

illustrated in Figure 3.5, installing multiple Standalone DRX cards can scale DRX performance with the number of accelerators. However, this Standalone DRX still incurs bandwidth oversubscription as the PCIe link to a shared, Standalone DRX card can become the bottleneck. The bandwidth contention could be worse with multiple Standalone DRX cards are under the same PCIe switch.

Compared to Integrated DRX, a Standalone DRX has the potential to reduce the data movement if Fianchetto implements a point-to-point PCIe connection between DRX card and accelerator cards. This way, a Standalone DRX can localize the communication under the PCIe switch to which other accelerator cards are installed.

PCIe-Integrated DRX. This configuration integrates DRX onto a PCIe switch (Shown in Figure 3.6). Compared to a Standalone DRX, A PCIe-Integrated DRX saves a round-trip between DRX and the PCIe switch. However, PCIe-Integrated DRX requires DRX to operate at the aggregated rate of all downstream PCIe ports, which adds considerable hardware complexity. Also, computation on switches only permits limited memory usage and a limited number of instructions per packet [BGK+13, CFM+17, SAA+17, TDP+19]. This configuration requires

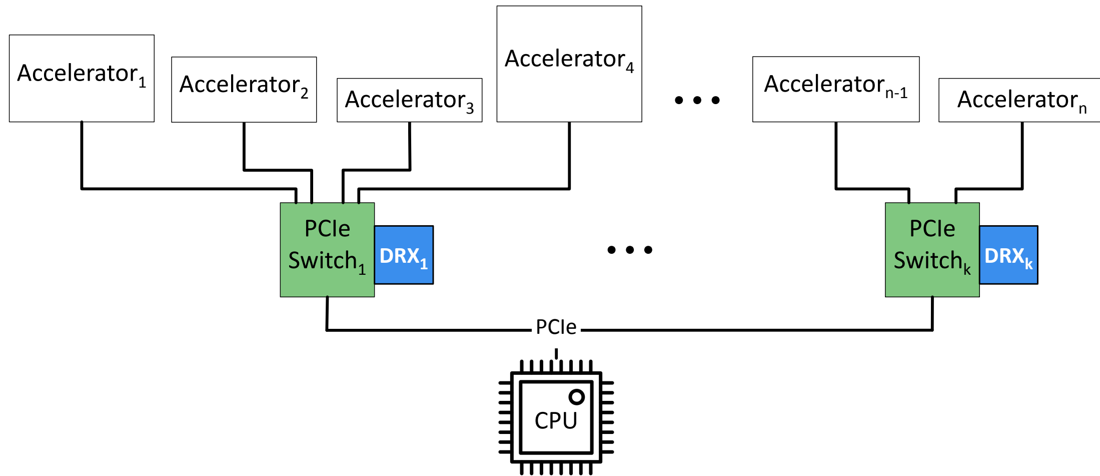


Figure 3.6: PCIe-Integrated DRX.

significant engineering effort to redesign the PCIe hardware and related software stack.

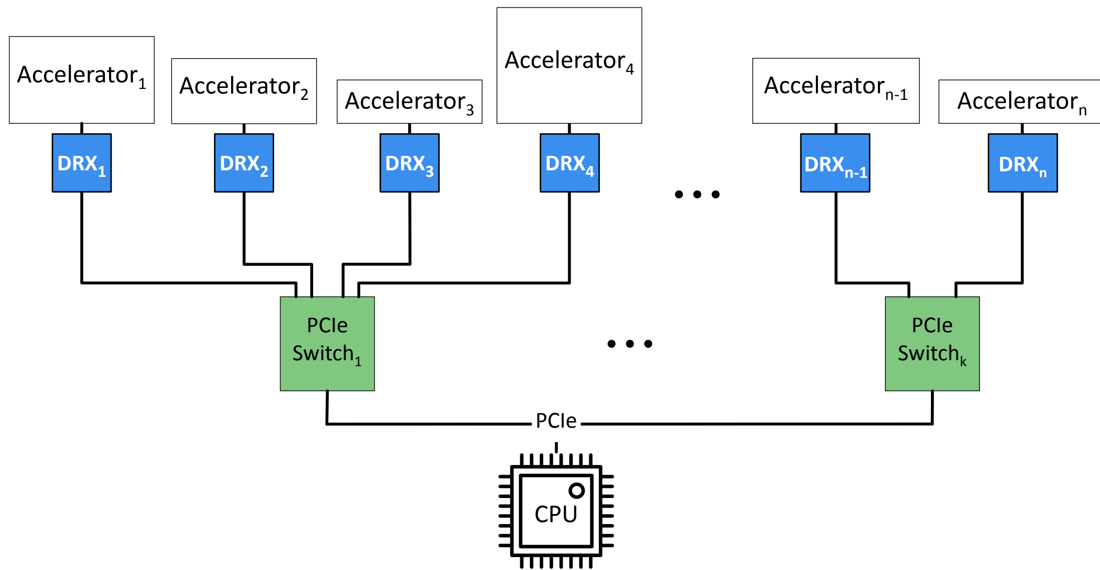


Figure 3.7: Bump-in-the-Wire DRX.

Bump-in-the-Wire DRX. Lastly, we introduce a Bump-in-the-Wire DRX configuration inspired by Catapult [PCC+14] that connects an exclusive DRX to each accelerator (Figure 3.7).

Bump-in-the-Wire configuration avoids overprovisioning of PCIe links and DRX resources for a multi-accelerator system and enables Fianchetto to scale with the hardware resources compared with the other configurations. More importantly, Bump-in-the-Wire DRX placement

reduces the data movement to a minimum when accelerators communicate with each other. Coupled with a programmable DRX that enables offloading of any data restructuring operation (c.f., Sec.3.4), Bump-in-the-Wire DRX serves as an option to build future scalable multi-accelerator systems.

3.4 Data Restructuring Accelerator (DRX) Design

As discussed in Sec.3.1, the CPU is not an optimal place to perform data restructuring operations. In this section, we first analyze different data restructuring operations by profiling their execution on the CPU. This analysis guides us in devising a programmable accelerator specialized for the data restructuring domain. Refer to Sec.3.6 for more information on the experimental setup.

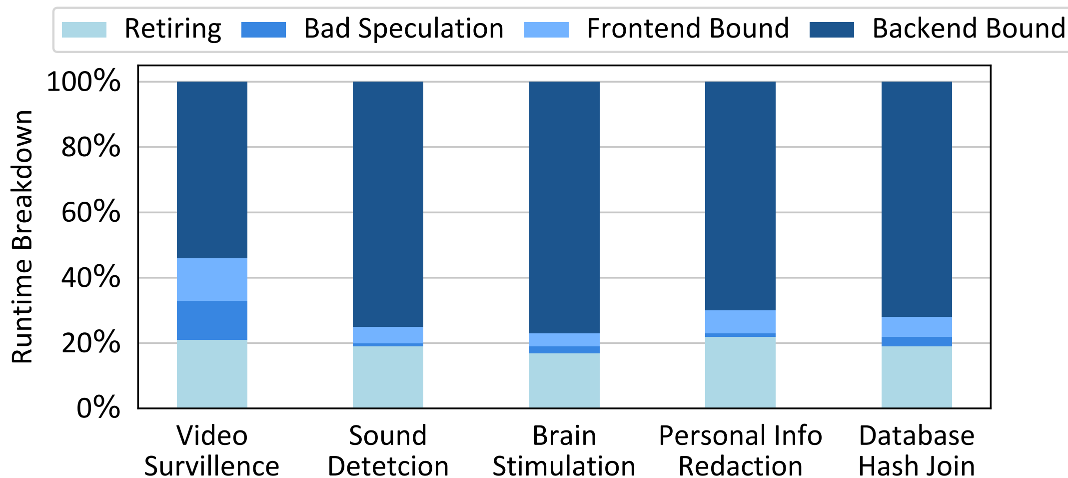


Figure 3.8: Top-down breakdown of stall cycles for data restructuring operations.

3.4.1 Data Restructuring Characterization

Figure 3.8 shows the top-down [Yas14] breakdown of stall cycles for data restructuring operations. We characterize data restructuring operations with the top-down analysis of Intel

VTune [intf] on an Intel Xeon Gold 6242R processor. The processor has the same microarchitecture as our testbed setup on AWS (See Sec.3.6 for details). Across different data restructuring operations, we see at most 12.5% Bad Speculation Bound and 14% Front-End Bound cycles. A deeper analysis of *Video Surveillance* reveals that this distinct behavior is linked to a higher number of branch instructions, resulting in a relatively larger number of cycles spent on branch re-steer and uOp cache switches. On the other hand, the Back-End Bound cycles range from 53% to up to 77.6% of total cycles. The culprit for Back-End Bound cycles is both the unavailability of functional units and misses in the data cache. 23.2% of Back-End Bound cycles are Core-Bound and 46% are Memory-Bound.

The profiling shows that data restructuring operations have low L1I cache Misses Per Kilo Instructions (MPKI). The average L1I MPKI for data restructuring is 2.3. As a reference, our measurements for online services from CloudSuite [noa] report an average of 7.8 L1I MPKI. Such low L1I MPKI suggests a small instruction working set for data restructuring operations that fit inside the L1I cache of the core.

The profiling results show that all data restructuring operations have a high degree of vector unit utilization. The data restructuring kernels use 100% of available vector unit capacity which is 256 bits wide AVX-256 on our servers. We also observe a high number of ephemeral threads that are spawned by the Intel Math Kernel Library while restructuring the data. The number of threads that are spawned while running the data restructuring operations is between 130 to 140. These threads operate on the data in parallel and illustrate the high data-level parallelism and inefficiency of CPUs in executing the data restructuring operations.

3.4.2 DRX Hardware Architecture

We use the above insights to design a programmable DRX that specializes in the data restructuring domain. The main observations driving DRX design are the abundance of data-level parallelism, streaming access pattern, and non-trivial operations of data restructuring. Figure 3.9

overviews the architecture of DRX hardware.

DRX uses a decoupled access-execute architecture that consists of a programmable front-end specialized for walking over multi-dimensional data structures, and a configurable number of interleaved vector processing units dubbed Restructuring Engine (RE) in the same pipeline. It also includes a Transposition Engine for data transposition operations and a programmable Off-chip Data Access Engine for off-chip load/store which also houses a DMA engine that initiates data movement with other accelerators. For evaluation, we configure the DRX to contain 128 lanes of RE, a 64KB instruction cache, a 64KB data scratchpad, and 8GB of DDR4 DRAM. A DDR4 3200 memory channel sustains ∼25GBps, therefore DRX implements a single DDR4 channel to match the bandwidth of an x8 PCIe Gen 4 link.

DRX ISA. The DRX ISA and hardware architecture are optimized based on the observation that data restructuring workloads consist of known-shape, pre-located multidimensional arrays. Such arrays can be indexed using a set of loops. As shown in Figure 3.10, the DRX ISA includes specialized loop, compute, off-chip memory access, and synchronization instructions for vector operations while preserving the option for scalar operations, enabling serial tasks like pointer dereferencing.

The DRX ISA significantly departs from traditional SIMD semantics, offering optimizations for memory, loops, and data packing. For memory optimization, DRX employs software-managed on-chip scratchpads instead of vector register files and the conventional cache hierarchy found in common SIMD ISAs. Memory instructions configure the Off-chip Data Access Engine to fetch data directly from DRAM to the on-chip scratchpads. For loop optimization, DRX utilizes hardware loops within an Instruction Repeater unit to reduce branch instruction overhead. Loop instructions configure the Instruction Repeater based on the dimensions of the kernel's multidimensional arrays. For data packing optimization, the DRX compiler partitions the kernel's multidimensional arrays across the REs, eliminating the need for pack/unpack instructions.

During the vector execution, loop instructions first configure the Off-chip Data Access

Engine and Strided Scratchpad Address Calculator with sets of (Base, Stride, Iteration) configurations that correspond to the input/output loop dimensions and data location. After the Off-chip Data Access Engine loads the data to scratchpad banks, compute instruction is issued with scratchpad addresses calculated by the Instruction Repeater by traversing the dimensions of multidimensional arrays based on the configurations in the Strided Scratchpad Address Calculator. This data access scheme significantly reduces memory and address calculation overhead and is applied to all operations on multidimensional arrays such as data transformation, memory access, and compute operations. Finally, synchronization instructions are issued at the start and the end of the instruction stream to ensure proper program order. For scalar execution, DRX turns off all but one REs and operates as a scalar in-order CPU.

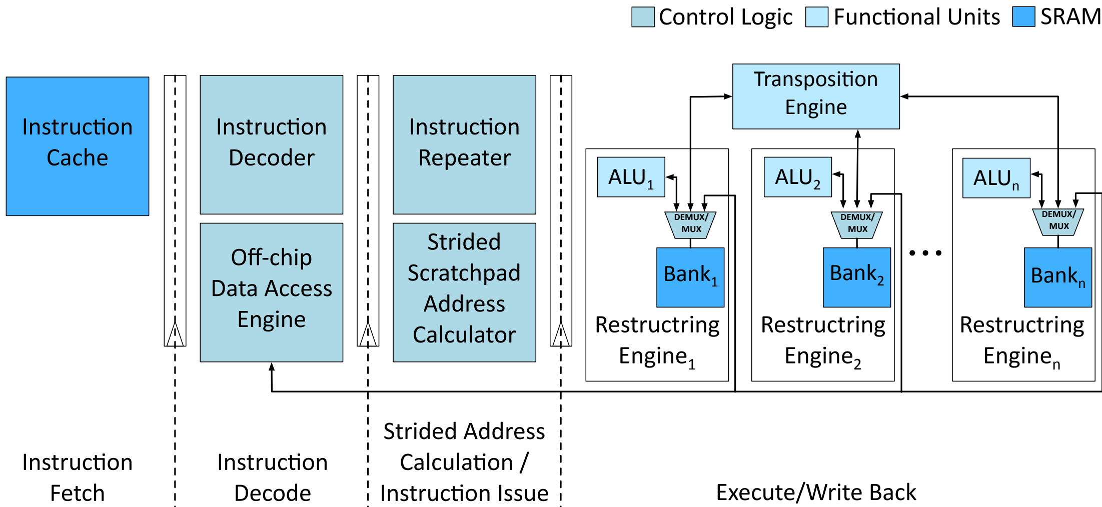


Figure 3.9: DRX Hardware Architecture.

	2 bits	4 bits	26 bits
Loop	Operaton	Function	Loop Dims, Base, Iter, Stride
Compute	Operaton	Function	Dest Addr, Src1 Addr, Src2 Addr
Off-chip Memory	Operaton	Function	Base/Tile Control, Req Size
Synchronization	Operaton	Function	Instruction Group, Start/Done

Figure 3.10: DRX instruction types.

DRX compiler. Inspired from prior works [CMJ$^+$18, CZY$^+$18] in other domains, DRX compiler

compiles high-level data restructuring kernels into DRX instructions based on the DRX ISA. The DRX compiler takes two inputs: a high-level representation of the data restructuring kernel and an architecture configuration file that defines the DRX hardware configurations such as the number of REs and on-chip scratchpad size. The compiler first maps the data restructuring kernel to the intermediate representation of the kernel operations. It then optimizes tiling and relaxes dependency on the intermediate representation based on the hardware configuration and the dimension of multidimensional arrays. Finally, it generates instructions based on DRX ISA from the optimized intermediate representation. Figure 3.11 shows a sample of the DRX kernel.

3.5 System Integration and Programmability

In this section we discuss the system integration and programmability of Fianchetto with Bump-in-the-Wire DRX placement. The system integration of other DRX placements share many similarities with Bump-in-the-Wire DRX.

Programming model. Fianchetto implements an OpenCL-style programming model that has a host program on the CPU and kernels on accelerators or DRX. Application kernels are executed on accelerators while data restructuring kernels are executed on DRX. Because Fianchetto runs the control plane on the CPU, it does not compromise the programmer's productivity and does not incur any additional accelerator orchestration overhead compared to the baseline multi-acceleration system.

The host program creates an execution context for each instance of the application kernel or data restructuring kernel. The context includes (1) the hardware – e.g. the accelerator or DRX– involved in the applications, (2) application or data restructuring kernels, and (3) a per accelerator *command queue* that is mapped to the global host address space. The command queue is used for buffering the output of the application kernels and the restructured input of the next application kernel before being transferred to the destination.

The host program uses user-level OpenCL API to create the execution context. It also uses the API to interact with the accelerators and DRXs through their own *command queue* on each device. The command queue accepts commands to enqueue kernels for execution, transfer data, or synchronize memory buffers. The execution of a command can be blocking or non-blocking. Blocking execution does not return to the host program before the current command completes. Non-blocking execution, on the other hand, requires a detailed description of the dependency between kernels and data restructuring programs. For a single command queue, the queued commands are executed in the order they are enqueued.

The application kernels execute domain-specific kernels of the end-to-end application on different accelerators. The data restructuring kernels perform the required data restructuring operations when two accelerators are communicating. The host program executes the serial portion of the application and runs a daemon to orchestrate the execution of application and data restructuring kernels running on accelerators and DRXs, respectively. The data restructuring kernels are shipped to DRXs that understand the exact input and output format of each accelerator. The data restructuring kernels are engaged to ensure that properly structured input/output data is moved directly between accelerators and DRX.

Driver support for Fianchetto. At a high level, Fianchetto enumerates both accelerators and DRXs as PCIe devices connected to the CPU. Each DRX unit has a driver to initialize the command queues, exchange the start and end pointers of the queue to other DRXs at the start, and orchestrate data restructuring operations. The drivers use GEM [lina, linb] for command executions and memory-related operations. DRX driver executes commands and reads/writes/maps operations using ioctl syscall. For setting up point-to-point DMA between DRX and accelerators, the drivers use dma-buf API [dma]. The vendor-specific accelerator drivers should support point-to-point DMA in order to work with Fianchetto. By default, we operate accelerators and DRXs in interrupt mode for sending notifications to the CPU. The interrupt handling of the drivers utilizes interrupt coalescing for the bursty arrival of interrupts. If the arrival rate of interrupts exceeds a

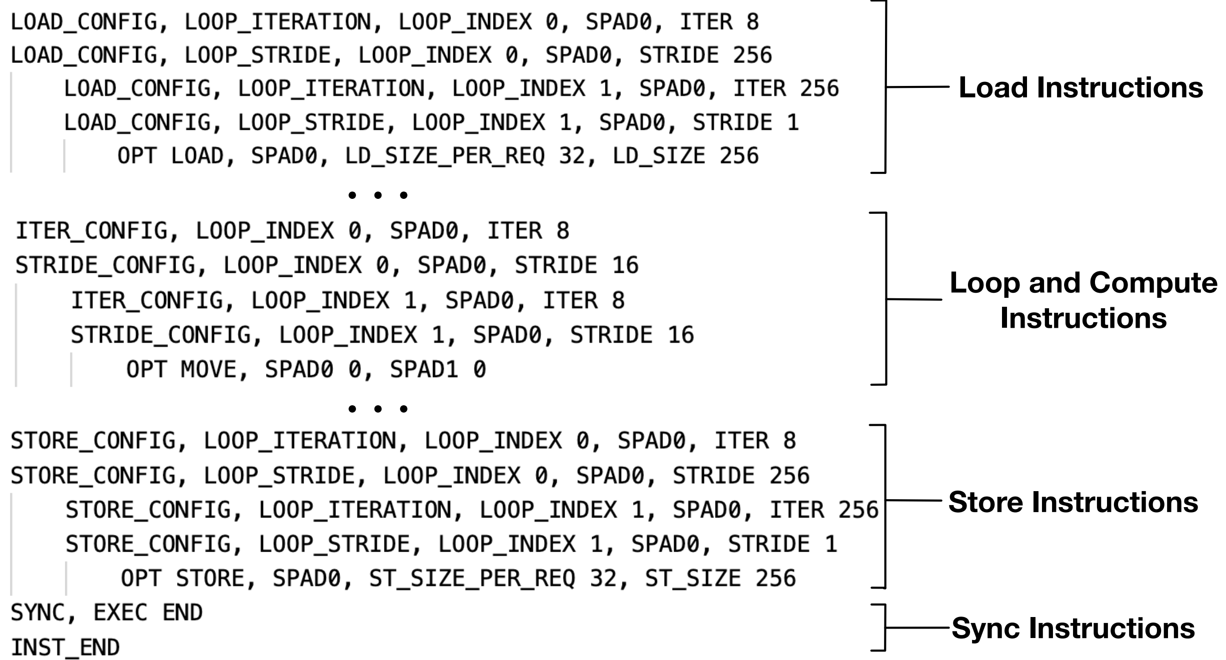


Figure 3.11: Sample DRX kernel.

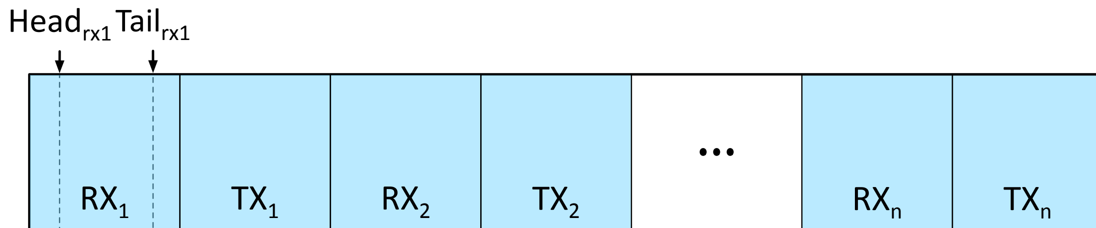


Figure 3.12: RX/TX data queue pair architecture in Bump-in-the-Wire DRX. DRX uses the data queue as a circular buffer with head and tail pointers. The output of the accelerator that is destined for $Accelerator_i$ is enqueued in RX_i before being restructured and stored in TX_i for transmission to $Accelerator_i$. Current DRX implementation supports up to a total $n = 40$ accelerators.

certain threshold, the drivers switch to polling. This design is similar to Linux NAPI design [nap].

Although Bump-in-the-Wire DRX is attached to each accelerator, each DRX unit should be able to set up a point-to-point connection with all the other accelerators and DRXs in the system. The memory address space of each DRX is statically partitioned between all the accelerators as well as DRXs in the system to implement two pairs of RX/TX *data queues* per accelerator on each DRX: one pair of queues for direct DRX-accelerator communication and another pair of

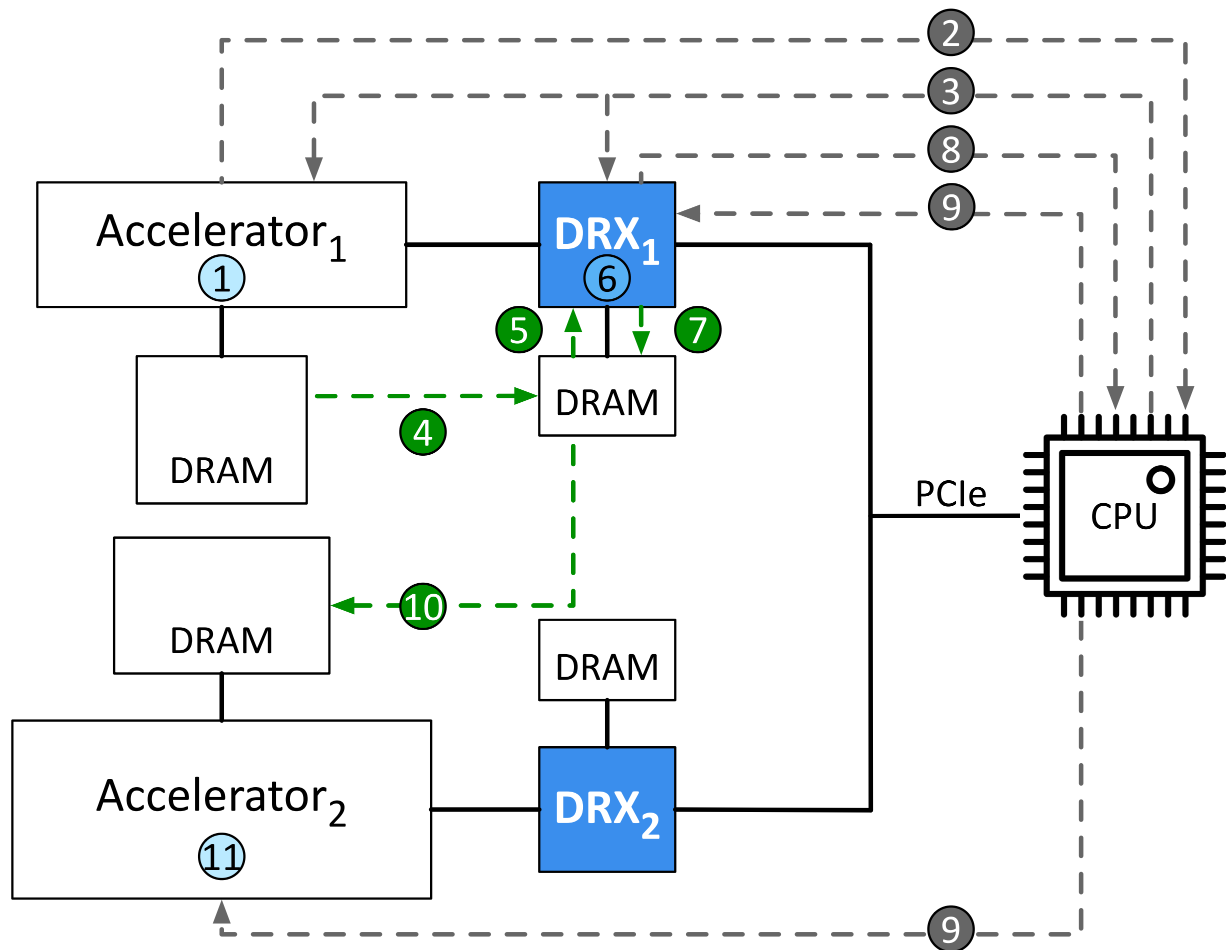


Figure 3.13: Point-to-point DMA workflow involves two accelerators and the sending side DRX. The DMA bypasses the receiving side DRX. Fianchetto supports other communication patterns such as broadcast and multicast among DRXs and between DRXs and accelerators.

queues for DRX-DRX communication.

The number of accelerators is determined at PCIe enumeration time when it discovers connected accelerators that need data restructuring. We provision 8GB of memory space for implementing data queues on each DRX. The size of each data queue pair is 100MB. This will enable Fianchetto to support up to 40 accelerators on a server. DRX driver maintains a head and tail pointer for each data queue to keep track of the data that is enqueued for restructuring. RX and TX data queues on a DRX are shown in Figure 3.12. A point-to-point DMA moves data between data queue pairs and accelerator memory.

GEM allocates and frees data buffers opaquely because it is agnostic to the data content in the buffer. The allocated data buffers are referred to by their handle, which is equivalent to a file descriptor.

Table 3.1: End-to-end benchmarks.

Benchmark	Kernel 1	Kernel 1 Accelerator	Data Restructuring	Kernel 2	Kernel 2 Accelerator	Input Dimension
Video Surveillance [APR20]	H.264 Codec	Xilinx Video Codec Unit [xila]	Mul, MaxPool, Reshape, Cast	Object Detection	DNN Accelerator [SPM+16]	(960, 540, 3)
Sound Detection [SJB14]	FFT	Xilinx Vitis DSP Library [xile]	Pow, Add, Mul, Div, Log10, Cast	Support Vector Machine	Xilinx Vitis Data Analytics Library [xilb]	(8192, 768)
Brain Stimulation [KdCL+20]	FFT	Xilinx Vitis DSP Library [xile]	Pow, Div, Mul, Cast	Proximal Policy Optimization	DNN Accelerator [SPM+16]	(256, 1024, 8)
Personal Information Redaction [micc]	AES-GCM	Xilinx Vitis Security Library [xilg]	Concat, Flatten	Regular Expression	Xilinx Vitis Data Analytics Library [xilb]	(4, 2048, 768)
Database Hash Join [SOI+17]	Gzip	Xilinx Vitis Data [xilc] Compression Library	Concat, Reshape, Cast	Hash Join	Xilinx Vitis Database Library [xild]	(4, 1024, 512)

End-to-end data motion acceleration. Figure 3.13 shows the interactions between accelerators, CPU, and Bump-in-the-Wire DRX when $Accelerator_1$ tries to communicate with $Accelerator_2$. Although Figure 3.13 depicts the accelerator and its DRX as separate chips with separate DRAM modules, DRX can be integrated into the accelerator chip and share its physical DRAM modules. When $Accelerator_1$ completes kernel execution in step ①, it raises an interrupt to the CPU in step ②. The driver of $Accelerator_1$ captures the interrupt and setup a point-to-point DMA between $Accelerator_1$ and the TX data queue corresponding to $Accelerator_2$ on DRX_1. DRX_1's driver shares the offset of RX_2 data queue (i.e., RX data queue corresponding to $Accelerator_2$) in step ③ with $Accelerator_1$. This enables the $Accelerator_1$ to access and write to the RX_2 data queue on DRX_1. A DRX driver then configures $Accelerator_1$ to perform a point-to-point DMA and move data from $Accelerator_1$'s memory to the next available buffer in RX_2 data queue on DRX_1 in step ④. The DRX processing unit on DRX_1 reads the output on $Accelerator_1$'s memory from RX_2 data queue, performs data restructuring, and writes the output to the next available buffer in TX_2 data queue as shown in step ⑤ to ⑦. In step ⑧, DRX_1 raises an interrupt to the CPU to notify the DRX_1 driver about the completion of data restructuring. Next, a point-to-point

DMA is configured between DRX_1 and $Accelerator_2$ in step ⑨. In step ⑩, point-to-point DMA between DRX_1 and $Accelerator_2$ passes through an internal PCIe multiplexer without invoking DRX_2 because it does not need further data restructuring on it. In step ⑪, $Accelerator_2$ runs the kernel on its DRAM.

One-to-many and many-to-one data movement. Supporting broadcast and multicast between the accelerator chain is necessary for load balancing as well as efficient collective communication implementation. The workflow of such movement patterns is similar to that of Figure 3.13, except that for one-to-many, the source DRX transfers the restructured output of the source accelerator to multiple accelerators (or DRXs) using multiple back-to-back point-to-point DMA transfers. Variations of many-to-one data movement can be used to implement reduction collectives by setting up direct data transfer from multiple source DRXs to a single destination DRX that also performs the reduction operation. The Fianchetto support for broadcast and multicast facilitates the efficient implementation of various collective operations.

3.6 Experimental Methodology

Benchmarks. We create five diverse cross-domain and end-to-end applications inspired by real-world scenarios. Table 3.1 lists the five benchmark applications, their cross-domain kernels and corresponding accelerators, the data restructuring operations needed to chain the kernels, and the dimensions of the input data. Each application is a pipeline of two kernels, where the first kernel outputs intermediate data, which requires restructuring before it can be processed by the second kernel. The Video Surveillance decodes input video streams into video frames and passes them to an object detection kernel [RF18]. Sound Detection performs Fast Fourier Transform (FFT) on audio snippets and use the transformed snippets to determine the genre of input audio [SJB14]. Brain Stimulation receives electromagnetic input signal generated from a brain simulation model, processes it with FFT and data restructuring operations before outputting the data to reinforcement

learning kernel [KdCL+20]. Personal Information Redaction decrypts privacy-sensitive text and uses a regular expression kernel to detect personally identifiable information and redact them from the text with blanks [micc]. Database Hash Join decompresses database tables and hash joins the tables [SOI+17, CMM+22]. To exercise the system performance with respect to resource contention on interconnect bandwidth and compute for data restructuring, we use 1, 5, 10, to 15 concurrent running applications for the benchmarks.

DRX hardware implementation. We implement DRX using Verilog in RTL and synthesize it on Xilinx UltraScale+ VU9P FPGA using Xilinx Vivado 2022.2. The synthesized design achieves an operating frequency of 250 MHz. We also synthesize an ASIC version of DRX using Synopsys Design Compiler R-2020.09-SP4 with the FreePDK 15nm standard cell library [SCW+07]. The ASIC implementation achieves a 1 GHz operating frequency.

Baseline FPGA-based multi-acceleration system. Beside DRX, we also synthesize application kernels discussed earlier in this section on FPGA to implement a baseline multi-acceleration system without data motion acceleration (i.e., that uses CPU for performing data motion). This setup consists of multiple AWS Xilinx UltraScale+ VU9P FPGAs [ama] connected through PCIe x16 to Intel Xeon Platinum 8260L CPUs operating at 2.4 GHz with 64 GB of memory and hyperthreading disabled.

We implement the application kernels on the FPGA using the following methods: hard-IP blocks, High-Level Synthesis (HLS), or Register-Transfer Level (RTL) implementation. For the video codec kernel, we use a pre-existing hard-IP available on the VT1 instance of AWS [awsc]. We use Xilinx's Vitis libraries [xilf], which provide HLS implementations, for kernels such as FFT, support vector machine, AES-GCM, Gzip decompression, regular expression, and database hash join. We use the RTL implementation from open-sourced accelerators [SPM+16] for the remaining kernels that use deep neural networks such as object detection and proximal policy optimization. We synthesize both the HLS and RTL implementations on the FPGAs operating at 250 MHz clock frequency.

In this FPGA multi-acceleration implementation the host CPU runs the control plane (refer to Sec.3.5) and performs the data restructuring operations while the FPGAs accelerate the application kernels.

Performance evaluation. We use the FPGA setup to collect cycle-level latency of executing end-to-end applications on a baseline without data motion acceleration (we refer to this baseline as *Multi-Axl* configuration in Sec.3.7). We then scale the performance of FPGA acceleration using scaling factors based on ASIC implementation and clock frequency (250 MHz to 1GHz). We develop an end-to-end system emulation infrastructure to compare the performance of different configurations of Fianchetto with a multi-acceleration baseline without Fianchetto. The input to the emulation setup are cycle-level latency numbers for executing application kernels, data restructuring on the CPU or DRX, communication over PCIe, and software stack overheads for interrupt and polling.

Energy evaluation. We measure the energy of the CPU using Intel RAPL [intd]. We use the post-synthesis power of the FPGA and multiply it by the execution time of the kernels to estimate the energy consumption for the accelerators. We also include the energy consumption of the PCIe switch [bro] and the energy for data transfer over PCIe [BWPN18].

3.7 Experimental Results

3.7.1 End-to-end Performance Improvement

Speedup. Figure 3.14 compares the end-to-end execution time of cross-domain applications withiout (*Multi-Axl*) and with Fianchetto. Note that Fianchetto uses Bump-in-the-Wire DRX placement. On average, accelerating the data motion provides 3.5× to 8.2× speedup for running one to 15 concurrent applications. The higher the number of accelerators in use, the greater the data motion between the accelerators. Therefore, as DRX accelerates the data restructuring portion of the end-to-end application, the speedup grows as the number of concurrent applications

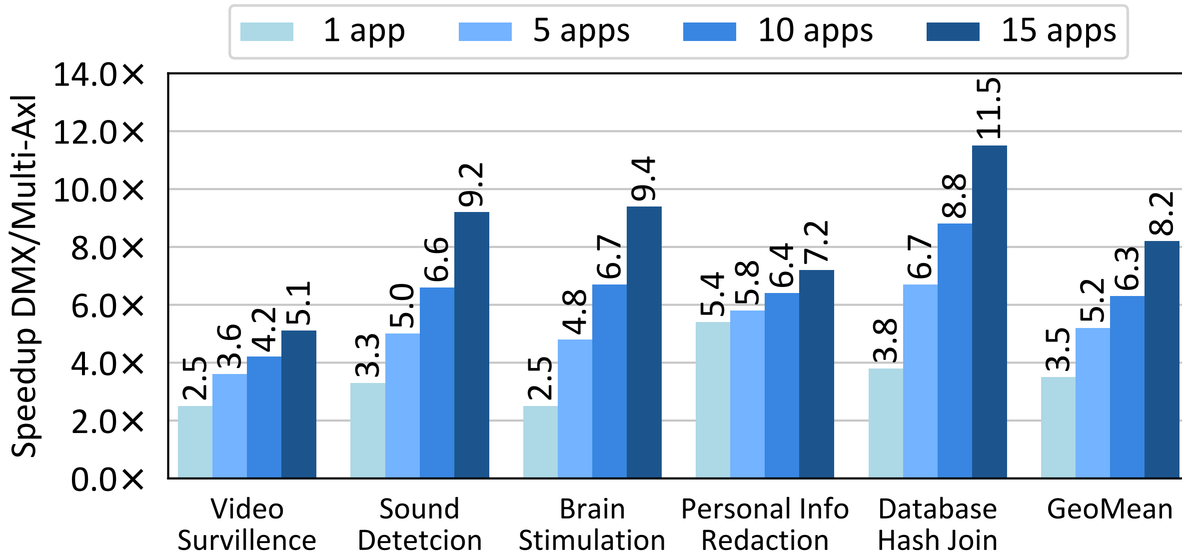
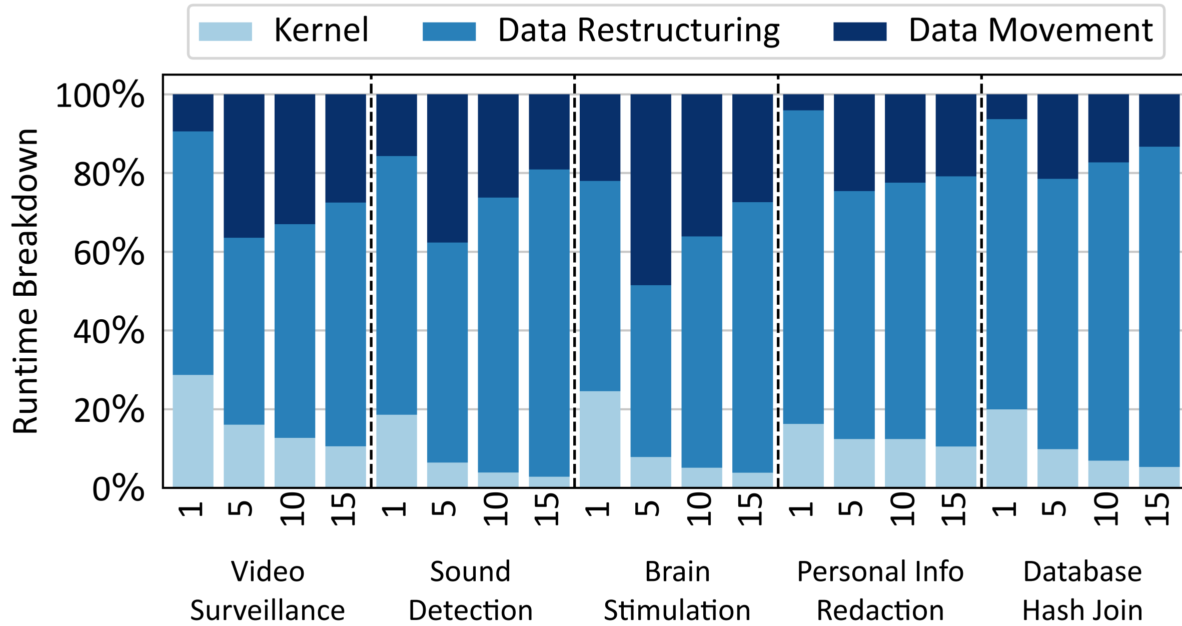


Figure 3.14: Fianchetto speedup over *Multi-Axl* configuration that uses CPU for data motion between accelerators. Fianchetto performance scales with the number of concurrent applications by using Bump-in-the-Wire DRX placement.

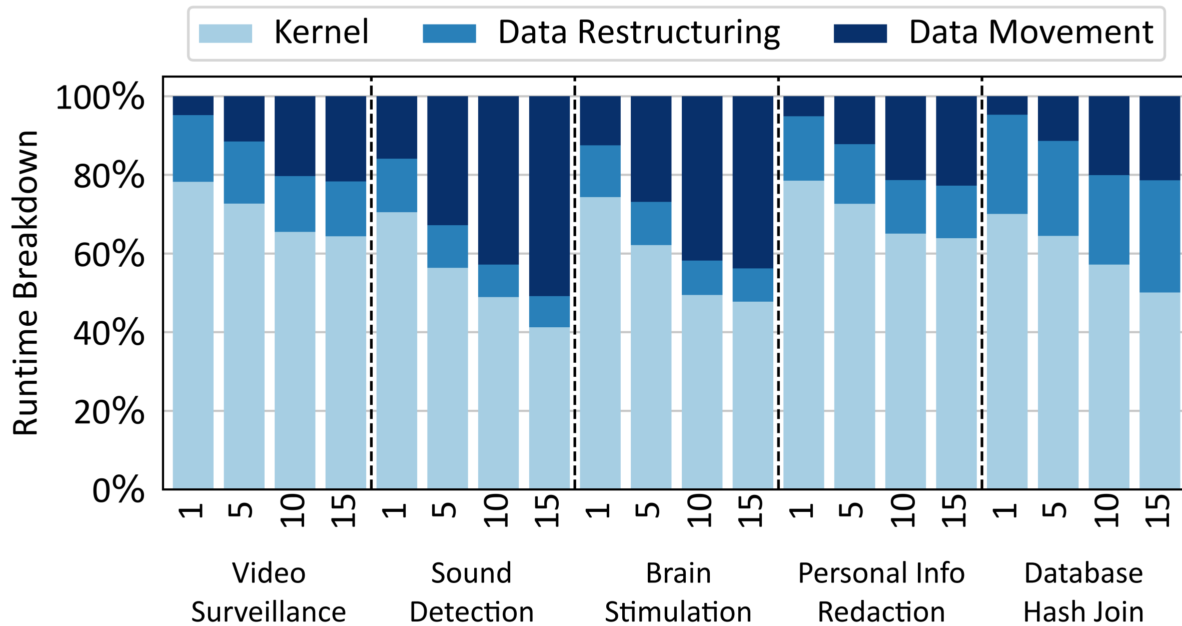
increases. Fianchetto yields less end-to-end speedup for Video Surveillance because the accelerator used for Video Surveillance provides less speedup compared to the other benchmarks. The speedup of Fianchetto is more pronounced for Database Hash Join because the data restructuring takes up the majority of the runtime for this benchmark which is significantly being accelerated by DRX.

To better understand the sources of benefits, Figure 3.15(a) and Figure 3.15(b) report the runtime breakdown for *Multi-Axl* baseline and Fianchetto across the three main runtime components: accelerated kernels time, data restructuring, and data movement time between CPU and accelerator for *Multi-Axl* and between accelerators for Fianchetto. Kernel execution latencies are the same for both *Multi-Axl* and Fianchetto. However, after we apply Fianchetto (Figure 3.15(b)), the kernel execution takes up larger portion of the runtime breakdown compared to the baseline (Figure 3.15(a)).

As shown in Figure 3.15(a), data restructuring accounts for the largest portion of the end-to-end runtime for the baseline. Data restructuring is on average 66.8%, 55.7%, 64.7%, and 71.7% of multi-acceleration end-to-end latency for 1, 5, 10, and 15 concurrent applications,



(a) The runtime breakdown of *Multi-Axl*.



(b) The runtime breakdown of Fianchetto.

Figure 3.15: The latency breakdown of the *Multi-Axl* baseline and Fianchetto. Fianchetto shrinks data restructuring ratio from 64.1% to 14.1% in average.

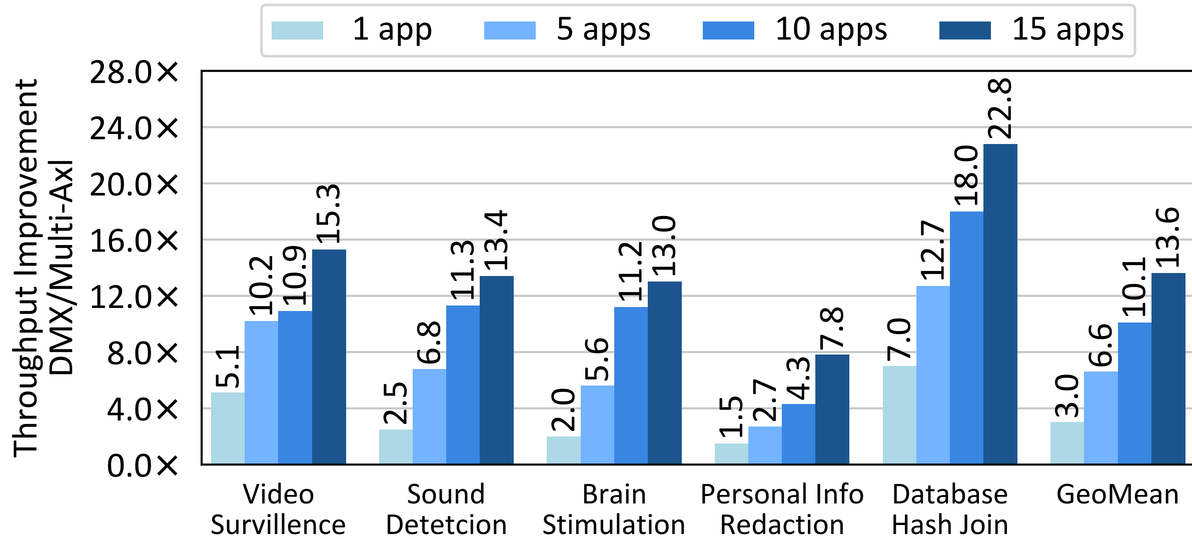


Figure 3.16: Fianchetto throughput improvement over *Multi-Axl*. Fianchetto resolves the throughput bottleneck of data restructuring and shifts the throughput bottleneck to the accelerated kernel.

respectively. Using DRX significantly accelerates data restructuring and shrinks data restructuring overhead to 17.0%, 15.3%, 13.5%, and 7.2% of Fianchetto end-to-end latency for 1, 5, 10, and 15 concurrent applications, respectively, as shown in Figure 3.15(b). Increasing the number of concurrent applications requires more accelerators, meaning more computation for data restructuring operations between accelerators. Furthermore, the data movement in the baseline system increases due to the bandwidth bottleneck caused by multiple accelerators sharing the PCIe switch's upstream bandwidth. On the contrary, Fianchetto accompanies each accelerator with its own local DRX and therefore avoids bandwidth contention on shared PCIe links.

Throughput improvement. Although the end-to-end execution latency of each request is important, in a real world setup, an application receives back to back requests that need to be processed in the cross-domain application pipeline. Therefore, assuming that each application consists of three pipeline stages (first kernel, data motion, and second kernel as shown in Figure 3.2), the throughput of an application is determined by the latency of the slowest stage. We compare the throughput of *Multi-Axl* baseline and Fianchetto assuming continuous arrival of

requests for each application.

Figure 3.16 shows the throughput improvement of Fianchetto over the multi-acceleration baseline. On average, Fianchetto achieves from 3.0× to 13.6× throughput improvements when running one to 15 concurrent applications, respectively. Data restructuring is the slowest stage of the application pipeline in the *Multi-Axl* baseline as demonstrated in Figure3.15(a). Hence it is the throughput bottleneck for all benchmarks, especially as the number of concurrent applications increases. Fianchetto leverages DRX to address this bottleneck and shifts the throughput bottleneck to the accelerated kernel. Personal Info Redaction shows relatively low improvement on the throughput as its throughput is limited by its regular expression kernel accelerator. Data movement is not the throughput bottleneck for the *Multi-Axl* baseline because the PCIe bandwidth never gets saturated due to the poor throughput of data restructuring operations on the CPU.

3.7.2 DRX Placement Analysis

One of the critical design decisions in Fianchetto is the location of the DRX in the system: Integrated, Standalone, Bump-in-the-Wire, PCIe-Integrated. This is because the placement of DRX impacts the data movement and the overall system design.

Speedup with different DRX placements. Figure 3.17 compares the latency speedup between Integrated DRX,Standalone DRX, Bump-in-the-Wire DRX, and PCIe-Integrated DRX. The figure reports the average speedup across the five benchmarks for one to 15 concurrent applications. For all setups from one through 15 concurrent applications, the results show that the speedups compared to the *Multi-Axl* baseline are in the following order: Integrated $\leq$ Standalone $\leq$ Bump-in-the-Wire $\leq$ PCIe-Integrated.

Integrated DRX shows 4.4× speedup with 15 concurrent applications compared to the baseline where data restructuring is performed on the CPU. However, when running more than one application in Integrated DRX, the concurrent applications contend for the shared DRX

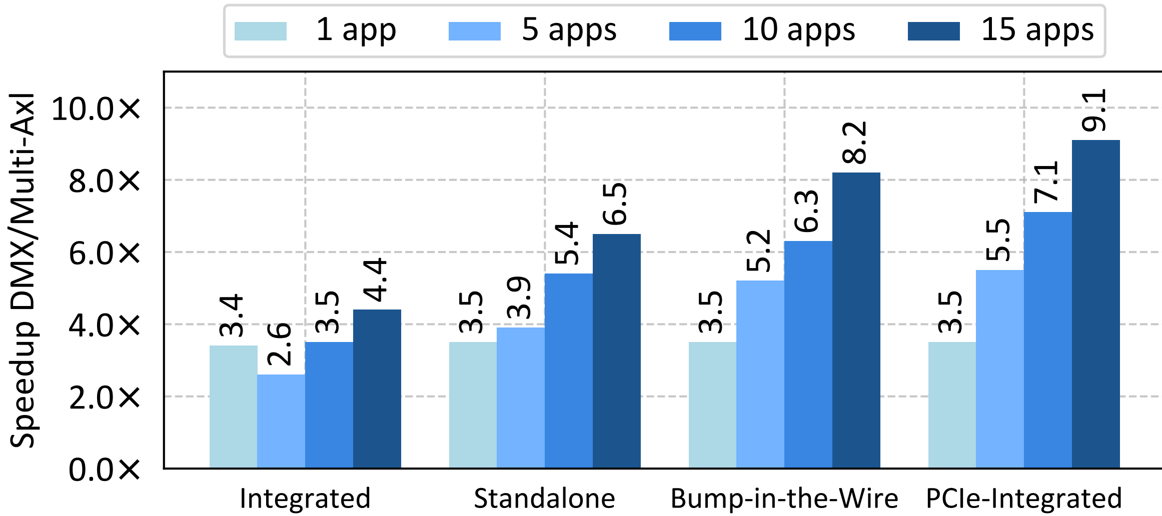


Figure 3.17: Comparison of end-to-end latency speedup with different DRX placements. Integrated DRX integrates a shared DRX on the CPU. Standalone DRX implements DRX as a standalone PCIe card shared by accelerators. Bump-in-the-Wire DRX is an exclusive DRX to each accelerator. PCIe-Integrated DRX integrates shared DRXs with PCIe switches connecting accelerators.

computation resources on the CPU and the PCIe bandwidth to access the shared DRX. The upstream port of the PCIe switch connecting to the CPU uses a single link (8 lanes) while the downstream ports connecting to accelerators use multiple links. Also, a PCIe transaction pays 110 ns or more port-to-port latency tax to get through a PCIe switch [bro]. Despite the significant overhead from the contended PCIe links, Integrated DRX's speedup relative to the baseline increases as we add more accelerators. This demonstrates the benefits of using DRX instead of general-purpose CPU cores for data restructuring operations.

Standalone DRX shows 3% and 48% improvements compared to the Integrated for one and 15 concurrent applications, respectively. In the Integrated DRX, we have a single DRX that is integrated to the CPU for the entire system. On the other hand, the Standalone configuration scales the number of DRX with the number of concurrent applications by inserting more DRX PCIe cards. Therefore, the speedup compared to Integrated DRX can be attributed to the larger number of DRX in the system.

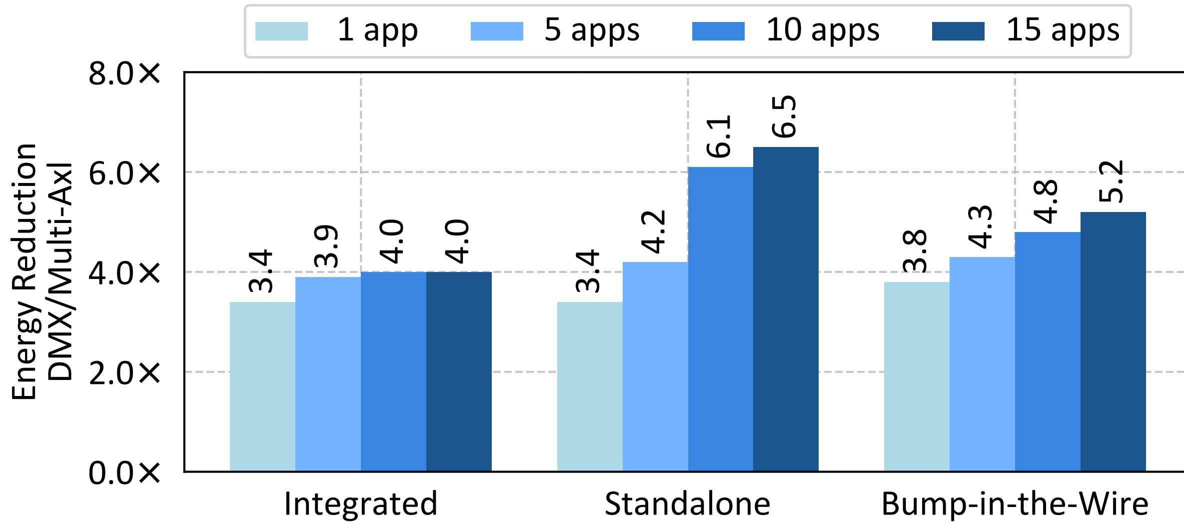


Figure 3.18: System-wide energy reduction, including host CPU cores, accelerators, and DRXs. Bump-in-the-Wire DRX achieves less reduction than Standalone DRX due to its internal PCIe multiplexer shown in Fig. 3.7. Integrated, Standalone, and Bump-in-the-Wire DRX draw up to 26%, 23% and 28% more power than the *Multi-Axl* baseline. PCIe-Integrated is not included because we are not able to estimate the power of a DRX-integrated PCIe switch.

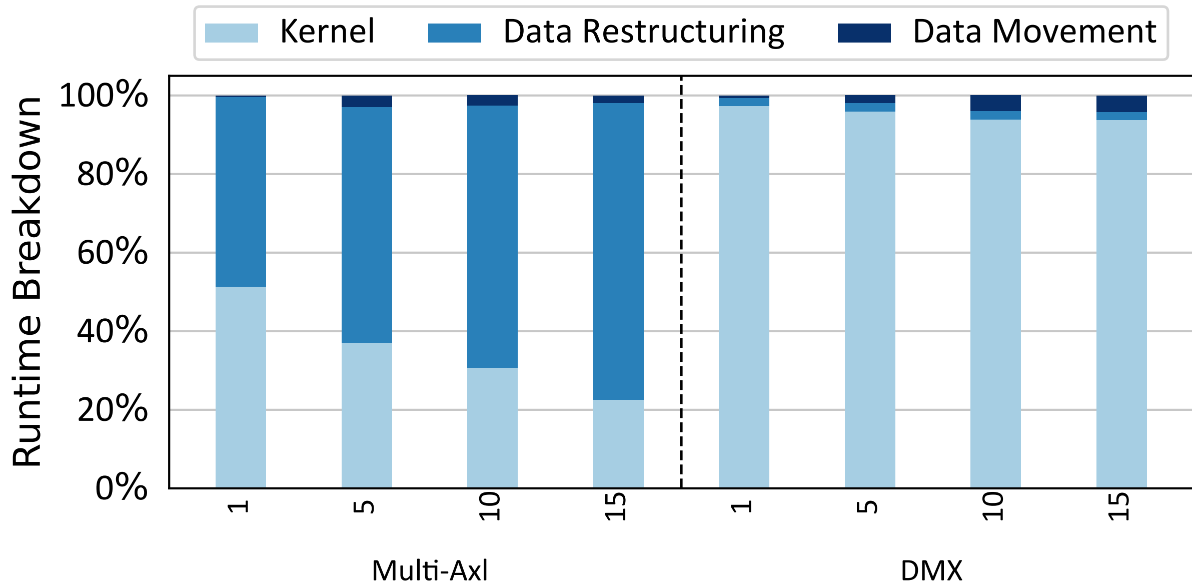
Bump-in-the-Wire DRX achieves 33%, 17%, and 26% higher speedup for 5, 10, and 15 concurrent applications compared with Standalone DRX. Bump-in-the-Wire DRX keeps its point-to-point DMA traffic between accelerators and DRX under the same PCIe multiplexer so the accelerators do not need to contend for PCIe bandwidth as in Standalone DRX placement on the CPU.

PCIe-Integrated DRX shows the highest speedup. The improvement of PCIe-Integrated DRX against Bump-in-the-Wire DRX comes from the saving of a round-trip between the source DRX and the source PCIe multiplexer and a pass-through of the destination PCIe multiplexer. However, it is important to note that the integration of DRX with a PCIe switch requires in-depth modification to make the PCIe switch programmable and process data at the line rate. In other words, despite the luring benefits, the prohibitive level of engineering effort to achieve it makes the Bump-in-the-Wire a reasonable choice of Fianchetto design that can achieve significant speedup with relatively affordable engineering efforts.

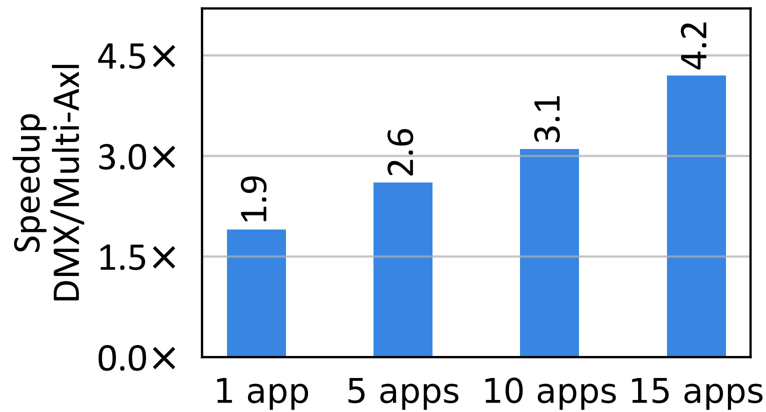
Energy reduction with different DRX placements. Figure 3.18 shows system-wide energy reduction provided by different DRX placements compared to *Multi-Axl* baseline. Integrated DRX provides 3.4×, 3.9×, 4.0×, and 4.0× of energy reduction. The energy reduction does not scale with the number of concurrent applications because it only benefits from the energy efficiency of the DRX hardware acceleration for data restructuring operations. Standalone DRX and Bump-in-the-wire DRX provide energy reduction scaling with the increased number of concurrent applications. Bump-in-the-wire DRX placement delivers the best energy reduction of 3.8× and 4.3× for 1 and 5 concurrent applications. Standalone DRX delivers the best energy reduction of 6.1× and 6.5× for 10 and 15 concurrent applications because of the reduced bandwidth contention on PCIe links. This is because the extra glue logic and the dual-port PCIe multiplexer are replicated in each Bump-in-the-Wire DRX placement, while such overhead is amortized across the applications on a large Standalone DRX. PCIe-Integrated is not evaluated for energy reduction because of the difficulty of estimating the energy consumption of a PCIe switch integrated with DRX.

3.7.3 Sensitivity Studies

Speedup with more than two kernels. As real-world applications can consist of multiple kernels across domains, it is important for Fianchetto to scale beyond two kernels. To evaluate Fianchetto's scalability with multiple application kernels, we add a third application kernel to the Personal Info Redaction benchmark, along with its additional data restructuring kernel consisting of reshaping and typecasting. This third kernel is a Transformer model fine-tuned for Named Entity Recognition (NER). NER identifies personal and sensitive information that is hard to capture for regular expression kernel [ner]. We use an open-source BERT implementation for the kernel [EGG+21]. Figure 3.19(a) shows the runtime breakdown of this three-kernel benchmark. Although the benchmark included the compute-intensive NER kernel, the runtime is still dominated by the data restructuring kernels for the *Multi-Axl* baseline. Fianchetto alleviates



(a) Runtime breakdown



(b) Speedup

Figure 3.19: Fianchetto reduces data motion overhead to less than 5% for Personal Info Redaction benchmark extended with Named Entity Recognition kernel.

the bottleneck of data motion and restores kernel to be the largest contributor that represents 97.2% to 93.7% of the end-to-end execution time for one to 15 concurrent applications. As such, Fianchetto provides 1.9× to 4.2× speedup for one to 15 concurrent applications shown in Figure 3.19(b).

One-to-many and many-to-one data movement. Cross-domain multi-acceleration of end-to-end applications entails using multiple accelerators. The data movements in multi-acceleration,

however, are not necessarily always one-to-one but likely include one-to-many and/or many-to-one data movement between accelerators. Therefore, we want to analyze whether Fianchetto design can cope with the one-to-many and/or many-to-one data movements in multi-acceleration. To this end, we compare Bump-in-the-Wire DRX against the *Multi-Axl* baseline for one-to-many (i.e., broadcast) and many-to-one (all-reduce) data movement using 4 to 32 accelerators. For broadcast, the baseline first passes the output of the source accelerators to the main memory of the CPU using DMA. After data restructuring on the CPU, the driver then copies the restructured data and initiates *N* DMA transfers *sequentially* to the destination accelerators. All-reduce has two stages: scatter-reduce and all-gather. Both require similar DMA transfers between CPU and accelerators; however, scatter-reduce entails additional steps to first sum the inputs from sources and then scatter the outputs to all destinations. On the contrary, Fianchetto's implementation of broadcast and all-reduce utilizes the Bump-in-the-Wire DRX for data restructuring and data movement.

Figure 3.20 shows that the Fianchetto achieves 3.7× to 5.2× speedup on broadcast and 5.1× to 10.5× speedup for all-reduce on 4 to 32 accelerators. This is because Fianchetto utilizes DRXs to (1) perform data restructuring and the DMA transfers *in parallel* and (2) eliminate the extra DMA transfers between the accelerators and the CPU. Furthermore, for all-reduce, Fianchetto uses DRX to accelerate the summation operations. The speedup also scales with the number of accelerators because the amount of data restructuring and data movement scale accordingly to the number of accelerators. There is a dip when using 16 or more accelerators, but this is due to the additional latency on the PCIe switches that scales with the number of accelerators. Fianchetto achieved higher speedup in all-reduce compared to broadcast because all-reduce involves more DMA transfers and data restructuring which provided more acceleration opportunity using DRX.

DRX hardware configurations. To understand the sensitivity of Fianchetto to the amount of compute resources in DRX, we sweep the number of RE lanes for DRX and compare its

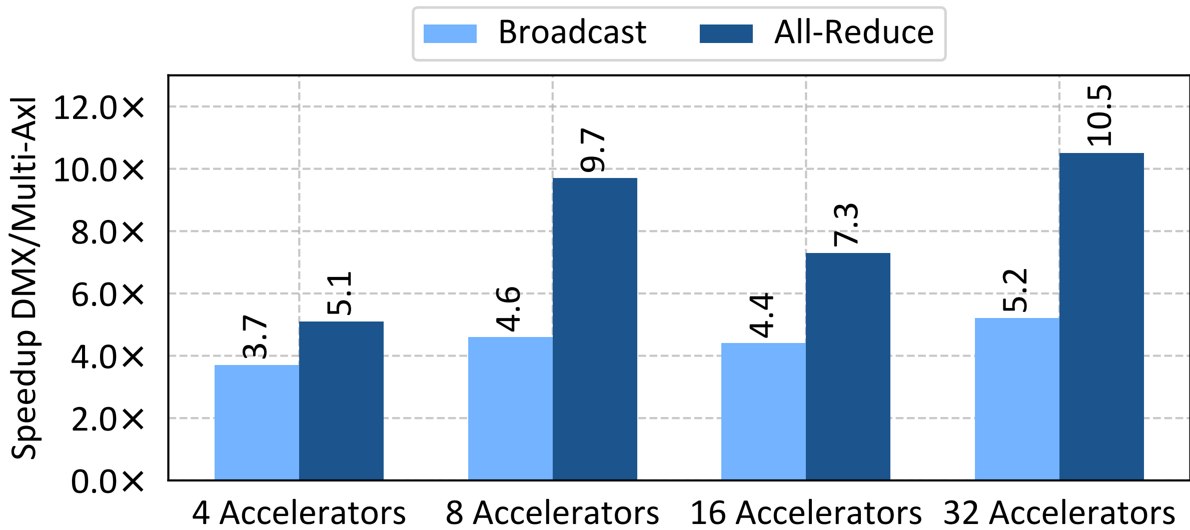


Figure 3.20: Fianchetto eliminates redundant DMA transfers and performs DMA in parallel for broadcast and all-reduce on multi-accelerator setup.

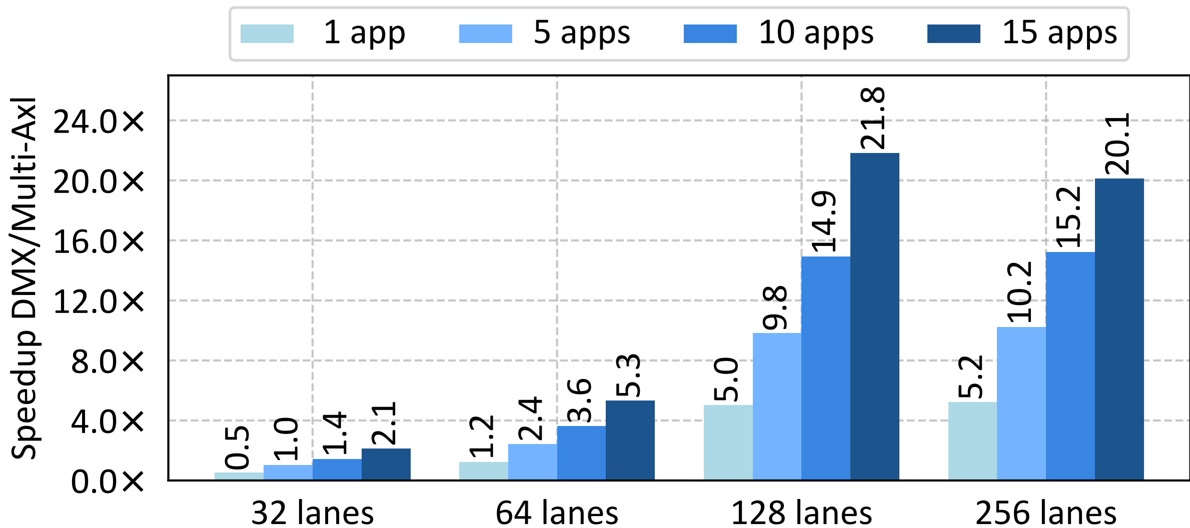


Figure 3.21: Data restructuring latency speedup with different numbers of RE lanes on DRX. The increase of speedup is limited after 128 lanes. which is our default configuration.

performance to the *Multi-Axl* baseline that performs data restructuring on CPU. Figure 3.21 shows the speedup achieved for the different number of lanes for DRX: from 32 to 256. The speedup improves with the number of lanes increasing up to 128 lanes by taking advantage of available data parallelism in data restructuring operations. However, the increase of speedup of DRX is

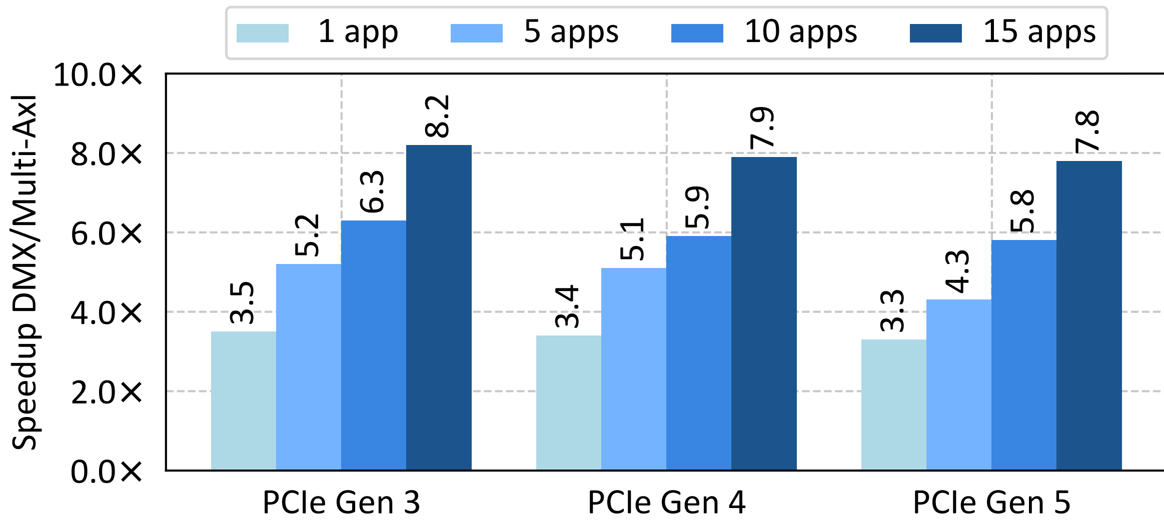


Figure 3.22: Fianchetto speedup across generations of PCIe. PCIe Gen4 and Gen5 result in a slight decrease of speedup because their corresponding *Multi-Axl* baselines improve more than their Fianchetto counterparts.

limited after 128 RE lanes, and increasing the lanes to 256 does not provide noticeable benefits. Therefore, we use 128 RE lanes as the default configuration for DRX throughout the experiments.

Different PCIe generations. Newer PCIe generation provides significantly more bandwidth and the increased bandwidth can potentially negate the performance benefit of Fianchetto. To understand the impact of different generations of PCIe, we compare the Bump-in-the-Wire DRX latency speedup on PCIe Gen 3 with PCIe Gen 4 and Gen 5. Figure 3.22 shows that using PCIe Gen 4 and Gen 5 resulted in a slight decrease of speedup because their corresponding *Multi-Axl* baselines improve more than their Fianchetto counterparts. Across PCIe generations, the baselines and Fianchetto show different levels of improvement only in data movement latency. Such differences come from the following two reasons. First, the baselines face more bandwidth contention than the Fianchetto and thus benefit more from the increased PCIe bandwidth per lane. Second, the baselines are able to use more PCIe lanes to reduce bandwidth contention from accelerators to CPUs with PCIe Gen 4 and Gen 5 compared to CPUs with PCIe Gen 3 [inta, intc, inte]. The results shown in Figure 3.22 suggests that the bottleneck of *Multi-Axl*

configuration is not just the PCIe interconnect, but also the data restructuring computation.

3.8 Related Work

Real-world applications span multiple domains, posing a challenge for end-to-end acceleration. While the research community has explored accelerators across diverse domains [WLP+14, KGP+13, SOI+17, CMM+22, APR20, HMSA+21, MGPM+22, CKL+22, KGP+13, MFJQ+16, SMLE18, LZT+21, NPB+21], the adoption of these heterogeneous accelerator to accelerate a single end-to-end application is challenging. The challenge arises due to the diverse data formats generated and consumed by each accelerator. This necessitates restructuring inputs and outputs across accelerators. While some prior works have focused on performed data restructuring using CPUs, Fianchetto introduces the concept of data motion acceleration of for efficient cross-domain multi-acceleration with heterogeneous DSAs. We review the most relevant related work in three areas: data movement, data restructuring, and interconnect fabrics integration below.

Data movement. Prior works studied point-to-point data movement between GPUs [gpu], between GPU and storage [TZZ+16, BBCS17, LT21], between NIC and accelerator [NKA20, TMS20, EFM+22], and between on-chip accelerators [CGG+12]. Prior works have used various techniques such as scheduling [MYS+22, MAW+22, DK13, TMS20, EFM+22] to co-locate multiple domains on the same system. While these works only optimize the data movement, non-trivial operations of the data restructuring still consumes a significant fraction of the data motion. Intel Data Stream Accelerator [intb] and DCS [AKK+15, KAC+18] share a similar insight, both lack programmability and hence have limited capacity to optimize data restructuring. This work in contrast leverages DRXs as a compute-enabled glue that links different heterogeneous accelerators together and makes them appear as a monolithic but composable accelerator for the application.

Data restructuring. For message serialization, Optimus Prime [PGK+20] and Protobuf accel-

erator [KLK+21] design an accelerator for RPC message serialization. HGum [ZAC17] and Fletcher [PVSW+] implement serialization on FPGAs for acceleration. For machine learning pipelines, tf.data [MvKI21], DSI [ZAB+22], DALI [nvi] optimize data restructuring on GPU with programmable operations. In contrast to these prior works that only optimize data restructuring for a single accelerator, this chapter investigates data restructuring and movement for multi-acceleration with heterogeneous devices.

Interconnect fabrics. Previous works have used PCIe's Non-Transparent Bridge (NTB) to enable PCIe to support multiple hosts with more than one root complex, which performs address translation for operations in a specific memory range [HJZ+13, MKH+21]. Point-to-point DMA over PCIe fabric is enabled by a shared address space across all devices [gig]. CXL 3.0 or later allows accelerators on different servers to be connected seamlessly by using fabric switching to link racks of devices and accelerators [cxl]. DUA [SCC+19] creates an overlay fabric on top of the existing physical communication stacks, such as PCIe, Ethernet, DDR, etc. These works can connect accelerators without addressing data restructuring for multi-acceleration. This work, however, tackles data motion challenges to maximize the performance of multi-acceleration.

3.9 Conclusion

In this chapter, we quantified the data motion performance and cost of chaining heterogeneous domain-specific accelerators for multi-acceleration. The results showed that the data motion overhead curtails the end-to-end speedup of accelerating each domain on a set of heterogeneous accelerators. The chapter introduced Fianchetto that seamlessly weaves together multiple accelerators that deliver the performance of a large, monolithic cross-domain accelerator. On average, Fianchetto provides between 3.4× to 8.2× speedup, 3.0× to 13.6× higher throughput, and 3.8× to 5.2× energy reduction.

Even with current single-domain accelerators, overheads of moving data on- and off-chip

is presently a dominant factor that limits the performance and energy efficiency of gains [Hor14, Dal23]. The impact of the data motion–highlighted in this chapter–will worsen when cross-domain accelerators are chained in future datacenters to cater to the requirements of emerging end-to-end applications. This even includes the multimodal generative AI applications that use multiple models and require acceleration beyond neural networks (e.g., vector database lookups, search, etc.). Heterogeneous/3D integration coupled with emerging high-bandwidth chiplet-to-chiplet interconnects such as UCIe can improve data movement, but not data restructuring that requires computation. As such, embedding our Fianchetto concept and architecture within these interconnects can synergistically unlock the the potential of cross-domain multi-acceleration for next-generation dataceneters.

3.10 Sources for Material Presented in This Chapter

Chapter 3, in part, reprints material as it appears in a paper titled: ”Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators” by Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh [WXM+24]. The dissertation author was the primary researcher and author of this material.

Chapter 4

Aurelia: Scalable CXL fabric

With the trend toward disaggregation and composable infrastructure, different resources in datacenters are taken from a logical pool to satisfy the demand from applications. Existing efforts of disaggregation relies on RDMA over Ethernet, a fabric not designed with disaggregation in mind [SHCZ18, ABAL+20, MC20, RSAB20, ZWL+22, LMC+22, WQM+23]. However, retrofitting existing hardware prevents applications from achieving their optimal performance under disaggregation.

Recent works focus on co-designing hardware with software to realize disaggregation [CIP+21, Sha23, MWD+23, LBN+23]. These works focus on externalizing hardware interconnects to create a fabric that connects many devices in a disaggregated setting Fabrics directly connect all the devices allowing access to remote devices in a manner similar to accessing local devices on a server's PCIe slots. PCIe is an example of this type of fabric, but it currently only connects local devices within a server. An ideal fabric for disaggregation offers direct connections between devices while providing low latency and high bandwidth at a specific scale, e.g. a single rack or a few neighboring racks.

ompute Express Link (CXL) has emerged as a viable candidate, supported by a converged industry standard following its absorption of OpenCAPI and Gen-Z. CXL is built on top of PCIe

with the addition of memory accessing semantics (CXL.mem), caching semantics (CXL.cache), and peer-to-peer memory access between devices (Unordered I/O in CXL.io). More importantly, CXL can be a fabric through its support of multi-level switching.

Experimental CXL fabric demonstrates on-par performance with lower cost in the case of machine learning model training on tens of GPUs [fab]. CXL fabric offers bandwidth as high as 63 GB/s with PCIe 5.0 now and 121 GB/s PCIe 6.0 on the horizon within the next two years [pcib]. CXL fabric offers low latency by operating on a device-attached interface that uses direct load/store instructions. It avoids the network software stack overhead and the PCIe transition between device and NIC on the sending (Device $\rightarrow$ PCIe $\rightarrow$ NIC) and receiving (NIC $\rightarrow$ PCIe $\rightarrow$ Device) path. The network stack and PCIe transition create latency overhead and throughput bottleneck. First, the network stack using kernel bypassing still incurs latency overhead in the range of microseconds [KCH+19, OFB+19, KKA19, MdKA+19]. Second, the PCIe latency overhead of 1500 B packets reaching the wire can be as high as 77% [NAZ+18]. Third, the PCIe link to NIC is a potential throughput bottleneck when multiple devices share the NIC. Dedicating a NIC for each device circumvents the throughput bottleneck but at the cost of requiring more NICs and switches.

4.1 Motivation and Background

Compute Express Link (CXL) has emerged as an enhancement of PCIe, providing cache coherency (CXL.cache), host-managed or fabric-attached memory (CXL.mem), and peer-to-peer memory access between I/O devices (CXL.io). CXL.mem provides host-managed memory that CPU and accelerators are able to read/write into each other's memory directly. This avoids redundant DMA operations for moving data back and forth [LGS+20, HMCX22, Sha23]. CXL.mem enables fabric-attached memory providing a shared memory pool for applications with different demands [Jun22, GLKJ22, LBN+23]. CXL.cache supports a fully coherent cache on the devices.

These devices, however, do not open their local, private memory to CXL-capable hosts. CXL.io uses unordered I/O for peer-to-peer memory accesses between non-coherent devices over its fabric.

Fabric Routing. CXL routes packets with a per-device ID called Port ID on the fabric. The Port ID-based routing (PBR) addresses each device with a 12-bit ID. A packet using PBR contains a specific source port ID and destination port ID before it leaves an edge CXL switch that directly connects with devices. Each CXL fabric has a single fabric manager responsible for initializing, binding/unbinding devices to ports, and handling event notifications, such as device removal or failure, from the switch. This fabric manager functions similarly to a centralized network controller, as it controls per-port forwarding and is aware of all route changes.

Flow control. CXL inherits point-to-point flow control from PCIe, which was designed for communication between the device and CPU rather than for a fabric. The flow control operates between two directly connected endpoints. They exchange credit tokens to evaluate the available buffer space on each side.

QoS Telemetry:. CXL fabric offers a rate throttling mechanism for hosts called QoS Telemetry. It is used for devices with local memory, including memory expansion devices and accelerators with device memory, such as GPUs, FPGAs, and ASICs. QoS telemetry enables memory devices to indicate their current internal load with 2 bits in CXL.mem response packets. Senders use the reported internal load to monitor and throttle their request rate to avoid device overload and potential fabric congestion. The rate throttling specifically targets devices, mainly memory devices, that are associated with a host in the current design. In addition, QoS telemetry includes a mechanism called Egress Port Backpressure (EP Backpressure). It monitors the flow control backpressure situation on each CXL switch egress port. If the port cannot transmit packets for a period of time due to a lack of credits, it marks the EP Backpressure value with 2 bits in the device load field of the outgoing request. The overall load of a device is determined by the maximum of the device's internal load and EP Backpressure.

4.1.1 What is Different with CXL Fabric?

CXL fabric exposes memory traffic that used to be internal to a server to all endpoints connected to the fabric. The memory traffic, such as cache coherence, memory access, and I/O-style accesses, runs between processor and device endpoints on the CXL fabric. This contrasts with standard datacenter traffic, which runs with encapsulated packets from per-server NICs outside the internal memory fabric of a server. Processors and accelerators access remote devices using load/store instructions through the CXL fabric. They synchronously request data from remote devices, such as memory expansion modules, accelerators, and storage devices. However, these synchronous data requests cannot tolerate significant latency, as it will stall the execution of the requesting hardware while awaiting the requested data. This poses stringent latency requirements for the fabric and necessitates proper system-level support. Additionally, the CXL fabric supports up to 4096 endpoints. Given the scale of thousands of endpoints and the mixture of memory traffic, this introduces a challenge to the scalability of the underlying protocol design. A centralized scheduler is a possible solution for tens or even hundreds of endpoints on racks. However, the scheduler is very likely to become a performance bottleneck because it needs to sustain and determine the order of every load/store instruction. The scheduler can also become a single point of failure for all memory traffic. A centralized design for the scheduler limits the scalability of CXL, especially when using a longer-distance physical layer compared to PCIe. To understand the practical challenges, use cases of the CXL fabric from the CXL specification and the literature are discussed next [cxl, GLKJ22, LBN+23].

4.1.2 Use Cases of CXL Fabric

The use cases of CXL fabrics demand large memory capacity, high bandwidth, and low latency. Emerging and existing workloads in datacenters, such as machine learning models, large-scale key-value stores, and high-performance computing (HPC) applications, can benefit

from CXL fabrics.

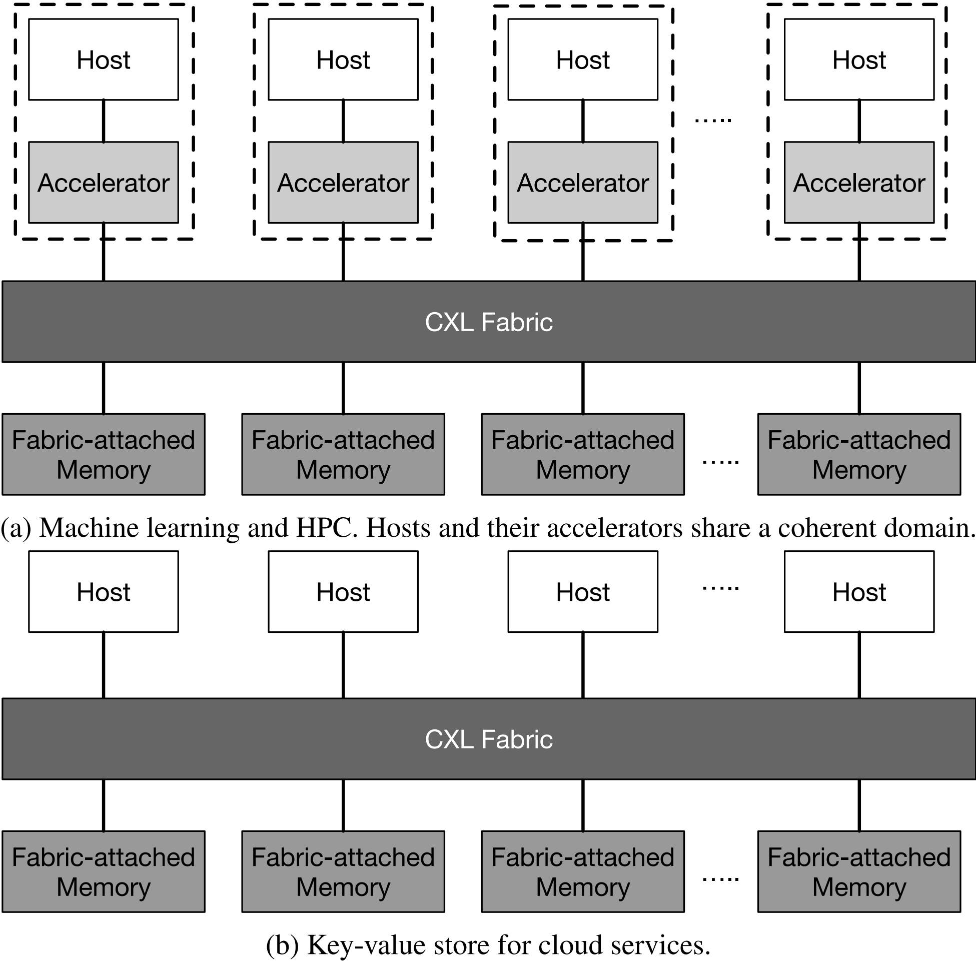
First, current machine learning models require a large amount of memory on an accelerator, such as a GPU or TPU, which is beyond the capacity of individual accelerators. To make matters worse, the size of state-of-the-art machine learning models ranges from tens of GB to tens of TB [RRRH20, RRR$^+$21, MHH$^+$22] and continues to grow every few months. Training and inference of machine learning models now require multiple accelerators to jointly fit the model and intermediate variables into their memory. The CXL fabric could expand accessible memory for accelerators by providing fabric-attached memory. Fabric-attached memory increases the memory capacity and bandwidth to all available memory on the fabric [cxl, PKK$^+$22, mem]. A host is connected with an accelerator with CXL, and they share a coherence domain (Shown in Figure 4.1a). Accelerators access the fabric-attached memory and each other's memory in a producer-consumer fashion of I/O coherency.

Second, datacenters run cloud services with key-value stores. Many high-performance key-value stores use RDMA inside data centers to speed up communication and operations [DNCH14, KKA14, KKA19, WCC20]. These operations are sensitive to latency and are on the performance-critical path of applications. DirectCXL demonstrated that CXL has 8.3x lower latency than RDMA for 64B reads and incurs less overhead when replacing RDMA [GLKJ22]. DirectCXL provides a lower bound on CXL latency because its fabric uses a single switch and is not subject to stress or congestion. Hosts do not maintain cache coherence between themselves. Instead, the fabric-attached memory module maintains coherence between itself and the host address space it has mapped. (Shown in Figure. 4.1b).

Third, HPC workloads demonstrate high utilization (greater than 90%) of memory bandwidth as well as capacity for representative applications [doe, cro, exa]. However, each application reaches its peak memory usage for different durations. The cluster must be provisioned to accommodate peak bandwidth and capacity to avoid significant slowdowns [Rad19, Ham20, DMN$^+$23].

Machine learning and HPC workloads both involve collective communication, while

key-value stores involve bursty, non-structural communication. Collective communication is structural and can be optimized with data prefetching to minimize stalls on pending memory accesses. This relaxes their requirements on latency and reduces burstiness. Training and inference of machine learning models use all-reduce operations to update model weights across each accelerator after each iteration. The size of state-of-the-art models ranges from tens of GB to tens of TB [RRRH20, RRR+21, MHH+22]. HPC workloads, compared to machine learning, have a more diverse communication pattern, such as sweeping or nearest-neighbors [LMM+19, emb, exa, doe]. Key-value stores and databases, however, serve bursty requests and are sensitive to latency [NFG+13, CD20].



(a) Machine learning and HPC. Hosts and their accelerators share a coherent domain.

(b) Key-value store for cloud services.

Figure 4.1: CXL fabric abstractive topology. Each solid line connecting to CXL fabric is 16 lanes of PCIe 5 or PCIe 6 with a total bandwidth of 128 GB/s or 256 GB/s.

4.2 Challenges

The current design of the CXL fabric [cxl] poses challenges regarding scalability and latency. First, its addressing and routing design limits the possibility for flexible and dynamic routing. Second, the lack of an end-to-end transport layer in the fabric makes it prone to congestion and latency spikes. More importantly, with the use of load/store instructions, processors and accelerators that synchronously request data are highly sensitive to latency, as it determines how long they need to stall their execution. The challenges related to addressing, routing, and transport layers are discussed in the following subsections.

4.2.1 Addressing and Routing Challenges

CXL routes packets with a per-device ID called Port ID on the fabric. The Port ID-based routing (PBR) assigns each device a 12-bit ID. A packet with PBR contains a specific source port ID and destination port ID before it leaves an edge CXL switch that connects directly with devices. Each CXL fabric has a single fabric manager to initialize, bind/unbind devices to ports, and handle event notifications, such as the removal or failure of devices, from the switch. The fabric manager is similar to a centralized software-defined network controller as it controls the per-port forwarding and is aware of all the route changes. However, the CXL fabric has (1) a limiting addressing scheme that is hard to support multi-path and adaptive routing, and (2) single-path and inactive routing regardless of the traffic condition.

Challenge: Limited addressing support for multi-path and adaptive routing. The current PBR routing scheme assigns an ID to devices only. PBR routes packets to the destination device through multi-level switches with routing installed by the fabric manager. Any routing reconfiguration needs to go through the fabric manager, making load-aware, adaptive routing inefficient and infeasible on a large scale. The centralized routing of PBR prevents the usage of classic multi-pathing techniques like packet spraying or ECMP because all the routes are

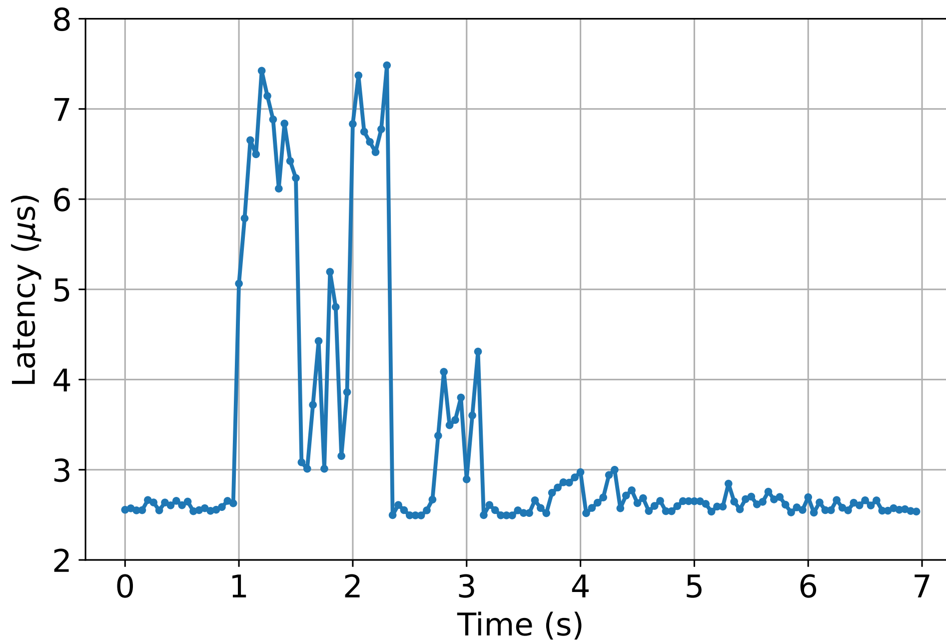


Figure 4.2: Congestion on a shared PCIe switch port causes latency spikes of RDMA writes going through the port.

pre-determined by the fabric manager. Figure. 4.4 shows an example of multi-level CXL fabric.

Challenge: inflexible routing over diverse topologies. CXL fabric supports flexible topology and thus opens the possibility of having a wide range of topologies such as a fully connected graph, fat-tree [AFLV08], Dragonfly [KDSA08], or reconfigurable topology [OYQ+19]. These topologies provide multiple paths for a source and a destination, but CXL fabric cannot route packets over multiple possible paths given its current design.

4.2.2 Transport-level Challenges

CXL inherits point-to-point flow control from PCIe, which was designed for the communication between the device and CPU rather than for a fabric. The flow control operates between two directly connected endpoints. They exchange credit tokens to evaluate the available buffer space on each side.

Challenges: Flow control cannot prevent congestion. The point-to-point, credit-based flow

control is focused on preventing buffer overruns only. It cannot handle pairs of endpoints sharing a port on the switch because no information is exchanged between them.

PCIe congestion experiment. We design an experiment of multiple flows sharing a port on a switch, and creating congestion. Given the lack of commercially available hardware for CXL 3.0, we use PCIe, which shares the same flow control mechanism, for our experiment. Interestingly, PCIe congestion has been identified and demonstrated under various setups [KM05, MKA+16, TWZL21].

We create artificial PCIe congestion on a PCIe switch port shared by two devices. The machine uses a SuperMicro X11SPA-TF motherboard with a Broadcom PEX8747 PCIe switch. The PCIe switch has one upstream port to the CPU and two downstream ports connecting to the devices: An Nvidia 2080 Ti GPU and a ConnectX-5 RDMA NIC. The PCIe switch connects with

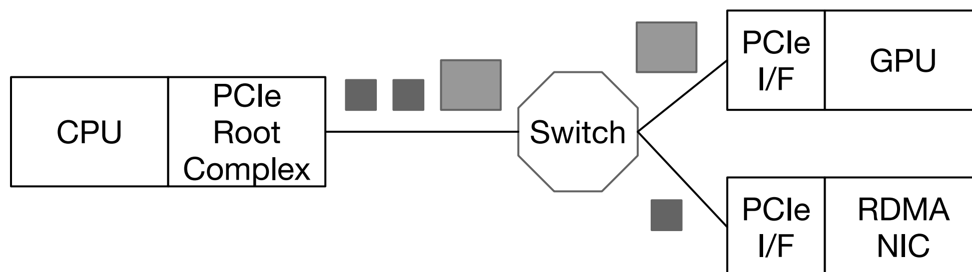


Figure 4.3: Experimental setup for PCIe congestion.

the host processor on one side and provides two ports connecting to GPU and RDMA NIC. The experimental setup is shown in Figure 4.3. During the experiment, the NIC periodically sends RDMA write requests to another machine every 100 microseconds. Each write request is sent from the CPU and travels through the PCIe switch to the NIC. After a second, a large integer array of 200 MB is moved from the main memory to the GPU. It causes heavy traffic on the upstream port and the downstream port to the GPU. Their traffic collides on the upstream port of the PCIe switch because flow control does not account for congestion caused by an outgoing link.

We use PerfTest of Linux-RDMA library to measure the RDMA write request latency [ofe] shown in Figure 4.2. The latency spikes to almost 3x from 2.593 μs to 7.483 μs at the peak of PCIe

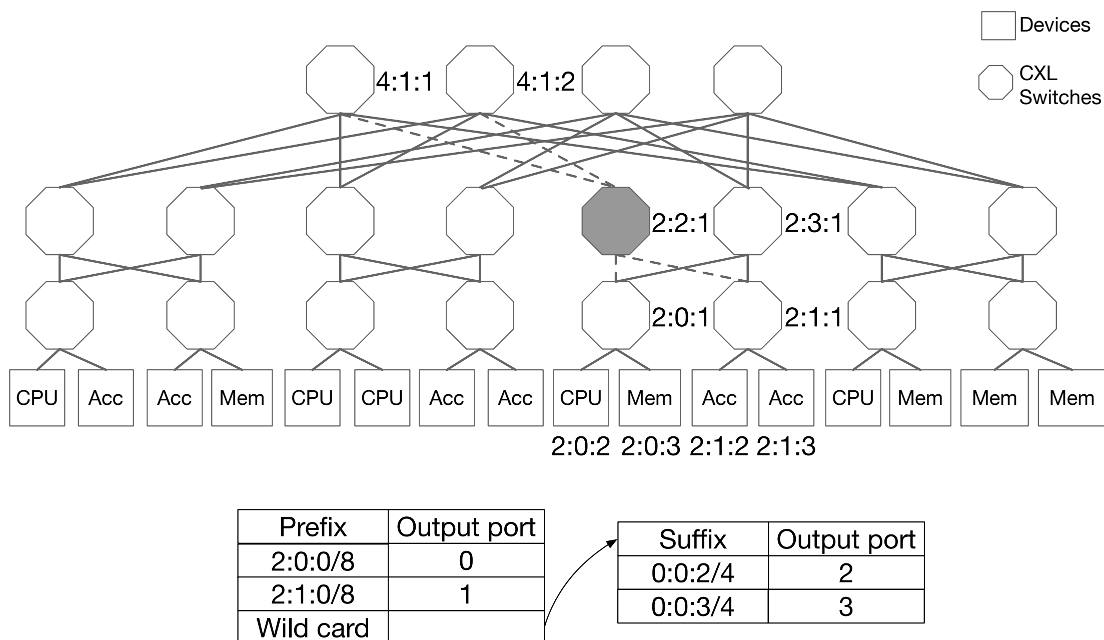
congestion. PCIe's virtual channel is a feasible but not scalable solution because it provides only 7 channels. Traffic on the same channel still suffers from the same congestion as demonstrated above.

Challenge: Rate throttling between host CPU and devices only: CXL fabric offers a rate throttling mechanism called QoS Telemetry to avoid device overload and possible fabric congestion. The current QoS telemetry is designed for CXL.mem between the host and devices with their local memory specifically. However, machine learning and HPC applications illustrated in Figure 4.1 rely on unordered I/O of CXL.io for peer-to-peer memory accesses. QoS telemetry is much needed for these two use cases like key-value stores using CXL.mem because the CXL fabric supports all CXL protocols to mitigate congestion and device overload.

Challenge: Inaccurate load & congestion information. QoS telemetry devises a mechanism called Egress Port Backpressure (EP Backpressure) to indicate the load of CXL switch ports. It monitors the flow control backpressure situation on each CXL switch egress port. If the port cannot transmit packets due to insufficient credits, the port marks a 2-bit EP Backpressure value on the device load field of the outgoing request. Also, each device reports its internal load. However, QoS telemetry does not distinguish the backpressure on fabric and the device's internal load. This prevents QoS telemetry from describing the load on fabric and device end points accurately and separately.

4.3 Design of Aurelia

We propose Aurelia, a network design involving devices and switches on the CXL fabric. Aurelia provides addressing, routing, and congestion control protocol design by augmenting necessary functionalities on the fabric interface and switches. For the rest of Sec. 4.3 and further sections, non-oversubscribed fat-tree is assumed as the default topology to demonstrate the primitive addressing and routing design of Aurelia. Different topologies with their related costs



Prefix	Output port
2:0:0/8	0
2:1:0/8	1
Wild card	

Suffix	Output port
0:0:2/4	2
0:0:3/4	3

Figure 4.4: CXL fabric as a fat-tree using 12-bit FAN-ID address *X:Y:Z*, which X, Y, and Z are hexadecimal values. Dash lines represent the routes from switch *2:2:1*'s routing table.

and configurations are discussed in Sec. 4.5.

4.3.1 Addressing & Flow

Aurelia proposes to generalize CXL's port ID scheme by assigning a 12-bit fabric node ID (FAN-ID) to a node that is either a device or a switch on the fabric. This is analogous to the 32-bit IP address describing hosts in an IP network. FAN-ID assignment is agnostic to the underlying topology as long as a FAN-ID is unique for a node on a CXL fabric.

With a similar analogy to "5-tuple" in the IP network, Aurelia is able to defines a flow by a sequence of CXL packets that share the same source device, destination device, protocol, and message classes. Thus, routable CXL packets can be identified with a quadruple: *(Source FAN-ID, Destination FAN-ID, CXL protocol, message classes)*. The CXL protocols include CXL.io, CXL.mem, and CXL.cache. The message classes are subtype of packets within each protocol. For example, a CXL.mem packet writing to a device belongs to a class of *M2S Request*

with Data and a CXL.cache packet that carries responses from the device to the host belongs to a class of *D2H Response with Data*.

4.3.2 Routing

Aurelia routes packets based on FAN-ID addresses similar to IP routing. CXL switches route a packet based on its routing table that maps a group of FAN-ID destinations onto a port. CXL switches transmit the packet out of a specific port to another switch that knows the next hop for this packet. The forwarding continues until the packet reaches the FAN-ID destination.

Routing: Fat-tree as an example. Constructing a fat-tree [AFLV08] with 12-bit FAN-ID address shows a k-ary fat-tree with $k = 4$ in Figure 4.4. Fat-tree uses a hierarchical scheme that assigns FAN-ID to nodes as *pod:switch:device* with three segments. Each segment is a 4-bit hexadecimal value. For example, the leftmost CPU has FAN-ID *2:0:2* representing it belongs to the third pod, the first switch, and the third node, which is right after the switch itself. Similar to IP routing, the routing on fat-tree follows a single shortest path despite that the fat-tree topology provides path diversity. This creates potential bottlenecks even for trivial communication patterns because of the underutilization of available bandwidth. The fat-tree implements a two-level routing table on each switch. One level of the table routes traffic down to the device, while the other routes traffic toward the core of the fabric. The table maps a set of destinations to a specific port. The table maps a set of destinations to a specific port. The routing lookup for destinations uses the address prefix for traffic downward to the devices and the address suffix for traffic going toward the cores. The address suffix approach spreads traffic toward the fat-tree core across different switches based on FAN-ID. The bottom of Figure 4.4 shows the routing table of CXL switch *2:2:1* filled in gray. The prefix table routes packets toward its downstream switches *2:0:1* and *2:1:1*. The suffix table routes the packet upward to core switches *4:1:1* and *4:1:2*.

Multi-path routing. The two-level routing table of the fat-tree is an implementation of multi-path routing tailored to a specific topology. Aurelia imposes no restriction on how the fabric should

be constructed. Instead, Aurelia is able to have routing tables that map a destination to multiple next-hop FAN-ID and its corresponding metric. These metrics can include distances in terms of hops or local congestion level on each switch port. Aurelia selects a next-hop with minimal metric and randomly selects one if there are multiple next-hop options with equal metrics. Considering the number of hops for shortest path routing, Aurelia can perform a random selection for the next hop on flow granularity or on packet granularity, i.e. Equal-Cost Multi-pathing (ECMP), or at the packet granularity, similar to packet spraying.

Adaptive routing. ECMP and packet spraying are oblivious to the workload. Aurelia is able to to achieve adaptive routin by further exploiting local per-port information on the switches or by manipulating the routing table on the fabric manager. Aurelia uses per-port EP Backpressure as a measure of local congestion. For selecting an egress port with minimal congestion on the switch, Aurelia uses power-of-2 choice [Mit01] to avoid congestion on ports based on possibly delayed congestion information. Additionally, Aurelia allows the fabric manager to insert routes and has the sole authority to modify the routing table at any time. This is useful for the workload that requires non-trivial routing tailored to specific workloads [TLG+15, VMB+22].

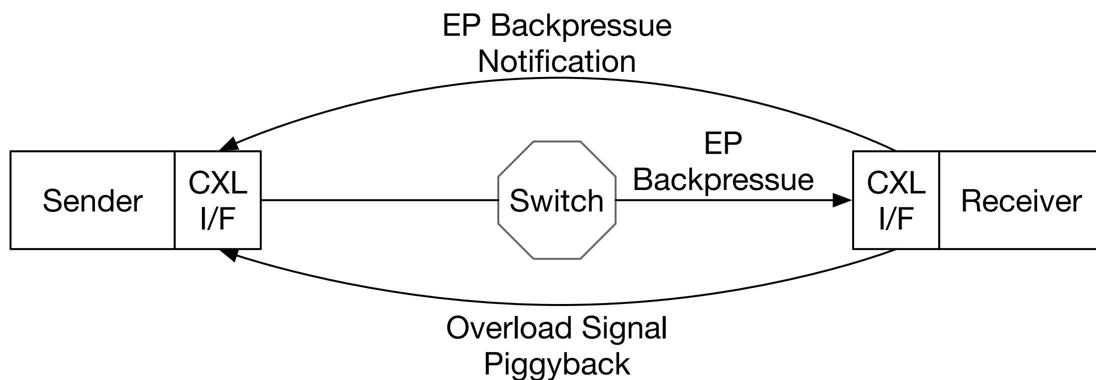


Figure 4.5: Aurelia uses EP Backpressure notification to resolve congestions and overload signal for flow control to avoid overrunning the buffer on the device.

4.3.3 End-to-end Congestion Control

CXL relies on point-to-point flow control that is prone to congestion and lacks a congestion control design for the fabric. As stated in Sec. 4.2.2, CXL's QoS telemetry and EP Backpressure partially mitigate congestion for specific scenarios.

QoS telemetry currently supports only CXL.mem protocol between the host and devices with their locally attached memory. QoS telemetry enables memory devices to indicate their current internal load with 2 bits for CXL.mem response packets. The sender on the host CPU uses this reported internal load to monitor and throttle its request rate. However, not every sender on CXL fabric is on the host CPU. Peer-to-peer memory accesses using Unordered I/O of CXL.io allow devices to access the memory of another device directly. These peer-to-peer accesses under CXL.io are not throttled in the current design because QoS telemetry is limited to CXL.mem and does not consider the sender to be other than a host CPU.

EP Backpressure is not effectively utilized in the current design of QoS telemetry, because QoS telemetry determines the overall load of a device by taking the maximum of the device's internal load and EP Backpressure. This approach subsumes EP Backpressure that offers valuable congestion information on the fabric. By simply taking the maximum value of the separated piece of information, QoS telemetry provides neither accurate device internal load nor congestion on the fabric. This inaccuracy hinders QoS telemetry's ability to perform effective end-to-end flow control and congestion control for the transport protocol. The aforementioned challenges motivate us to design Aurelia.

Extending existing mechanisms. Aurelia implements end-to-end congestion control with overload avoidance by extending the existing mechanisms of CXL. The mechanisms are EP Backpressure on switches, internal load reported on receiving devices, and rate throttling on sending devices. EP Backpressure triggers switches to mark packets on their way to the receiving device when noticeable queuing occurs on the switch. This is similar to Explicit Congestion Notification (ECN) for Ethernet and Infiniband because they react when the queuing situation

begins to indicate congestion. Internal load reporting triggers an overloading signal when the device is overloaded. Device overload is likely to cause significant device-side delay or even loss of CXL packets. Aurelia, unlike CXL's design, separates EP Backpressure and the device's internal load since they represent fabric and device information. Rate throttling controls the sending rate into the fabric to avoid congestion and overload.

Algorithm. Aurelia's congestion control algorithm operates on the switch, the receiving device, and the sending device. On the switches, Aurelia mandates the switches to mark the packets when the ratio of EP Backpressure events in their recent window is larger than a specific threshold. On the receiving devices, Aurelia makes the receivers piggyback an overloading signal to the sending device when its sending rate exceeds the receiver's processing rate. The receiving device notifies the sender with EP Backpressure notification (EPN) in its response to the sending device when it receives EP Backpressure marked packets. On the sender side, the sending device throttles the sending rate based on the 1-bit signal of EPN and the 1-bit overloading signal. When the sender receives a response packet marked with EPN or an overloading signal, the sender records its current rate R_c as the target rate R_t for later recovery and cuts its rate half by default. The rate cut can also be determined by also a rate reduction factor α, similar to DCTCP and DCQCN [ZEF+15]. The recovery of the reduced sending rate has two different paths depending on the trigger. If the reduction is triggered by EPN, then the recovery to the target rate R_t is expected in a fixed number of iterations. If the reduction is triggered by an overloading signal, then the recovery is an additive increase.

Protocol implementation. Aurelia relies on hardware implementation to throttle sending rate for congestion control due to sub-μs latency and peer-to-peer memory access requirements. First, rate throttling has a tight latency requirement of sub-μs on CXL fabric. Pond measured end-to-end CXL.mem latency and obtained latency ranging up to 270 ns on CXL system with a single switch [Sha22, LBN+23]. Under the same assumption, the latency of CXL packet traveling through a two-level fat-tree takes around 680 ns, which is under 1 μs. Second, peer-

to-peer memory access between devices is expected, especially with the usage of accelerators. Therefore, Aurelia extends CXL's original design to generalize the reporting of device load for all kinds of memory access, including peer-to-peer memory access between devices. Peer-to-peer memory access between devices without an additional embedded CPU motivates the necessity of implementing rate throttling in hardware on the CXL interface. The control plane of these hardware devices is kept on the host CPU, but the execution of rate throttling is on hardware to meet these requirements. Hardware-based rate throttling ensures all devices on the CXL fabric are able to control their sending rate and further reduces the congestion and overload. Implementing rate throttling logic on hardware has been used on RDMA over Infiniband and Ethernet with RoCEv2 support and thus suggests the feasibility in the case of CXL fabric.

Comparable protocol design on lossless fabric. Aurelia's congestion control design is inspired by congestion control on existing lossless fabric, such as Infiniband and DCQCN for RoCEv2, which implemented a rate-based congestion control algorithm requiring Explicit Congestion Notification (ECN) on switch to indicate congestion. Infiniband switches mark packets with a Forward ECN (FECN) bit when congestion is detected [inf]. The receiver receives the FECN-marked packets and sends a Backward ECN (BECN) marked packet to the sender. The sender throttles its injection rate when it gets packets with BECN. The throttling over time reduces congestion and the fabric returns to a state without any congestion. The FECN marking of packets requires switch support and the implementation of BECN and rate throttling requires additional logic in CXL interface hardware. DCQCN, a congestion control protocol designed for RoCEv2, relies on the ECN of Ethernet to notify the sender to adjust its injection rate, similar to Infiniband [ZEF+15].

4.4 Evaluation of Aurelia's Design

We simulate the design of Aurelia because CXL hardware supporting CXL 3.0 with fabric is not available to the author at the time of writing. We evaluate Aurelia's congestion control design using routing described in Sec. 4.3.2. Additional routing designs, such as multipath and adaptive routing, are not enabled for the evaluation. The evaluated workloads are: (1) key-value store on memory expansion modules [PKK+22], and (2) machine learning model inference on GPUs/accelerators involving collective communication [awsa]. They represent different uses of CXL fabric: one focuses on CPU and memory expansion, while the other focuses on the data exchange between accelerators. The baseline of the evaluation is vanilla CXL described in its specification [cxl]. It performs point-to-point flow control for packets of each CXL protocol. It does not employ any end-to-end congestion control or flow control mechanisms.

4.4.1 Packet-level Simulation using ns-3

We use NS3 [ns-] to simulate packet level behavior on a CXL fabric. NS3 is a discrete-event network simulator that is open-sourced and well-established in the research community. Our implementation uses primitive, built-in NS3 classes and starts from scratch because there is no existing simulator for the CXL fabric. Previous simulators of other lossless fabrics focus on RDMA over Converged Ethernet (RoCE) [ZEF+15, LML+19, BBRL+20]. These implementations are not usable for CXL fabrics because they assume the use of Ethernet, IP, and the RDMA interface cards. CXL fabrics assume none of those, as the devices on the fabric issue load/store instructions directly.

The hardware parameters are cross-checked with published literature [LBN+23, SYY+23, h3p]. The latencies on each hardware component are calibrated with H3 platform's CXL expansion chassis supporting CXL 2.0 [h3p]. The chassis integrates a CXL switch [xco] supporting CXL.mem and CXL.io that are used in the evaluation.

We simulate CXL's transaction layer and link layer in 256B mode assuming a PCIe 6.0 physical layer. The physical layer of PCIe 6.0 with 256 GB/s is on a 16-lane configuration. All links in the simulation use this configuration. All messages classes of CXL.mem and CXL.cache are supported. CXL.io functionalities for peer-to-peer memory access are supported. Packing of CXL.mem and CXL.cache messages is implemented to ensure the number of packets on the fabric is accurate to the specification.

The simulator enables multi-level switching and implements Aurelia's addressing, routing, and congestion control mechanism on top of CXL's port-based routing. There are 16 nodes on a 2-level fat-tree topology. In the case of machine learning model inference, the topology connects 8 accelerators and 8 memory expansion devices. In the case of the key-value store, the topology connects 12 memory expansion devices with 4 CPUs.

4.4.2 Simulation of Large Model Inference

We study the improvement of inference throughput on the LLaMA2 70B model [TLI+23] due to improved congestion control on the CXL fabric. The inference of the model operates on the unit of tokens for a sequence of input and output data. First, the fabric connects the accelerators to the memory expansion devices that store the model weights. Partial weights of the model are kept on the accelerator as the weights are loaded partition by partition. Each accelerator stores an 8.125 GB partition on the device and another partition on the memory expansion devices. This leaves enough capacity for intermediate generated tokens on the accelerators. A run of inference performs 2 weight loading operations between the accelerators and the memory expansion devices. Second, the fabric supports the communication among accelerators that aggregates tokens. A run of inference performs 2 operations of all-reduce collective communication among the accelerators to aggregate the intermediate and the output tokens. Each all-reduce operation exchanges 810 $\times$ 8 MB of data among the accelerators. The all-reduce operations are 3 milliseconds apart as these durations represent the computational part of the inference run. The simulation executes

1000 runs of inference based on the aforementioned parameters. Aurelia's improved congestion control aims to speed up both the weight loading and the all-reduce operations.

Aurelia achieves 21% improvement on median throughput and 27% improvement on the 10th percentile throughput because of faster weight loading and all-reduce operations. These operations encounter 2.1× fewer pauses on the links than the baseline when the links are out of credits to send packets. EPN of Aurelia throttles the sending rate to avoid triggering pauses on links. The traffic pattern of weight loading and all-reduce operations does not create much imbalance therefore there is no overload on any of the accelerators that requires an overload signal for remediation. Imbalanced traffic among accelerators could test Aurelia's effectiveness on such kind of traffic pattern. We leave it for future investigation.

4.4.3 Simulation Results on YCSB Benchmarks

We study the 99th percentile latency on key-value stores using YCSB benchmarks [CST+10]. YCSB benchmarks are widely used to evaluate key-value stores with a mix of insert, read, and update operations in memory or on SSD. In the case of CXL fabric, we evaluate key-value stores on fabric-attached memory expansion. Memory expansion devices manage their own coherence between themselves and connected CPUs. They invalidate data in the CPU cache to maintain the coherency.

We evaluate YCSB A, B, C, and F for read, write, and read-modify-write operations on the key-value store. YCSB D and E are not evaluated because their performance depends on the recently accessed keys and also how the keys are hashed when inserted. These additional requirements make them less ideal for us to objectively evaluate the improvement provided by Aurelia. YCSB A, B, C, and F, on the other hand, access key-value pairs based on a fixed Zipf distribution. The Zipf distribution makes the probability of accessing n^{th} key inversely proportional to n. The plot of a Zipf distribution in linear scale is similar to an exponential decay curve so the first few n^{th} keys are way more likely to be frequently accessed. This makes the

traffic imbalanced because the requests to a key-value pair are more likely to be headed to a certain memory expansion device.

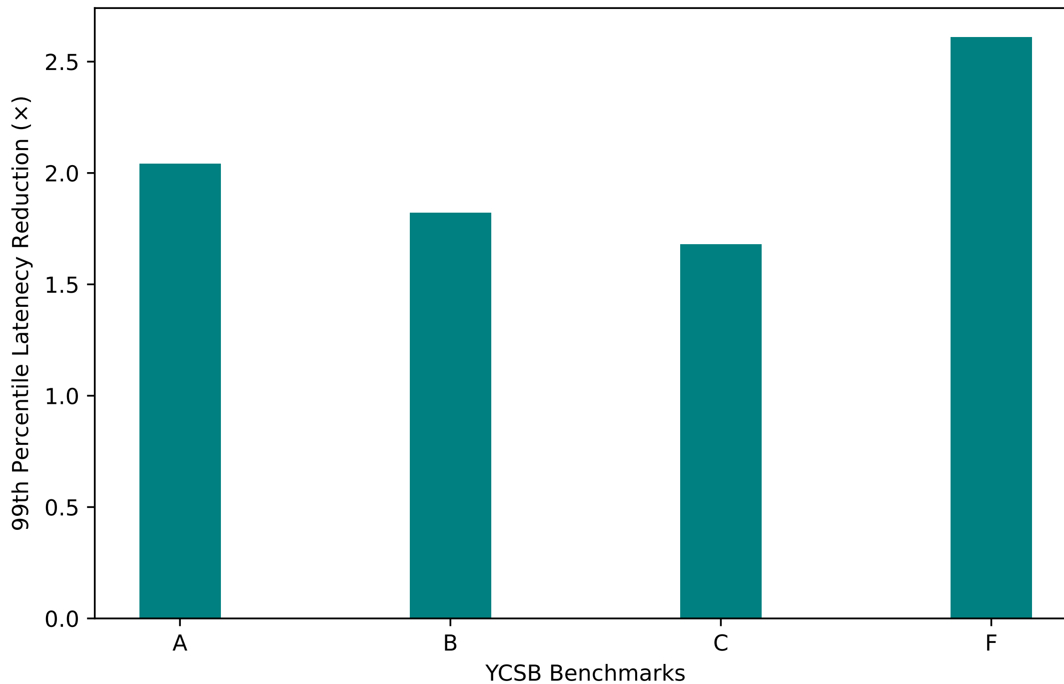


Figure 4.6: YCSB benchmarks with higher ratio of writes demonstrate more improvement.

Aurelia's improvement on 99th percentile latency for YCSB benchmarks is shown in Figure 4.6. Aurelia's EPN and overload signal combined handle the imbalance of traffic caused by Zipf distribution. The improvement for read-only YCSB C is the lowest because it incurs a request-response pair to a specific memory expansion device without any cache invalidation. The improvement for YCSB A and B is higher because YCSB A and B incur write and subsequent cache invalidation from the memory expansion device to other CPUs. The improvement of YCSB A is higher than YCSB B because it has 50% of operations as writes operations compared to 5% of YCSB B. EPN has a higher contribution to the latency improvement because the overload signal contributes 22% more improvement on YCSB F compared to Aurelia without it. This is expected as overload should not be a frequent event unless memory capacity and bandwidth are not properly provisioned On average, Aurelia without overload signal is only 14% worse on all

YCSB benchmarks.

4.5 Discussion

Aurelia demonstrates a fabric for disaggregation based on the CXL specification [cxl]. However, CXL is not the panacea for every issue and challenge in the dataceneters. First, alternative fabrics like NVLink for GPUs and PCIe for existing devices are discussed with their advantages over CXL. Second, CXL is limited to a shorter range of reach compared to Ethernet. With CXL's limitation in the physical layer from PCIe, Ethernet is still much needed for inter-rack communication. Third, CXL fabric flexibility on topology motivates us to discuss multi-pathing strategy based on specific topologies. Lastly, CXL externalizes system interconnect to connect more devices than ever and thus further exposes itself to more potential side-channel attacks

Alternatives: NVLink and PCIe fabric. NVLink is an interconnect by Nvidia for high throughput between GPUs [nvl]. NVLink supports GPU-CPU interconnect with cache coherency on the recent Grace-Hopper 200 hardware [dgxa]. NVLink presents an interesting alternative to CXL but has three limitations. First, GPUs using NVLink with cache coherence are equivalent to CXL type 2 devices. However, NVLink does not support CXL type 3 devices, which expand memory independently of the main memory capacity. The capability to scale memory capacity is attractive for memory-hungry large language models [BMR+20, TLI+23]. Second, NVLink scales to 256 endpoints but connects only GPUs as endpoints [dgxb]. Third, NVLink as a proprietary interconnect limits its usage beyond Nvidia's hardware. PCIe using Non-Transparent Bridge (NTB) expands beyond a single root complex to multiple hosts as a fabric with many more connected devices [pcia]. However, PCIe, included in CXL as CXL.io, lacks the memory and caching semantics that are much needed in the face of accelerators and memory expansion. This is exactly the reason that motivates the creation of CXL. Both NVLink and PCIe fabric support a subset of CXL's use cases but do not support all of them.

Co-existence of CXL and Ethernet. Given the scaling discussion, CXL is still unlikely to completely replace Ethernet in the datacenter due to its current PCIe based physical layer design. We speculate CXL fabric is beneficial on a rack-scale due to its signal integrity and CXL switch hardware cost. CXL fabric is not intended to replace existing Ethernet completely but to be a cost-effective alternative at the rack level. CXL packets are converted to Ethernet frames at a location equivalent to a top-of-rack switch for cross-rack traffic. Previous work investigating the co-existence of Ethernet and a memory fabric follows a similar approach [GHL+22].

Cost-effective scales of CXL fabric: CXL is an exciting technology, but it comes with its limitations. First, CXL requires a retimer to maintain its signal integrity after a 500 mm distance [micb, ast]. The retimer raises the hardware cost and incurs an extra 20 ns latency every 500 mm [LBN+23]. Second, to scale CXL supporting multi-level switching, multiple CXL switches are used. They cause an estimated 70 ns latency for each hop over a switch. Considering the combined cost of switches, retimers, and additional memory controllers, Pond [BEL+23] suggests a scale smaller than 32-socket for their memory expansion usage (shown in Figure 4.1b). However, the exact scale of cost-effective CXL fabric for model training and HPC usage (shown in Figure 4.1a) remains to be determined in future work.

Topologies and multipathing. The flexible topology of CXL fabric empowers the fabric operator to optimize their topology based on their cost consideration and traffic pattern. Though fat-tree offers straightforward addressing and routing, it requires many switches and incurs high wiring costs. Topologies require fewer switches and less wiring, e.g. Dragonfly or other low-diameter alternatives, could be considered if cost is the primary concern. The recently released CXL 3.1 supports native multipathing. Topologies with high path diversity, such as Dragonfly, can be easily implemented with CXL's multipathing. Packet spraying and ECMP on fat-tree-related topologies can benefit from this as well. With CXL switch prototype supporting 32 ports [xco], we expect these higher radix switches to enable more design options. For example, the fabric operator can assign a high over-subscription ratio if the traffic pattern demonstrates high locality

under a CXL switch. As another example, the fabric operator can add or reduce the bandwidth of certain CXL links to better match the bandwidth to the actual demand between particular nodes

Security implication: Delay based side channel. CXL fabric exposes the server interconnect to a shared fabric that may contain malicious devices. The fabric exposes a delay-based side channel caused by interconnect congestion. This side channel enables attackers to recover information from different delay patterns. Previous work [TWZL21] investigate the same issue with PCIe on a server. CXL fabric enlarges the attack surface by leaving all devices vulnerable to delay probing. Delay probing requires accurate timing measurement. CXL's use of direct load/store makes it easy for an attacker to acquire accurate timing of every memory load/store. Defense against this specific type of side channel attack is interesting for future security research.

4.6 Conclusion

Emerging standard CXL facilitates disaggregation with the support of multi-level switching. However, the current CXL fabric presents scalability and latency limitations. Aurelia addresses these challenges by designing effective addressing, routing, and transport layer design.

4.7 Sources for Material Presented in This Chapter

Chapter 4, in part, reprints material as it appears in a published WORD'23 paper titled: "Aurelia: CXL Fabric with Tentacle" by Shu-Ting Wang and Weitao Wang [WW23]. The dissertation author was the primary researcher and author of this material.

Chapter 5

Conclusion

The dissertation investigates the redundant communication between servers for large-scale web and cache requests and redundant data movement between accelerators for compute-intensive applications. The redundancy is an impending and critical issue for datacenters designed for hardware accelerators and disaggregated resources. The dissertation makes the following three contributions to address this.

The first contribution of the dissertation is Daronpon. Daronpon is a datacenter-wide, inter-rack distributed load-balancing system for replicated datacenter services. Daronpon, at its heart, is a highly efficient gossip protocol that ensures that requests originating at any point in the datacenter can be directed away from overloaded replicas at sub-RTT timescales. Through a mixture of simulation and deployment within AWS's cloud network, we show that Daronpon reduces the 99th percentile of latency for common workloads by up to $2\times$ while admitting 10% more requests per unit time.

The second contribution of the dissertation is Fianchetto. Fianchetto acts as a compute-enabled bypass for inter-accelerator communication. The data restructuring and communication overhead of executing a single application using a chain of accelerators is defined as the data motion overhead. With the current paradigm of using accelerators, the data motion overhead

is very likely to outweigh the benefits from all these chained heterogeneous accelerators. In contrast to previous works on accelerators that deal with accelerating only compute kernels, Fianchetto focuses on accelerating data motion within a chain of heterogeneous accelerators in a multi-accelerator datacenter. To that end, Fianchetto reduces data movement, accelerates data restructuring, and enables interoperability between heterogeneous accelerators from different domains through a cross-stack hardware-software solution. The results from five end-to-end applications show that utilizing Fianchetto offers up to 8.2×, 13.6×, and 5.2× improvement in latency, throughput, and energy efficiency in a multi-accelerator system, respectively.

The third contribution of the dissertation is Aurelia. Aurelia leverages the emerging interconnect of CXL to investigate the design of a scalable fabric for accelerators and fabric-attached memory expansion. Compute Express Link (CXL) has emerged as a frontier for disaggregation by providing a fabric supporting memory accessing, caching, and peer-to-peer memory access between devices. CXL externalizes the internal memory fabric of a server and blurs the notion of server for realistic disaggregation. The key feature of enabling CXL as a memory fabric is its support of multi-level switching up to 4096 endpoints. However, CXL's current multi-level switching poses challenges on scalability and latency. To cope with the expected scale of CXL fabric and take full advantage of disaggregation, we propose Aurelia. Aurelia architects addressing, routing, and transport as networking layers, which are typical in host networking, for CXL fabric. Aurelia uses existing CXL mechanisms to realize the much-needed functionalities for a scalable fabric. With these networking layers, Aurelia improves by 27% on throughput for large language model inference and up to 2.6 × for key-value stores on YCSB benchmarks.

Bibliography

[ABAL+20] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[ABR+20] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. Data compression accelerator on ibm power9 and z15 processors : Industrial product. In *ISCA*, 2020.

[ACFM16] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.

[AED+] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '14.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM Conference on Special Interest Group on Data Communication*, 2008.

[AKK+15] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. Dcs: A fast and scalable device-centric server architecture. In *MICRO*, 2015.

[ama] Amazon EC2 F1 Instances.

[AMH+] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16.

[APR20] Afzal Ahmad, Muhammad Adeel Pasha, and Ghulam Jilani Raza. Accelerating tiny yolov3 using fpga-based hardware/software co-design. In *IEEE ISCAS*, 2020.

[ast] Pcie retimer. `https://www.asteralabs.com/smart-retimers/pci-express-retimers-vs-redrivers-an-eye-popping-difference/`.

[awsa] AWS inferentia.

[awsb] AWS trainium.

[awsc] Aws vt1 instance.

[azu] Azure serverless. `https://azure.microsoft.com/en-us/solutions/serverless/#overview`.

[BBCS17] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless operating system integration of Peer-to-Peer DMA between SSDs and GPUs. In *ATC*, 2017.

[BBRL+20] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[BEHL+19] Subho Sankar Banerjee, Mohamed El-Hadedy, Jong Bin Lim, Zbigniew T. Kalbarczyk, Deming Chen, Steven S. Lumetta, and Ravishankar K. Iyer. Asap: Accelerated short-read alignment on programmable hardware. *IEEE Transactions on Computers*, 2019.

[BEL+23] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 2023.

[BGK+13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2013.

[BGP+19] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *ISCA*, 2019.

[Bis21] Arijit Biswas. Sapphire rapids. In *Hot Chips*, 2021.

[BMPR17] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, 2020.

[BPP+16] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), December 2016.

[bro] Broadcom pex88000 managed pci express 4.0 switches.

[BTY+] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20.

[BWPN18] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. Zeppelin: An soc for multichip architectures. In *IEEE ISSCC*, 2018.

[CCP+16] 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 *MICRO*, 2016.

[CD20] Zhichao Cao and Siying Dong. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, 2020.

[CDS+14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[CFM+17] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. Drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[CGG+12] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich cmps. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[CGH+18] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. Smem++: A pipelined and time-multiplexed smem seeding accelerator for genome sequencing. In *FPL*, 2018.

[CIP+21] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[CKB+20] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *MICRO*, 2020.

[CKL+22] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S. Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, Gagandeep Singh, Juan Gómez-Luna, Nour Almadhoun Alserr, Mohammed Alser, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping. In *ISCA*, 2022.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[CMM+22] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. Hardware acceleration of compression and encryption in sap hana. *Proc. VLDB Endow.*, 15(12):3277–3291, 2022.

[CQS+22] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. Democratizing domain-specific computing. *Commun. ACM*, 66(1):74–85, dec 2022.

[cro] Crossroads benchmarks. `https://www.lanl.gov/projects/crossroads/benchmarks-performance-analysis.php`.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[cxl] Cxl 3.0 specification. `https://www.computeexpresslink.org/download-the-specification`.

[CYES19] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *JETCAS*, 2019.

[CZY+18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, pages 3389–3400, 2018.

[Dah00] M. Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1033–1047, 2000.

[Dal23] Bill Dally. Accelerator clusters: the new supercomputer. In *HOTI*, 2023.

[DB] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80.

[DFC+15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, 2015.

[DGR+74] 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. *JSSC*, 1974.

[dgxa] DGX GH200. `https://www.nvidia.com/en-us/data-center/dgx-gh200/`.

[dgxb] DGX SuperPOD. `https://www.nvidia.com/en-us/data-center/dgx-superpod/`.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[dma] dma-buf.

[DMN+23] Nan Ding, Pieter Maris, Hai Ah Nam, Taylor Groves, Muaaz Gul Awan, LeAnn Lindsey, Christopher Daley, Oguz Selvitopi, Leonid Oliker, Nicholas Wright, and Samuel Williams. Evaluating the potential of disaggregated memory systems for hpc applications, 2023.

[DNCH14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, April 2014.

[doe] Characterization of the doe mini-apps. `https://portal.nersc.gov/project/CAL/doe-miniapps.htm`.

[DPD21] DPDK. Data plane development kit (DPDK), 2021.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[EFM+22] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. Flexdriver: A network driver for your accelerator. In *ASPLOS*, 2022.

[EGG+21] Hadi Esmaeilzadeh, Soroush Ghodrati, Jie Gu, Shiyu Guo, Andrew B Kahng, Joon Kyung Kim, Sean Kinzer, Rohan Mahapatra, Susmita Dey Manasi, Edwin Mascarenhas, et al. Verigood-ml: An open-source flow for automated ml hardware synthesis. In *ICCAD*, 2021.

[emb] Ember. `https://proxyapps.exascaleproject.org/app/ember-communication-patterns/`.

[exa] Ecp proxy applications. `https://proxyapps.exascaleproject.org/app/`.

[EYC+] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI ’16.

[fab] Fabric saving for gpu. `https://gigaio.com/2023/01/gigaio-doubles-gpu-performance-cost-savings/`.

[fac20] Facebook shard manager. https://engineering.fb.com/2020/08/24/production-engineering/scaling-services-with-shard-manager/, 2020.

[FOP+18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

[FSZ+18] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *ISCA*, 2018.

[FWO+20] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Seedex: A genome sequencing accelerator for optimal alignments in subminimal space. In *MICRO*, 2020.

[GAK+20] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO*, 2020.

[GHL+22] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[gig] Gigaio fabrex.

[GKK+23] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023.

[GLH+] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14.

[GLKJ22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[GLS+20] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *MICRO*, 2020.

[GLY$^+$] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20.

[gpu] Gpudirect.

[GYG$^+$] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. DRILL: Micro load balancing for low-latency data center networks. In *Proc. of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '17.

[GYP$^+$19] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *ASPLOS*, 2019.

[h3p] Falcon c5022 of h3 platform. `https://www.h3platform.com/product-detail/overview/35`.

[Ham20] Simon David Hammond. Compute memory trends: from application requirements to architectural needs. 2020.

[HBB$^+$18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[HFFA11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[HJZ$^+$13] Rui Hou, Tao Jiang, Liuhang Zhang, Pengfei Qi, Jianbo Dong, Haibin Wang, Xiongli Gu, and Shujie Zhang. Cost effective data center servers. In *HPCA*, 2013.

[HLL$^+$19] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm. In *MICRO*, 2019.

[HMCX22] Wenqin Huangfu, Krishna T. Malladi, Andrew Chang, and Yuan Xie. Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[HML$^+$20] Wenqin Huangfu, Krishna T. Malladi, Shuangchen Li, Peng Gu, and Yuan Xie. Nest: Dimm based near-data-processing accelerator for k-mer counting. In *ICCAD*, 2020.

[HMSA+21] Abbas Haghi, Santiago Marco-Sola, Lluc Alvarez, Dionysios Diamantopoulos, Christoph Hagleitner, and Miquel Moreto. An fpga accelerator of the wavefront algorithm for genomics pairwise alignment. In *FPL*, 2021.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[HWS+16] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, 2016.

[inf] Infiniband™ architecture specification. `https://www.infinibandta.org/ibta-specification/`.

[inta] Intel cascade lake.

[intb] Intel data streaming accelerator.

[intc] Intel ice lake.

[intd] Intel rapl.

[inte] Intel sapphire rapids.

[intf] Intel vtune profiler.

[JHYA+21] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *ISCA*, 2021.

[JLZ+] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'18.

[JPOB20] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4):58–66, 2020.

[Jun22] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *The ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2022.

[JYP+17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[KAC+18] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. Dcs-ctrl: A fast and flexible device-control mechanism for device-centric server architecture. In *ISCA*, 2018.

[KB] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20.

[KCH+19] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[KdCL+20] Dmitrii Krylov, Remi des Combes, Romain Laroche, Michael Rosenblum, and Dmitry V Dylov. Reinforcement learning framework for deep brain stimulation study. In *IJCAI*, 2020.

[KDH+15] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, June 2015.

[KDSA08] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *International Symposium on Computer Architecture (ISCA)*, 2008.

[KGH+] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proc. of the ACM International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '17.

[KGP+13] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 468–479, New York, NY, USA, 2013. Association for Computing Machinery.

[KHK+] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *Proc. of the Symposium on SDN Research*, SOSR '16.

[KIB] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20.

[KIMR20] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing. In *DATE*, 2020.

[KKA14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[KKA19] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, February 2019.

[KLK+21] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO*, 2021.

[KM05] V. Krishnan and D. Mayhew. Localized congestion control in advanced switching interconnects. *IEEE Micro*, 25(1):10–11, 2005.

[KPG+] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making rpcs first-class datacenter citizens. In *Proc. of the USENIX Annual Technical Conference*, ATC '19.

[KSK18] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ASPLOS*, 2018.

[KVHD] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proc. of the ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14.

[KXH+] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proc. of the EuroSys Conference*, EuroSys '18.

[LBB+22] Cedric Lichtenau, Alper Buyuktosunoglu, Ramon Bertran, Peter Figuli, Christian Jacobi, Nikolaos Papandreou, Haris Pozidis, Anthony Saporito, Andrew Sica, and Elpida Tzortzatos. AI accelerator on IBM Telum processor: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, page 1012–1028, 2022.

[LBN+23] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D.

Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *To be appeared in ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[LCL+15] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[LDT+16] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *ISCA*, 2016.

[LGS+20] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. Fulcrum: A simplified control and access mechanism toward flexible and practical in-situ accelerators. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[LGY+20] Ann Franchesca Laguna, Hasindu Gamaarachchi, Xunzhao Yin, Michael Niemier, Sri Parameswaran, and X. Sharon Hu. Seed-and-vote based in-memory accelerator for dna read mapping. In *ICCAD*, 2020.

[lina] Linux kernel drm-gem drivers.

[linb] Lwn.net article on gem.

[LKK+18] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *ISSCC*, 2018.

[LKY+17] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. Extrav: Boosting graph processing near storage with a coherent accelerator. *PVLDB*, 2017.

[LLKB21] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *HPCA*, 2021.

[LMC+22] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, February 2022.

[LML+19] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[LMM+19] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. A large-scale study of mpi usage in open-source hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[LMS+] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[LNM+] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[LRW+] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17.

[LT21] Yu-Chia Liu and Hung-Wei Tseng. Nds: N-dimensional storage. In *MICRO*, 2021.

[LWL+21] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, Huawei LI, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *TC*, 2021.

[LZT+21] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. Pointacc: Efficient point cloud accelerator. In *MICRO*, 2021.

[MAW+22] Rohan Mahapatra, Byung Hoon Ahn, Shu-Ting Wang, Hanyang Xu, and Hadi Esmaeilzadeh. Exploring efficient ml-based scheduler for microservices in heterogenous clusters. In *Machine Learning for Computer Architecture and Systems 2022*, 2022.

[MBA+18] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *MICRO*, 2018.

[MBS19] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *MICRO*, 2019.

[MC20] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, July 2020.

[MdKA+19] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *ACM SIGOPS 27th Symposium on Operating Systems Principles*, 2019.

[mem] Dram resource scalability enabled by cxl. `https://www.computeexpresslink.org/post/dram-resource-scalability-enabled-by-cxl`.

[Mem21] Memcached. Memcached. https://memcached.org/, 2021.

[MFJQ+16] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J. Sorin. The microarchitecture of a real-time robot motion planning accelerator. In *MICRO*, 2016.

[MGPM+22] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. Genstore: A high-performance in-storage processing system for genome sequence analysis. In *ASPLOS*, 2022.

[MHH+22] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[mica] Azure zipline.

[micb] Cxl retimer. `https://www.microchip.com/en-us/blog/2020/cxl-use-cases-driving-the-need-for-low-latency-performance-reti`.

[micc] Presidio: Data protection and anonymization sdk.

[Mit00] M. Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.

[Mit01] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[MKA+16] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R. Alam, Thomas C. Schulthess, and Torsten Hoefler. A pcie congestion-aware performance model for densely populated accelerator servers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[MKGT16] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: Specializing the datacenter. In *ISCA*, 2016.

[MKH+21] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. Smartio: Zero-overhead device sharing through pcie networking. *ACM Trans. Comput. Syst.*, 38(1–2), 2021.

[Mon21] MongoDB. https://www.mongodb.com/, 2021.

[MPA+16] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.

[MvKI21] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12), 2021.

[MWD+23] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[MYS+22] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. In *SIGMETRICS*, 2022.

[MZK+] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17.

[nap] Napi.

[NAZ+18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[NBB+21] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021.

[ner] Transformers based named entity recognition models.

[NFG+] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. page 14.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[NKA20] Ryo Nakamura, Yohei Kuga, and Kunio Akashi. How beneficial is peer-to-peer dma? In *APSys*, 2020.

[noa] CloudSuite | A Benchmark Suite for Cloud Services.

[NPB+21] Sabrina M. Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology. In *ASPLOS*, 2021.

[NRB+19] Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. Gencache: Leveraging in-cache operators for efficient sequence alignment. In *MICRO*, 2019.

[ns-] ns-3. `https://www.nsnam.org/`.

[nvi] Nvidia dali.

[nvl] NVLink. `https://www.nvidia.com/en-us/data-center/nvlink/`.

[OAVR] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18.

[ods] ODSA-BoW specifications.

[OFB+19] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[ofe] Open fabrics enterprise distribution (ofed) performance tests. `https://github.com/linux-rdma/perftest`.

[OO] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ATC'14.

[OYQ+19] Matheus Ogleari, Ye Yu, Chen Qian, Ethan Miller, and Jishen Zhao. String figure: A scalable and elastic memory network architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[PBY+] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13.

[PCC+14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, 2014.

[pcia] Pci-sig specifications. `https://pcisig.com/specifications`.

[pcib] Pcie 6.0 and 7.0. `https://www.xda-developers.com/pcie-6-to-launch-in-2024-pcie-7-in-2027/`.

[PGK+20] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *ASPLOS*, 2020.

[PGW+20] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, 40(4):93–102, 2020.

[PKB] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '17.

[PKK+22] S. J. Park, H. Kim, K.-S. Kim, J. So, J. Ahn, W.-J. Lee, D. Kim, Y.-J. Kim, J. Seok, J.-G. Lee, H.-Y. Ryu, C. Y. Lee, J. Prout, K.-C. Ryoo, S.-J. Han, M.-K. Kook, J. S. Choi, J. Gim, Y. S. Ki, S. Ryu, C. Park, D.-G. Lee, J. Cho, H. Song, and J. Y. Lee. Scaling of memory performance and capacity with cxl memory expander. In *IEEE Hot Chips 34 Symposium (HCS)*, 2022.

[PRM+17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.

[PVSW+] Johan Peltenburg, Jeroen Van Straten, Lars Wijtemans, Lars Van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *FPL*.

[QSK+20] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. *HPCA*, 2020.

[RAAGG21] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO*, 2021.

[Rad19] Milan Radulovic. *Memory bandwidth and latency in HPC: system requirements and performance impact*. PhD thesis, Polytechnic University of Catalonia, Spain, 2019.

[RAGG20] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *MICRO*, 2020.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[roc21] RocksDB. https://rocksdb.org/, 2021.

[RRR+21] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[RRRH20] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[RSAB20] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[RSC+21] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. Warehouse-scale video acceleration: Co-design and deployment in the wild. In *ASPLOS*, 2021.

[RZB+] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proc. of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15.

[SAA+17] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017.

[SCC+19] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct universal access: Making data center resources available to FPGA. In *NSDI*, 2019.

[SCV+19] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO*, 2019.

[SCW+07] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. Freepdk: An open-source variation-aware design kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 173–174, 2007.

[SFM17a] Yongming Shen, Michael Ferdman, and Peter Milder. Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer. In *FCCM*, 2017.

[SFM17b] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *ISCA*, 2017.

[Sha22] Debendra Das Sharma. Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022.

[Sha23] Debendra Das Sharma. Novel composable and scale-out architectures using compute express link. *IEEE Micro*, 2023.

[SHCZ18] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[SHPG] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12.

[SJB14] Justin Salamon, Christopher Jacoby, and Juan Pablo Bello. A dataset and taxonomy for urban sound research. In *ACM Multimedia*, 2014.

[SMLE18] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *ISCA*, 2018.

[SOA$^+$] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proc. of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15.

[SOI$^+$17] David Sidler, Muhsen Owaida, Zsolt István, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. In *FPL*, 2017.

[SPM$^+$16] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kim, Chenkai Shao, Asit Misra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, 2016.

[SQLC17] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.

[sup] Intel built-in accelerators.

[SWH$^+$16] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *DAC*, 2016.

[SYY$^+$23] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine

cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[SZQ+18] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *HPCA*, 2018.

[Tay18] Michael Bedford Taylor. Invited: Basejump stl: Systemverilog needs a standard template library for hardware design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[TBD18] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *ASPLOS*, 2018.

[TDP+19] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[TLG+15] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jing, and Steven Swanson. Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources. Technical report, UC San Diego, 2015.

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[TMS20] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *ASPLOS*, 2020.

[tof20] Intel tofino 2 p4 programmability with more bandwidth. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series/tofino-2.html`, 2020.

[TVK+20] Michael Bedford Taylor, Luis Vega, Moein Khazraee, Ikuo Magaki, Scott Davidson, and Dustin Richmond. Asic clouds: Specializing the datacenter for planet-scale applications. *Commun. ACM*, 63(7), 2020.

[TWZL21] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible probe: Timing attacks with pcie congestion side-channel. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[TZZ+16] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *ISCA*, 2016.

[uci] UCIe 1.1 specifications.

[VKO20] S. Vargaftik, I. Keslassy, and A. Orda. Lsq: Load balancing in large-scale heterogeneous systems with multiple dispatchers. *IEEE/ACM Transactions on Networking*, 28(03):1186–1198, 2020.

[VMB+22] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[VPA+] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17.

[VSDS] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16.

[WCB] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01.

[WCC20] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[Win77] Wayne Winston. Optimality of the shortest line discipline. *Journal of Applied Probability*, 14(1):181–189, 1977.

[WLP+14] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 255–268, New York, NY, USA, 2014. Association for Computing Machinery.

[WQM+23] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, April 2023.

[WW23] Shu-Ting Wang and Weitao Wang. Aurelia: Cxl fabric with tentacle. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, 2023.

[WXM+24] Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh. Data motion acceleration: Chaining cross-domain multi accelerators. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1043–1062, 2024.

[xco] Xconn cxl 2.0 switches. `https://memverge.com/wp-content/uploads/2022/10/CXL-Forum-OCP_Samsung-Xconn.pdf`.

[xila] Xilinx u30 vcu.

[xilb] Xilinx vitis data analytics library.

[xilc] Xilinx vitis data compression library.

[xild] Xilinx vitis database library.

[xile] Xilinx vitis dsp library.

[xilf] Xilinx vitis libraries.

[xilg] Xilinx vitis security library.

[Yas14] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[YDH+20] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *HPCA*, 2020.

[YZL+18] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. An efficient graph accelerator with parallel data conflict management. In *PACT*, 2018.

[ZAB+22] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *ISCA*, 2022.

[ZAC17] Sizhuo Zhang, Hari Angepat, and Derek Chiou. Hgum: Messaging framework for hardware accelerators. In *ReConFig*, 2017.

[ZBL+19] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, 2019.

[ZDZ+16] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016.

[ZEF+15] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *ACM Conference on Special Interest Group on Data Communication*, 2015.

[ZKC+] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20.

[ZLJ+21] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. Depgraph: A dependency-driven accelerator for efficient iterative graph processing. In *HPCA*, 2021.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[ZLZK] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proc. of the Internet Measurement Conference*, IMC '17.

[ZMTC18] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *ASPLOS*, 2018.

[ZWL+22] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, July 2022.

[ZZB+] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proc. of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '17.

[ZZL+21] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. Lccg: A locality-centric hardware accelerator for high throughput of concurrent graph processing. In *SC*, 2021.

[ZZW+18] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *HPCA*, 2018.