EFFICIENT INTERCONNECTION NETWORK DESIGN FOR HETEROGENEOUS ARCHITECTURES

A Dissertation

by

JIAYI HUANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Eun Jung Kim
Committee Members,	Daniel A. Jiménez
	Dilma Da Silva
	Jiang Hu
Head of Department,	Scott Schaefer

August 2020

Major Subject: Computer Engineering

ABSTRACT

The onset of big data and deep learning applications, mixed with conventional general-purpose programs, have driven computer architecture to embrace heterogeneity with specialization. With the ever-increasing interconnected chip components, future architectures are required to operate under a stricter power budget and process emerging big data applications efficiently. Interconnection network as the communication backbone thus is facing the grand challenges of limited power envelope, data movement and performance scaling. This dissertation provides interconnect solutions that are specialized to application requirements towards power-/energy-efficient and high-performance computing for heterogeneous architectures.

This dissertation examines the challenges of network-on-chip router power-gating techniques for general-purpose workloads to save static power. A voting approach is proposed as an adaptive power-gating policy that considers both local and global traffic status through router voting. In addition, low-latency routing algorithms are designed to guarantee performance in irregular power-gating networks. This holistic solution not only saves power but also avoids performance overhead.

This research also introduces emerging computation paradigms to interconnects for big data applications to mitigate the pressure of data movement. Approximate network-on-chip is proposed to achieve high-throughput communication by means of lossy compression. Then, near-data processing is combined with in-network computing to further improve performance while reducing data movement. The two schemes are general to play as plug-ins for different network topologies and routing algorithms.

To tackle the challenging computational requirements of deep learning workloads, this dissertation investigates the compelling opportunities of communication algorithm-architecture co-design to accelerate distributed deep learning. MultiTree allreduce algorithm is proposed to bond with message scheduling with network topology to achieve faster and contention-free communication. In addition, the interconnect hardware and flow control are also specialized to exploit deep learning communication characteristics and fulfill the algorithm needs, thereby effectively improving

the performance and scalability.

By considering application and algorithm characteristics, this research shows that interconnection networks can be tailored accordingly to improve the power-/energy-efficiency and performance to satisfy heterogeneous computation and communication requirements.

DEDICATION

To my loving family

To the past, the current and the future

ACKNOWLEDGMENTS

Without the support of many people, this dissertation would not have been possible. First and foremost, I would like to express my heartfelt gratitude to my advisor, Prof. Eun Jung (EJ) Kim. Since the first day, her continuous encouragement has inspired me to go through the ups and downs along this journey. Her inspiring guidance, generous support and continuous trust has motivated new directions and perspectives in interconnection networks, leading to the success of this research. In addition to her technical expertise, EJ has also offered me advice on my pursuit of career goals, I am truly indebted to her.

Many thanks to my committee members and collaborators. I am thankful to Prof. Ki Hwan Yum for his suggestions and efforts in shaping this research. I am grateful to Prof. Daniel A. Jiménez, Prof. Dilma Da Silva and Prof. Jiang Hu for the insightful feedback and the mentoring they have provided for my research and career development. Daniel has always been an enthusiastic consultant and supporter for my research and career. It has been a great honor to organize the CSE System Seminar with Dilma and Prof. Chia-Che Tsai. The collaboration with Prof. Abdullah Muzahid and Prof. Chia-Che Tsai has been an enjoyable time in the last two years. Thank you. I also thank Greg Diamos, my internship mentor, for introducing me to deep learning for systems research and teaching me the methodology for achieving one's long-term vision.

It would not have been a wonderful graduate life without friends. First, thanks to my HPC research group mates, especially Pritam and Sungkeun who have made my office life very colorful through chit-chat, debates and jokes. Thanks to Kyung Hoon for his encouragement and feedback on my research. I also appreciate the efforts that Ram, Shilpa and Rahul have contributed to this research. Thanks to Di, Mengyuan, Zelun, Jay, Jasmine, Captain, Xinxin, Wangie, and Ocean who have been supporting me since we started graduate studies together. To my bros and sis back home in China, Kow-Leng, Tsong-Tow, Low-Chu, Gem-Fay, Huk-Yen, Sa, Chuanjiang, Zetao and Guanyao, who have always been my listeners and supporters since high school and college.

I am also grateful to the teachers and mentors who have been guiding and supporting me along

my education journey. I am thankful to my junior high teacher Mr. Yanan Wang for teaching me how to take the leadership and responsibilities to lead the team to achieve the goals. Many thanks to my high school teachers Ms. Chuangqun Kuang, Ms. Michelle Xiao and Mr. Jianxin Yang. Ms. Kuang and Ms. Xiao are not only great mentors but also very close friends who have been consistently encouraging me with care and trust since high school.

Last and most importantly, my deepest appreciation to my dearest family. Papa, Mama, my twin brother Jiayue and my sister-in-law Lijuan, as well as my adorable nephew Shuoshuo, whose birth has filled the family with love and joys, thank you all for your unconditional love and support. Without you, this dissertation would not have been possible.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Eun Jung Kim (advisor), Daniel A. Jiménez and Dilma Da Silva of the Department of Computer Science and Engineering, and Professor Jiang Hu of the Department of Electrical and Computer Engineering.

In Chapter 3, the FLOV routing algorithm was derived in part by Ningyuan Wang and the FLOV+ routing algorithm was conducted in part by Shilpa Bhosekar while they were Masters students at Texas A&M University. Part of the results in Chapter 3 was published in 2017 in the proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS) [1]. The remaining results of Chapter 3 were submitted to IEEE Transactions on Computers for publication while writing the dissertation. The synthetic simulation results in Chapter 4 were in part collected by Rahul Boyapati while he was a doctoral student at Texas A&M University. The analysis in Chapter 4 was published in the proceedings of International Symposium on Computer Architecture (ISCA-44) in 2017 [2]. The simulation framework used in Chapter 5 was co-developed with Pritam Majumdar, Ramprakash Reddy Puli, and Sungkeun Kim. The results in Chapter 5 were published in 2019 in the proceedings of International Symposium on High Performance Computer Architecture (HPCA-25) [3]. The materials in Chapter 6 were under submission to a conference while writing this dissertation.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by Teaching Assistantships from the Department of Computer Science and Engineering, Texas A&M University and research grant CCF-1423433 from National Science Foundation, as well as a Dissertation Fellowship from Texas A&M University Office of Graduate and Professional Studies.

NOMENCLATURE

NoC	Network-on-Chip
VC	Virtual Channel
Flit	Flow Control Digit
CMP	Chip-Multiprocessors
ISA	Instruction Set Architecture
FLOV	Fly-Over Mechanism
NI	Network Interface
MC	Memory Controller
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
SoC	System on a Chip or System-on-Chip
APPROX-NOC	Approximate Network-on-Chip
AVCL	Approximate Value Compute Logic
VAXX	Value Approximation Mechanism
HMC	Hybrid-Memory Cube
NDP	Near-Data Processing
PIM	Processing-in-Memory
DNN	Deep Neural Network
ART	*Active-Routing* Mechanism
MULTITREE	MultiTree All-Reduce Algorithm

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

The physics limitations of semiconductor technology and grand computational requirements of emerging applications have inspired computer architecture to evolve to heterogeneous systems as shown in Figure 1.1, embracing general-purpose CPUs, specialized accelerators and disruptive computation paradigms such as approximation and near-data processing (NDP). On the one hand, such a system needs to operate under a stricter power budget due to the end of Dennard Scaling [4] and diminution of Moore's Law [5]. On the other hand, the onset of big data and deep learning applications urge future architectures to process humongous data efficiently. With the ever-increasing interconnected chip components, interconnection networks as the communication backbone is facing the enormous challenges of limited power envelope, data movement and performance scaling issues. Thus, it is demanding to design efficient interconnect solutions to tackle these multifaceted challenges for future heterogeneous architectures.

1.1 The Problem: Heterogeneous Communication Requirements

The mixture of general-purpose workloads and emerging big data applications running on heterogeneous architectures show distinct characteristics. General-purpose programs exploit data locality and deep cache hierarchy with very low average communication rate, overusing power for light traffics. In contrast, big data applications, such as graph processing and machine learning, have large memory footprint with low data reuse rate and requires tremendous communication

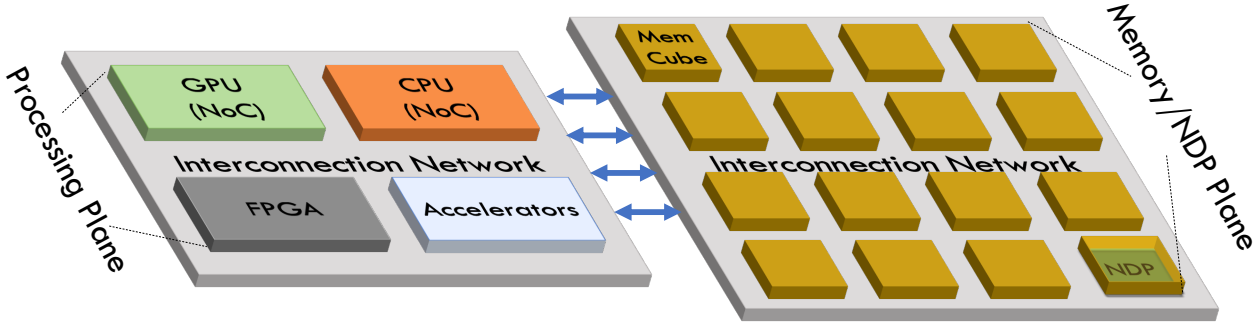


Figure 1.1: A heterogeneous system architecture example.

bandwidth to fulfill the substantial data movement demands. Furthermore, the immense advances in distributed deep learning show long-lasting communication data streams, which is very dissimilar to random short messages in conventional workloads. These diverse characteristics show heterogeneous communication requirements and reveal different inefficiency in terms of power utilization, data movement, and performance scaling on the current interconnect solutions.

1.1.1 Low-Power Interconnect in Dark Silicon

Scalable networks-on-chip (NoCs) have become the *de fecto* interconnection mechanism in large-scale chip multiprocessors (CMPs). Not only are NoCs devouring a large fraction of the on-chip power budget, but static NoC power consumption is becoming the dominant component as technology scales down. General-purpose programs running on CMPs typically have low communication traffics [6], making the standby leakage a waste of the power envelop. Therefore, it is essential to reduce static NoC power consumption for power-efficiency in dark silicon. Power-Gating as an effective static power saving technique can be used to power off inactive routers. However, packet deliveries in irregular power-gated networks suffer from detour or waiting time overhead to either route around or wake up power-gated routers. Previous research has proposed to power-gate routers attached to inactive cores in a centralized manner [7], leading to high overhead for network reconfiguration. Other research has introduced a bypass ring network to route packets even in disconnected power-gated networks [8]. However, it can introduce long latency with too many detours. In addition, they either make power-gating decision depending on the local traffic or the average global traffic, thereby either losing the big picture of the whole network or missing details of independent routers. As a consequence, they introduce performance overhead and has limited power saving. Thus, it is essential to provide a solution that captures both global and local status for power-gating decision, meanwhile achieving more power saving and low-latency packet deliveries.

1.1.2 High-Throughput Communication for Big Data Applications

The explosion of data availability and the demand for faster data analysis have led to the emergence of applications exhibiting large memory footprint and low data reuse rate. These workloads, ranging from graph processing to neural networks [9, 10], require significant data movement throughout the memory hierarchy, posing heavy stress on the communication fabrics. Furthermore, due to the gap between dense CPU computation power and deficient data supply, computer systems fail to achieve their peak computational capability. Therefore, architectural innovations are imperative to reduce data movement for substantial improvements in terms of system performance as well as energy efficiency. Prior research has adopted compression in network-on-chip to increase the throughput for communication [11, 12, 13]. However, these compression techniques are not adequate to provide the demanding throughput for big data. Other work has proposed various techniques to reduce data movement with near-data processing by moving computation near data-resident locations. In-place computation in cache uses bit-line SRAM circuit technology to transform existing cache elements to active very large vector computational units, avoiding moving data between different levels in the cache hierarchy [14, 15, 16, 17]. Processing-in-memory (PIM) has also enabled computation in the memory for data processing [18, 19, 20, 21, 22]. Although effectively reducing data movement, these approaches miss the opportunity of in-network optimization for further improvement. Thus, network level optimizations are needed to reduce data movement during communication.

1.1.3 Performance Scaling of Communication for Distributed Deep Learning

With the movement of artificial intelligence, deep learning has been transforming the landscape of computing from applications to hardware. The onset of big data era and rapid advances of accelerator architectures have enabled deep neural networks (DNNs) to achieve superhuman accuracy on complex real-world problems [10, 23, 24, 25, 26]. State-of-the-art DNN models have tens to hundreds of millions of parameters, requiring billions of compute operations and hundreds of megabytes of storage and bandwidth. Recent work projects that orders of magnitude growth

of dataset and model size are required to exceed human-level accuracy, which will take weeks to train a single epoch for language modeling [27]. As data keep exploding and DNNs evolve to be larger and deeper, it is crucial to provide scalable solutions to fulfill the trend in grand computing requirements. To this end, accelerator pod with hundreds to thousands of processors have been deployed to train these giant DNNs in a parallel and distributed manner [28, 29, 30]. During the iterative distributed training, all-reduce communication is frequently invoked for gradients synchronization, dominating the communication time and stalling the computations of the next training epoch. Widely used all-reduce algorithms either suffer from contention or long latency, resulting in resource under utilization [31, 32, 33]. Although communication algorithms are optimized in theory, they lack hardware support for coordination and communication scheduling, and thus, miss potential optimization opportunity to further improve performance. Furthermore, the fine-grained flow control and arbitration designed for general purpose network can be inefficient to support such large gradient exchanges, resulting in poor performance and huge energy/power overhead. Comparing to the significant investment in computation acceleration, little attention has been paid to communication, which can quickly become a bottleneck for large-scale distributed training.

1.2 The Solution: Communication Specialization

Along the wave of heterogeneous computing, specialization has been broadly practiced in computation acceleration tailored to various applications, such as CPU for general-purpose programs, GPU for graphics and TPU for deep learning. In contrast, specialized communication has not been widely investigated. With the divergent application domains and their heterogeneous communication requirements, the one-for-all interconnect solution is no longer efficient for their multifaceted characteristics. To keep up with the improvement of computation efficiency, communication specialization is urgently needed to tackle the challenges of power-/energy-efficiency, data movement and performance scaling accordingly. Thus, this dissertation proposes four specialized interonnect solutions for these problems as an attempt towards efficient communication in terms of power/energy and performance.

1.2.1 Low-Power Network-on-Chip for General-Purpose CMP

In general, since general-purpose programs have average light traffic loads, NoC static power consumption is a large waste of the total on-chip power budget. Static power consumption for the chip is also increasing drastically, while the feature size becomes smaller and the operating voltage gets closer to the near-threshold level [34]. Therefore, it is highly desirable to achieve power-efficient NoC designs for future CMPs. This dissertation proposes *Fly-Over* (FLOV), a voting approach for adaptive router power-gating, targeting for low-power NoCs for general-purpose CMPs. The adaptive power-gating policy works in synergy with low-latency routing algorithms in the irregular power-gating network. Such a holistic solution not only saves power but also avoids performance overhead.

1.2.2 Disruptive Computing Paradigms for Big Data Movement

Data movement has posed significant pressure on the communication backbone of modern processors due to the emergence of big data applications. Thus, it is crucial to provide communication solutions that can move data efficiently. Many of these applications, such as machine learning, image/video processing and pattern recognition have approximation in nature in their algorithm designs for either faster convergence or result estimations [35, 36, 37]. These workloads also have compute kernels that operate reduction over myriads of data. Therefore, this research exploits the error tolerance of these applications and proposes APPROX-NOC for approximate communication with lossy compression to improve the data content locality for higher compression rate, effectively improving communication throughput. In addition, by leveraging near-data processing, this dissertation proposes *Active-Routing*, an in-network computing architecture that maps compute kernels to the memory network for data-flow style processing, further reducing data movement compared to state-of-the-art processing-in-memory approach.

1.2.3 Algorithm-Architecture Co-Design for Distributed Deep Learning

Large-scale distributed deep learning training has enabled developments of more complex deep neural network models to learn from larger datasets for sophisticated tasks. In particular, dis-

tributed stochastic gradient descent intensively invokes all-reduce operation for gradient update, which dominates communication time during iterative training epochs. As computations are significantly accelerated through specialization, communication can soon become the bottleneck in distributed deep learning. To tackle this problem, this dissertation proposes MULTITREE All-Reduce algorithm and co-designs the architecture to satisfy algorithmic requirements. The algorithm takes topology and resource utilization into account for efficient and scalable all-reduce scheduling. Moreover, the interconnection network is specialized with heterogeneous bandwidth provisioning of injection/ejection and network channels in addition to flow control to cope with the algorithm in synergy.

1.3 Thesis Statement

Specializing interconnection networks tailored to heterogeneous architectures and workloads is an effective approach to achieve power-/energy-efficient, high-throughput and accelerated communication.

1.4 Contributions

This dissertation addresses the grand challenges of power-/energy-efficiency, data movement and performance scaling issues of communication in heterogeneous architectures using interconnect specialization. The main contributions of this research are the following.

- This research increases the understanding of the key factors that affect power saving and performance effects in designing router power-gating policy. The proposed routing algorithm achieves best-effort minimal route to avoid performance degradation. The proposed FLOV voting approach is well balance in the knowledge of local and global traffic status for adaptive power-gating decisions to achieve more static power saving while guaranteeing network throughput. This voting policy can be applied to other distributed power-gating schemes for performance-aware power optimizations.

- This research is the first study that introduce approximate computing to networks-on-chip for high-throughput communication provisioning. The proposed APPROX-NOC exploits

approximate data similarity in communication and translates it to high compression rate to reduce traffic loads in NoCs. Its approximate engine implements fast approximation logic that guarantees error tolerance within budget, making it suitable to work synergistically with other computation and memory approximation techniques in the same system. Furthermore, the proposed approximation module can be used in a plug-and-play fashion with any underlying data compression mechanisms.

- This research examines the first in-network computing idea in near-data processing to minimize the data movement in emerging memory networks. The proposed *Active-Routing* architecture maps compute kernels to the memory network for data-flow style processing by exploiting the pattern of aggregation over intermediate results of arithmetic operators. It seeks to schedule computations at routers attached to memory so as to take advantage of the massive bandwidth and parallelism in memory. Meanwhile, it dynamically builds topology-oblivious Active-Routing trees and leverages the network concurrency to optimize reduction operations along the routing path. This research demonstrates the promising potential for in-network computing with data-flow processing.

- This research is among the first studies that co-designing algorithm and architecture to accelerate communication for large-scale distributed deep learning. This work identifies the inefficiency in recent all-reduce algorithms and the opportunity of communication algorithm-architecture co-design. The proposed MULTITREE All-Reduce algorithm couples tree construction and communication scheduling, with topology and global link utilization awareness, to efficiently coordinate concurrent tree communications. The co-designed interconnection network is specialized to satisfy the fan-in/out of trees in the algorithm. In addition, the large size of gradients in deep learning is exploited by a simplified flow control and arbitration for better effective bandwidth utilization. This research is the first work to exploit characteristics of large-scale deep learning for communication acceleration.

1.5 Dissertation Organization

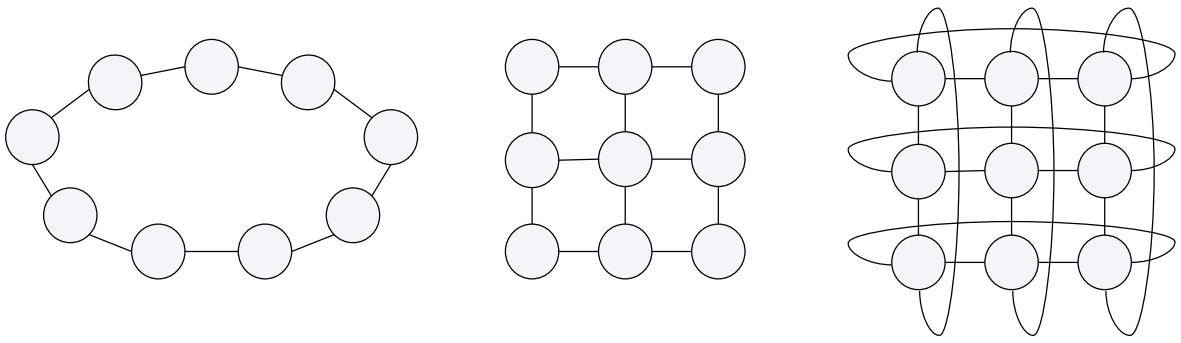
The rest of the dissertation is organized as follows. Chapter 2 introduces the fundamental concepts in interconnection network and network-on-chip (NoC) to lay down a foundation for the discussion of this research, and survey the literature of related work. Chapter 3 presents the voting-based adaptive NoC power-gating policy and its routing algorithm, designed for general-purpose CMPs. To minimize data movements for big data applications, Chapters 4 and 5 present two specialized interconnect solutions, approximate communication and in-network computing, respectively. Chapter 6 co-designs communication algorithm-architecture to accelerate communication for large-scale distributed deep learning. Finally, Chapter 7 concludes this dissertation with a discussion of the future research directions.

2. BACKGROUND AND LITERATURE SURVEY

This chapter presents the fundamental concepts of interconnection networks to facilitate the discussions of this dissertation with a literature review of the related work in the areas of low-power networks-on-chip, approximate computing, near-data processing, distributed deep learning and all-reduce collective communication, respectively.

2.1 Interconnection Network Basics

An interconnection network consists of router nodes and link channels as a communication fabric for interconnecting multi nodes in a system or multicores in in chip multiprocessors (CMPs). The construction of an interconnection network includes its network topology, routing algorithm, flow control protocol and router microarchitecture implementation. The important performance metrics for interconnection network evaluation is latency and throughput. Latency, or average packet latency defines the average time for sending a packet from one node to another while throughput defines the maximum traffic load the network can sustain for a traffic pattern before indefinite queuing or congestion happens at the end node.



(a) Ring topology (b) Mesh topology (c) Torus topology

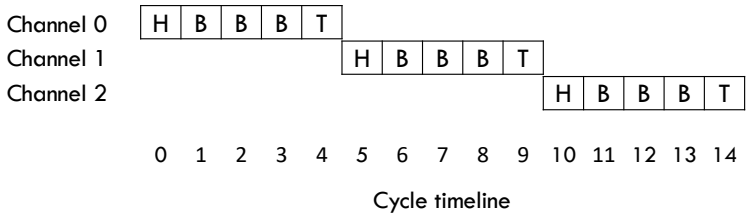
Figure 2.1: Three topology examples: ring (a), mesh (b), and torus (c).

2.1.1 Topology

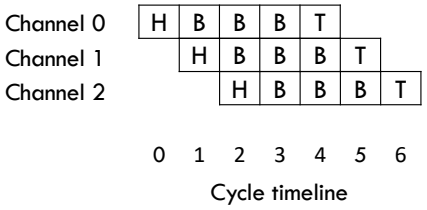
Network topology defines the connections between the router nodes and provides the possible paths between the nodes. Topology also determines how well the contention can be handled on various traffic patterns. Figure 2.1 shows three widely used topologies. The ring network shown in Figure 2.1a has minimum path diversity and hence has limited network throughput and high latency for most of the traffic patterns. Figure 2.1b shows a mesh network topology that has more choices at each node compared to ring, thereby improving performance. In Figure 2.1c, a torus network is showed to have lower network diameter and more path diversity than both ring and mesh. So it can deal with contention better to achieve lower latency and higher throughput.

2.1.2 Routing Algorithm

Given a network topology, an important factor that affects the performance is the routing algorithm that determines the packet traversal path(s) between two nodes. In general, there are two classes of routing algorithms, deterministic and adaptive routing. Deterministic routing algorithms are simpler to implement in hardware while limiting to a single choice for routing between two nodes. In contrast, adaptive routing has more path diversity for routing packets, thereby tending to have higher network throughput. One of the important issues needs to be handled in routing design is deadlock, where packets in the network are holding resources and waiting for resources held by others in a circular way indefinitely. Two classic mechanisms for deadlock handling are deadlock recovery [38] and deadlock avoidance [39]. In deadlock recovery, routing deadlocks are identified by detection techniques, then escape resources are provided to the deadlocked packet, therefore, circular dependence is mitigated to break the deadlock. A common deadlock detection technique is timeout mechanism to detect *possible* deadlock in the network. In deadlock avoidance mechanism, deadlock can be avoided by always providing choice from the a deadlock-free escape routing subfunction. So a packet can always go to the deadlock-free escape path when it is blocked.



(a) Store-and-forward example



(b) Virtual cut-through example

Figure 2.2: Flow control examples at low load without contention: store-and-forward flow control (a) and virtual cut-through flow control (b). Figures are redrawn and adapted from examples in [40], where `H` indicates head flit, `B` and `T` stand for body and tail flits, respectively.

2.1.3 Flow Control

Flow control manages how the resources, including buffers and links, are used by the packets along the route. The smallest unit that recognized by flow control methods is *flit control digit (flit)*. So, a packet usually consists of one or more flits. Since buffered routers are considered in this dissertation, we focus on buffer-based flow control. In packet-switching networks, a router needs to make sure downstream buffer availability before sending the packet. The reserved buffer lane can only be used by the reserving packet before it receives the last flit of the packet, which releases the reservation. Well-known packet-buffer flow control methods are *store-and-forward* and *virtual cut-through*. In store-and-forward, as the name suggests, the *whole* packet is received and stored before being forwarded to the next hop. And before sending packets, buffer space for the *whole* packet should be reserved in next hop. However, even when the buffers in the next hop are available, flits have been received are holding unnecessarily the buffers while waiting for the remaining flits for the same packet, as shown in Figure 2.2a. In virtual cut-through switching, received flits can start transmission as long as the buffer for the *whole* packet has been reserved in the next hop [41]. Figure 2.2b shows the timeline diagram for packet transmission using virtual cut-through flow control at low traffic load. Virtual cut-through make it possible to send flit earlier and recycles flit buffers faster, thus improving network latency and throughput. Wormhole flow control, a flit-buffer based method, relaxes the restriction on buffer reservation, where flits can be transmitted as long as there is a free flit buffer in the reserved lane [42]. Therefore, wormhole flow control can reduce the

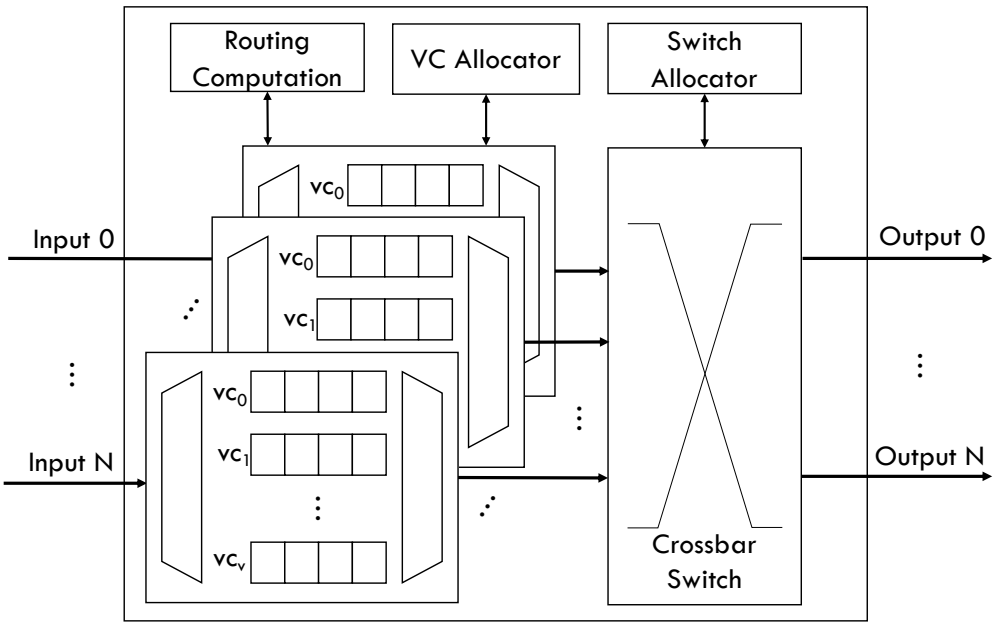


Figure 2.3: Virtual-channel router microarchitecture.

buffer requirements even with buffer size smaller than packet size. Virtual-Channel flow control, a more advanced flit-buffer based technique, associates multiple virtual channels (flit buffer lanes) with the same physical channel. Therefore, it remove the head-of-line blocking issue and increases the physical channel bandwidth utilization, thereafter improving network throughput.

2.1.4 Router Microarchitecture

Router microarchitecture is the implementation of a router switch. Figure 2.3 shows a typical virtual-channel router microarchitecture. The datapath includes the input virtual channels (VCs) and a switch crossbar switch connecting the input VC to the output ports. The incoming flits are stored in the input VC, and the routing computation unit computes the route and assigns the output port in packet basis. Then the VC allocator assigns the packet an output VC, which is associated with the downstream input VC. When the buffer resources are allocated, the switch allocator assigns the switch time in flit basis by configuring the connection of the crossbar to setup the datapath. In general, router traversals are pipelined to increase the clock frequency. Figure 2.4 shows a traditional router with five pipeline stages followed by link traversal, including buffer

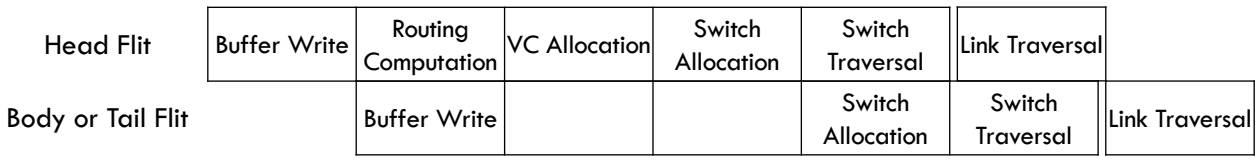


Figure 2.4: Traditional 5-stage router pipeline followed by link traversal.

write (BW), routing computation (RC), VC allocation (VC), switch allocation (VA), and switch traversal (ST). Only the head flit experiences all the stages while remaining flits skip the RC and VA stages because output port and VC have been determined in packet basis. Lookahead routing and speculation of VA and SA can be performed to reduce the executed pipeline stages [43]. If misspeculation happens, the flit falls back to the normal pipeline.

2.1.5 Network-on-Chip (NoC)

Network-on-Chip (NoCs) are on-chip interconnection networks in silicon that interconnect execution cores, slices of cache, memory and IO controllers. The design specifications of NoCs are more constrained in terms of power and area compared to its off-chip counterpart. Therefore, flit-buffer flow control is used so that buffer depth can be less than a packet size to reduce both area and power consumption. In terms of performance, link latency is the dominant component in off-chip network while router latency becomes the major portion for NoCs due to the on-chip fast wire signaling. In consequence, router latency is critical to the performance in NoCs. Therefore, speculation and pipeline are mostly applied to router datapath to reduce the latency.

2.2 NoC Power Problem and Router Power-Gating in Dark Silicon

Scalable Networks-on-chip (NoCs) such as 2D meshes, are *de facto* communication fabrics in these large CMPs. Studies show that NoCs consume a significant portion, ranging from 10% to 36%, of the total on-chip power [44, 45, 46]. Therefore, it is highly desirable to achieve power-efficient NoC designs for future CMPs. Static power consumption for the chip is also increasing drastically, while the feature size becomes smaller and the operating voltage gets closer to the near-threshold level. Previous studies show that the percentage of static power in the total NoC

power consumption increases from 17.9% at 65nm, to 35.4% at 45nm, to 47.7% at 32nm and to 74% at 22nm [47, 48]. Based on this trend, as we reach towards sub-10nm feature sizes, static power can become the major portion in NoC power consumption. Power-gating is an effective circuit technique to mitigate the worsening impact of on-chip static power consumption by cutting off supply current to idle chip components. Prior research has applied power-gating to NoCs for static power saving. In the rest of this subsection, we review the previous research related to power and performance issues of NoC power-gating.

Fine-grained Interconnect Power-Gating. Significant research has applied power-gating techniques in NoCs [49, 50]. Several fine-grained interconnect component power-gating techniques were proposed [51, 52, 53, 54]. Kim et al. introduced a dynamic link shutdown (DLS) technique together with dynamic voltage scaling to save link energy [55]. Soteriou et al. presented a power-aware network that reduces static power consumption by monitoring the link utilization and power-gating the underutilized links [51]. Matsutani et al. applied the power-gating technique to control the power supply of different components individually in an ultra fine-grained way [52]. Kim et al. proposed a buffer organization to adaptively adjust active buffer size by power-gating [53]. Parikh et al. introduced power-aware routing and topology reconfiguration to minimize detours while selected components in routers are power-gated [54]. These approaches work well to reduce the static power consumption, however, they only power-gate certain components (e.g. input buffer) of a router and requires additional circuits.

Coarse-grained NoC Router Power-Gating. Coarse-grained router power-gating has been broadly studied as well. In [56], lookahead routing is utilized to wake up sleeping routers two hops in advance to hide the wakeup latency. However, as clock frequency increases, wake up latency cannot be totally hidden. Chen et al. introduced Power Punch, a non-blocking power-gating scheme that wakes up powered-off routers along the path of a packet in advance, thereby preventing the packet from suffering router wakeup latency [57]. Zhan et al. presented a mechanism that can activate powered down cores for performance gains while considering thermal aware floor planning, and they also explored topological/routing support [58]. Catnap power-gates physical

sub-networks based on the priority and predicted traffic load [59]. Recently, Samih et al. introduced Router Parking (RP) to power-gate routers when their attached cores are sleeping while some of the routers are kept on to ensure network connectivity [60]. RP dynamically reconfigures the network among aggressive, conservative and no power-gating modes to trade-off power saving and performance. Upon reconfiguration, it estimates the power saving and decides the power-gating mode by collecting stats from all routers and distributing the recomputed routing tables. This scheme requires centralized control and typically takes a long time to reconfigure the network that may suspend new injections into the network during this phase. Chen et al. proposed node-router decoupling (NoRD) approach to leverage the independence of power-gating a core and its attached router, which provides a decoupling route through network interface to bypass the power-gated router [47]. The decoupling bypass links ensure network connectivity by using an escape bypass ring network. However, a bypass ring is not scalable to large network sizes. Another issue with NoRD is that a bypass can be constructed in a $(k \times k)$ mesh, if and only if k is even.

Bypassing Mechanisms. Some studies have suggested bypassing for different purposes in NoCs [61, 62, 63, 64, 65]. Kumar et al. proposed express virtual channels that virtually bypass intermediate routers for packet transmission to achieve high performance [61]. In [62], dual functional physical channel buffers are used to bypass a router and to keep packets in the links along the path. Long-range link [63, 64] and skip-link [65] were proposed to bypass routers for faster packet delivery. These studies has been designed purely for performance without power consideration. EZ-Pass has latches in all the directions and borrows the NoRD idea to bypass the router by going through the network interface, but it avoids the ring network [66]. EZ-Pass adds an extra routing computation unit in network interface for data bypassing, and unnecessarily going through network interface even when the packet has no need to make a turn. Furthermore, the unified VC state table increases the hardware complexity in order to support concurrent reads/writes for different ports and accessibility from both network interface and router. Muffin also incorporates similar ideas whereas it handles bypassing inside the router with extra control and arbitration [67]. In concurrent with this research, another similar bypassing mechanism is TooT, which waits for

the power-gated router to be powered on for turning purposes [68]. Later, Sponge presents a pivot router column for making turns [69]. Sponge also makes a coarser-grained power-gating decision for the whole column.

Routing Algorithms for Power-Gating NoC. In most of NoC power-gating proposals, regular network routing is used by waking up the power-gated routers [52, 57, 68]. In contrast, Router Parking recomputes routing tables at every reconfiguration and NoRD introduces a bypass ring network to solve the routing problem in irregular networks. Other research has proposed mathematical models or tools for routing in reconfigurable networks [70, 71]. However, they require more hardware complexity that increases power consumption, or their algorithm is too complex to perform in hardware to reflect rapid topology changes in the network. These tools are more suitable for application-specific multi-processor systems-on-chip (MPSoC). They can also be used to help analyze the routing design. Although fault tolerance is not the scope of this work, related routing algorithms are applicable in most cases [72]. But they are not easily extended to the cases where power-gated nodes may disconnect the network in the assumption of fault tolerance design, even the links are actually maintaining the connectivity in our setting.

2.3 Approximate Computing

In this section we discuss the related work in hardware approximation techniques and NoC data compression.

Approximation. Significant research has been done regarding approximated computation and data storage in hardware for applications that allow inaccurate outputs. Sampson et al. [73, 74, 75] have proposed code annotations and compiler framework for the programmers to define the data/-computations in the application that can be approximated. They have also proposed hardware mechanisms like voltage scaling, reducing DRAM refresh rate and SRAM supply voltage, width reduction in floating point computations for energy savings. Esmaeilzadeh et al. [76] have proposed dual voltage operation where precise computations use high voltage mode and approximate operations use the low voltage mode. Previous research has also proposed energy efficient accelerators based on neural networks and analog circuits [77, 78, 79, 80]. Liu et al. [81] has proposed to

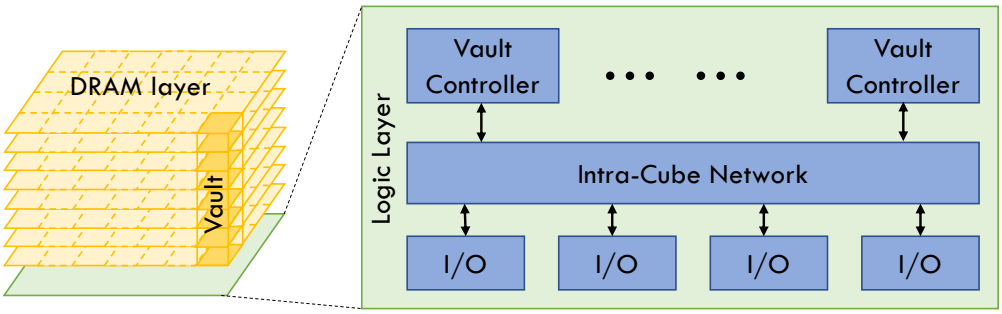


Figure 2.5: Hybrid memory cube.

reduce the refresh rate of DRAM memories which store data that can be inaccurate, using application level input. Miguel et al. [82] has designed Doppelganger, a cache mechanism that eliminates the storage of cache blocks with data that is similar (need not be an exact match). They keep the tags for all the cache blocks, but if two cache blocks are similar then only one is stored and both the tags point to this block.

NoC Data Compression. Previous research has explored data compression in NoCs. Das et al. [83] explored compression in caches and the NI of the routers while proposing techniques to amortize the decompression latency with communication latency. They observe that across a wide range of workloads data compression leads to significant network power savings and performance benefits. Zhou et al. [84] proposed a data compression mechanism in packet-based NoC architectures by tracking frequently repeated values in the on-chip data traffic. Zhan et al. [85] introduced a base-delta compression technique in NoCs to exploit the small intra-variance in data communication. Jin et al. [86] proposed a data compression mechanism that learns frequent data patterns using a table-based mechanism and adaptively turns the compression on/off based on the efficacy of compression on the network performance.

2.4 Near-Data Processing and In-Network Computing

Die-Stacked Memory. Advancements in memory technology have facilitated the integration of logic and memory dies using 3D stacking [87]. In die-stacked memory, DRAM layers are stacked on top of a logic layer. The DRAM layers are connected with the logic layer using high

bandwidth, and low-latency Through-Silicon Vias (TSV). Hybrid Memory Cube (HMC) [88] and High Bandwidth Memory [89] are two popular examples of die-stacked memory. Figure 2.5 shows the organization of HMC, which is partitioned vertically into several vaults consisting of multiple TSV connections to the logic layer. Each vault is controlled by a vault controller implemented on the logic die. The vault controllers are sparsely placed and that leave ample amount of unused silicon area to deploy more complex functional logic. It has been used to implement computation capability ranging from limited functionality [90, 91, 92] to full-fledged processors [93, 94]. HMC communicates with the processor or other memory cubes through four ports. Inside the cube's logic die, an intra-cube network is used to route the packets between the vaults and ports. HMC also enables larger memory size per package and provides abundant internal and external bandwidth with TSVs and high-speed link protocol. These advantages are leveraged in many existing processing-in-memory studies [93, 94].

Memory Network Conventional systems with DDR memory have capacity limits and bandwidth bottlenecks due to the limited number of pins per processor chip. Therefore, it requires more processor sockets in such systems to scale their memory capacity. However, the overweight data movement with respect to light computation in emerging data-centric applications can lead CPU to be under-utilized. In contrast, HMCs can be chained together to form a cost-effective memory network using packet switching and provide large memory capacity. In addition, commonly adopted processor-centric design optimizes processor-to-processor communication but overlooks the overall system bandwidth utilization. A recent study [95] has shown that memory-centric designs can achieve better bandwidth utilization as compared to processor-centric designs.

Near-Data Processing. Recently, a significant amount of research efforts to reduce data movement across the memory hierarchy to improve the system efficiency. Near-data processing (NDP), as a promising compute paradigm, has driven new architectures to move computations near data-resident locations, such as cache and memory. Aga et al. proposed compute cache [96] that uses bit-line circuit technology to perform simple computation in the cache to enable in-place computing. Processing-in-memory (PIM) [93, 97, 92, 91, 98, 99, 94, 100, 101] is an alternative NDP

design that introduces compute elements in memory for data processing. Recent studies [90, 102] have proposed to integrate PIM architectures within modern systems in a seamless fashion. They extended the instruction set to offload computations to data-resident memory modules. These mechanisms achieve better efficiency compared to conventional computing due to reduced data movements. They are most effective in the case of irregular memory accesses and atomic write operations. However, they are suboptimal when performing simple tasks over a large size of raw data, such as *dot product*, since they need to fetch part of the data across the memory network for further processing when data are not located in the same module that incurs communication and energy overhead. Ahn et al. proposed Tesseract [93], a programmable PIM accelerator for large-scale graph processing. Nair et al. [103, 97] proposed Active Memory Cube (AMC) by leveraging HMC to place vector processing units in the logic layer. AMC suffers from delays due to instruction pre-loading as well as delay and energy overhead of its complex interconnection network. Most recently Fujiki et al. [104] propose a programmable in-memory processor architecture, and data-parallel programming framework using non-volatile memory. Mondrian [94] takes an algorithm-hardware co-design approach to sequence irregular accesses for better locality. Recent study [105] analyzed Google workloads and discovered the data movement as the bottleneck for performance and energy efficiency, which is also the problem this research tries to solve.

In-Network Computing. Prior research [106, 107, 108] has advocated to provide computation power as well as routing functionalities in communication fabrics. Active Message [106] embeds the function pointer and arguments across the network to perform tasks in remote compute nodes. Pfister et al. [107] and Ma [109] proposed mechanisms to combine messages so as to reduce network traffic. Recently, IncBricks [108] implements an in-network caching middlebox for key-value acceleration in router switches. Several studies [110, 111, 112] proposed mechanisms to optimize shared value update or reduction in the network. The NYU Ultracomputer [110] introduced adders in routers to combine fetch-and-update requests for the same shared variable. Panda [111] and Chen et al. [112] proposed similar hardware to optimize reduction in the network interface for MPI collective communications. These mechanisms only support pure reduction operations and

cannot accelerate operations like *dot product*, thus requiring significant data movements across the memory hierarchy to first compute the intermediate results. Recently, Kwon et al. proposed MAERI [113] to improve efficiency for data-flow computations in deep neural network accelerators, which does not target general applications. The multiply operations require data to be brought to local SRAM and are calculated only at leaf nodes in the tree-based network topology. These in-network compute solutions have limited adaptivity since the reduction tree/ring is statically tied to the network topology.

2.5 Distributed Deep Learning and All-Reduce Collective Communication

2.5.1 Data-Parallel DNN Training

The training of a DNN model is usually done using stochastic gradient descent where each training sample goes through forward propagation, gradient calculation followed by backward propagation. Backward propagation uses the gradient to update weights of the DNN model in order to minimize loss function. To make training faster, minibatch is used where there is one pass of weight update for each minibatch of training samples. It is a daunting task to train large DNN models with a huge amount of training data. Therefore, training is often performed in a distributed environment of multiple compute nodes. Each compute node may be equipped with multiple GPUs and DNN accelerators. This creates a number of challenges regarding resource usage, communication bandwidth provisioning, and trade-off between computation and storage [114, 115]. Different models of parallelism have been utilized to make the training scalable and efficient in a distributed and parallel environment.

The most common model of parallelism in DNN training is data parallelism where a non-overlapping set of training samples are distributed to different compute nodes. Each node calculates gradients based on its own training set. Gradients are then aggregated to update weights. There are mainly two approaches to achieve this - centralized and decentralized approaches. The centralized approach relies on a parameter server where each node periodically reports its computed parameters or parameter updates to a (set of) parameter server(s) [116]. A common approach

is to use sharding of the model parameters and distribute the shards on multiple parameter servers which can be updated in parallel. However, parameter servers are not efficient in terms of bandwidth and latency for larger models. An alternative is the decentralized approach where compute nodes exchange parameter updates via an all-reduce operation. In this case, the network topology of compute nodes plays an important role. A common alternative is to employ a ring topology referred to as Ring All-Reduce [31, 117]. Ring-based approach only requires a tree topology to become bandwidth optimal. However, it is not latency optimal.

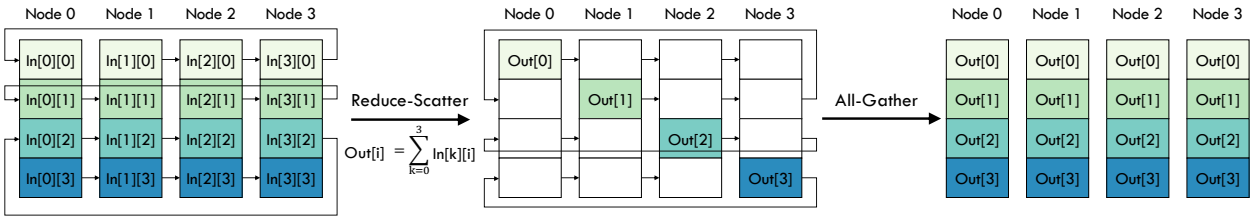


Figure 2.6: Reducde-scatter and all-gather in Ring All-Reduce.

2.5.2 All-Reduce Operation

Baidu popularized Ring All-Reduce using a sequence of reduce-scatter followed by allgather operations [117, 118]. Reduce-scatter and allgather operations are further optimized to exploit the hierarchical nature of communication bandwidths of heterogeneous network architecture [119].

Figure 2.6 shows an example. Let us assume that each row represents one segment of tensors with segment 0 being the top row and segment 3 being the bottom one. Each node forms a ring with the next node. Reduce-scatter is done on segment 0 starting from Node 1. In the first iteration, segment 0 is sent from Node 1 to Node 2 where the tensors are aggregated. Thus, two out of four sets of tensors are aggregated in the first iteration. In the second iteration, segment 0 is sent from Node 2 to Node 3 and in the third iteration, segment 0 is sent from Node 3 to Node 1. Thus, after 3 iterations, all tensors of segment 0 are aggregated to Node 0. Similarly, segment 1 starts from Node 2 and after 3 iterations, gets reduced to Node 1. Segments 2 and 3 end up getting reduced to

Nodes 2 and 3, respectively. Thus, it takes 3 iterations for reduce-scatter.

After the sequence of reduce-scatter operations, allgather operations are done in the reverse direction. In the first iteration, segment 0 is sent from Node 0 to Node 3. Now, Node 3 has two out of four segments (namely segments 0 and 3). Similarly, at the end of the first iteration, other nodes end up having 2 segments. In the second iteration, segment 0 is sent from Node 3 to Node 2 and subsequently from Node 2 to Node 1 in the third iteration. Thus, after 3 iterations, all nodes will end up having all 4 aggregated segments.

2.5.3 All-Reduce Algorithms

Several communication algorithms have been proposed to accelerate all-reduce [116, 117, 120, 33]. Widely used Ring All-Reduce is proved to be bandwidth optimal [31], which makes it suitable for larger gradient exchanges [117, 121, 122]. However, it faces link under-utilization and suffers from long latency as the number of accelerators scales up. Several attempts are proposed to improve utilization and reduce all-reduce latency by exploiting trees [123, 32, 120, 33]. Double-binary tree algorithm builds two logical binary trees in order to reduce latency for small to medium size messages [123, 32]. But it experiences congestion for giant models in larger networks due topology unconsciousness. Recent research also considers topology information with tree structures to improve all-reduce [120]. However, the linear programming complexity does not scale well to larger networks in practice. Another implementation applies partitioning optimization algorithm to build trees from leaves, which only supports a specific network topology [33]. Its backtracking operation using exhaustive search can take days to find a single solution even with a small network. Therefore, this solution is not practical and portable to various network configurations. In addition, its tree construction phase is decoupled from scheduling, leading to many conflicts during communication. Furthermore, modern interconnects designed for general purpose also lack hardware features necessary for these algorithms. More recently, Luo et al. has proposed a library for cloud to probe the physical network and schedule a two-level hierarchical aggregation plan for efficient gradient update [124]. Another recent attempt, Blink, uses approximate packing algorithms to find spanning trees for all-reduce in heterogeneous networks [125]. Recently,

Dong et al. propose a fully connected BiGraph topology for contention-free halving-doubling all-reduce [126]. However, it is nontrivial to scale out due to the fully connected complexity and not portable to other scalable topology. Additionally, the general purpose fine-grained flow control in modern interconnection networks generates many small packets for large gradients, incurring extra bandwidth and arbitration overhead. To tackle these problems, this research investigates algorithm and architecture codesign to support efficient and scalable all-reduce operation dedicated for large-scale distributed deep learning.

3. PERFORMANCE-AWARE NOC POWER-GATING [1]

Scalable Networks-on-Chip (NoCs) have become the standard interconnection mechanisms in large-scale multicore architectures. These NoCs consume a large fraction of the on-chip power budget, where the static portion is becoming dominant as technology scales down to sub-10nm node. Therefore, it is essential to reduce static power so as to achieve power-efficient computing. Power-Gating as an effective static power saving technique can be used to power off inactive routers for static power saving. However, packet deliveries in irregular power-gated networks suffer from detour or waiting time overhead to either route around or wake up power-gated routers. In this chapter, we present *Fly-Over (*FLOV*)*, a voting approach for dynamic router power-gating in a light-weight and distributed manner, which includes FLOV router microarchitecture, adaptive power-gating policy, and low-latency dynamic routing algorithms.

3.1 FLOV Router Microarchitecture

Figure 3.1 shows the FLOV router microarchitecture, which has multiplexers (muxes) and demultiplexers (demuxes) added to input/output links as well as a latch in each direction. When the power of the FLOV router is on, it works as a baseline 3-stage virtual-channel router, where muxes/demuxes are controlled to select the normal data path (path R), and the latches are power-gated. If the router is power-gated, all components of the baseline router are power-gated and the muxes/demuxes are set as 1 to activate the FLOV links. For the routers at the edge of a 2D mesh, if they are power-gated, the FLOV links are activated only in the dimension X or Y where there are neighbors in both directions. A handshake controller (HSC) block is introduced to connect all neighboring routers for handshaking purposes. Power state registers (PSRs) are added to keep track of the power states of the physically adjacent neighbors and the logical neighbors, which are the nearest powered-on routers in each direction. We modified the Credit Control Logic (CCL) to

[1] Part of the data reported in this chapter is reprinted with permission from "Fly-Over: A Light-Weight Distributed Power-Gating Mechanism for Energy-Efficient Networks-on-Chip" by Rahul Boyapati, Jiayi Huang, Ningyuan Wang, Kyung Hoon Kim, Ki Hwan Yum, and Eun Jung Kim, 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 708-717, doi: 10.1109/IPDPS.2017.77, Copyright © 2017 IEEE.

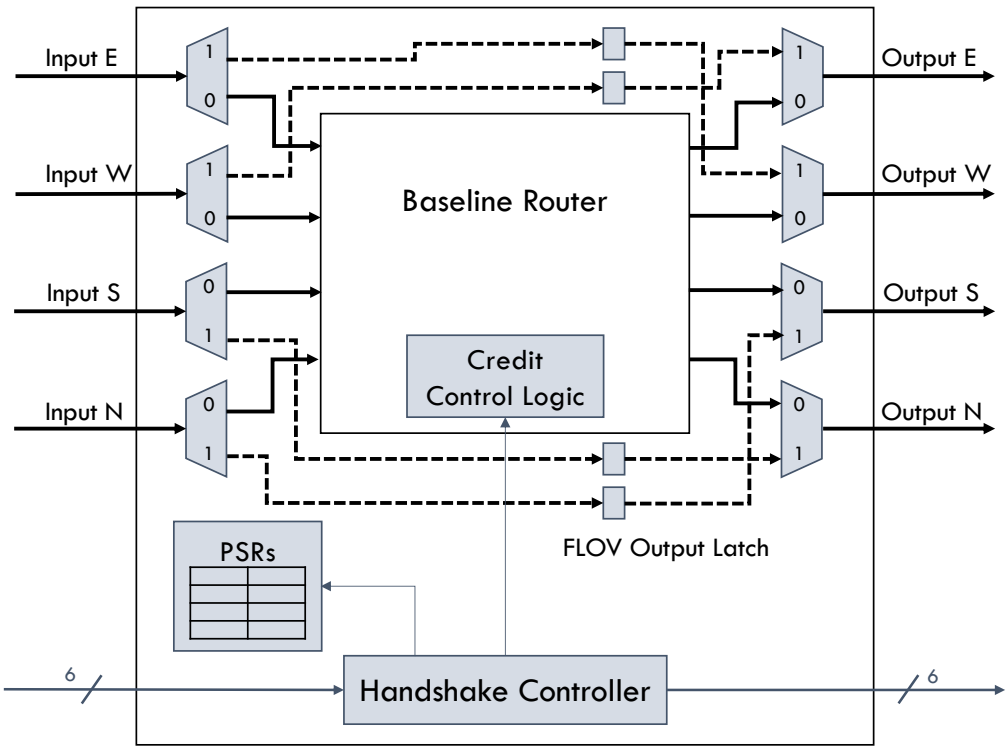


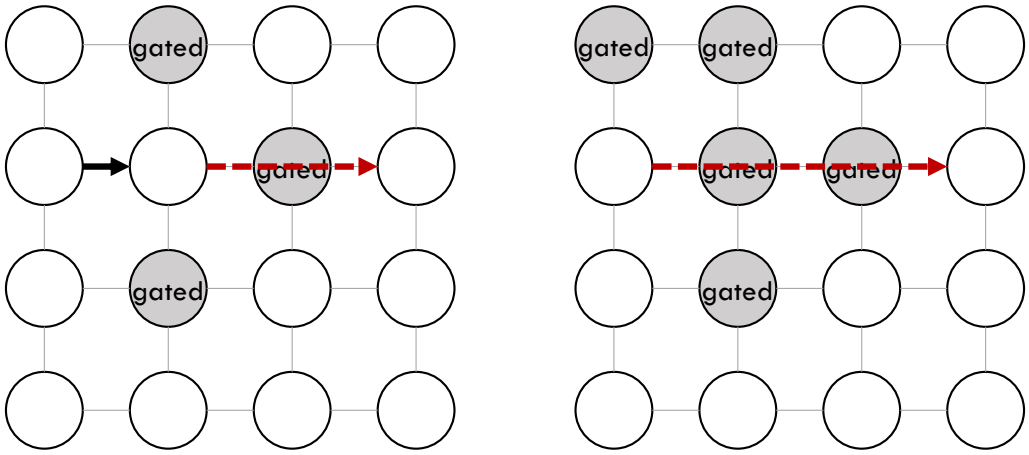
Figure 3.1: A block diagram of the FLOV router microarchitecture.

interact with HSC in order to always hold the buffer availability (credit) information of the logical neighbor routers.

3.2 Adaptive FLOV Power-Gating Policy

Using the FLOV router microarchitecture in Section 3.1, we introduce two power-gating modes, restricted FLOV (R-FLOV) and generalized FLOV (G-FLOV), in addition to the baseline no power-gating mode (NO-FLOV). R-FLOV achieves better performance with limited power saving, while G-FLOV trades throughput for better power. Figures 3.2a and 3.2b show the routing examples of R-FLOV and G-FLOV, respectively. In Figure 3.2a, two routers at the right and left edges in the second row can exchange packets passing through the smallest number of the routers instead of detouring, although there is a power-gated router on the path. This path is possible owing to the FLOV link. In Figure 3.2b, there are consecutive power-gated routers that are right next to each other. This placement is not allowed in R-FLOV but it is allowed in G-FLOV. In order to

adapt to traffic loads for performance guarantee, we propose an adaptive power-gating policy to adjust router's power-gating modes through router voting, where each router periodically decides its power-gating mode depending on the collected votes from routers in the same dimensions. The details of R-FLOV, G-FLOV and adaptive power-gating policy are described in the following subsections.



(a) R-FLOV mode (b) G-FLOV mode

Figure 3.2: Examples of R-FLOV (a) and (b) G-FLOV with power-gated routers. The gray 'gated' circle indicates a power-gated router.

3.2.1 Restricted FLOV (R-FLOV)

Figure 3.3 depicts the power state transition diagram of a router. If the core is powered-gated, the attached router sends a control signal to its neighbors using out-of-band control lines to indicate that it is in the *Draining* state. During this draining state, its neighbors cannot initiate any new packet transmissions to this router, while it is allowed to finish current packet deliveries.

In R-FLOV, a router is not allowed to power down if any of its neighboring routers is or to be power-gated. If a router in the *Draining* state receives the same signal from its neighboring router, only one of them with a smaller router ID is allowed to proceed, and the other router reverts back to normal *Active* state.

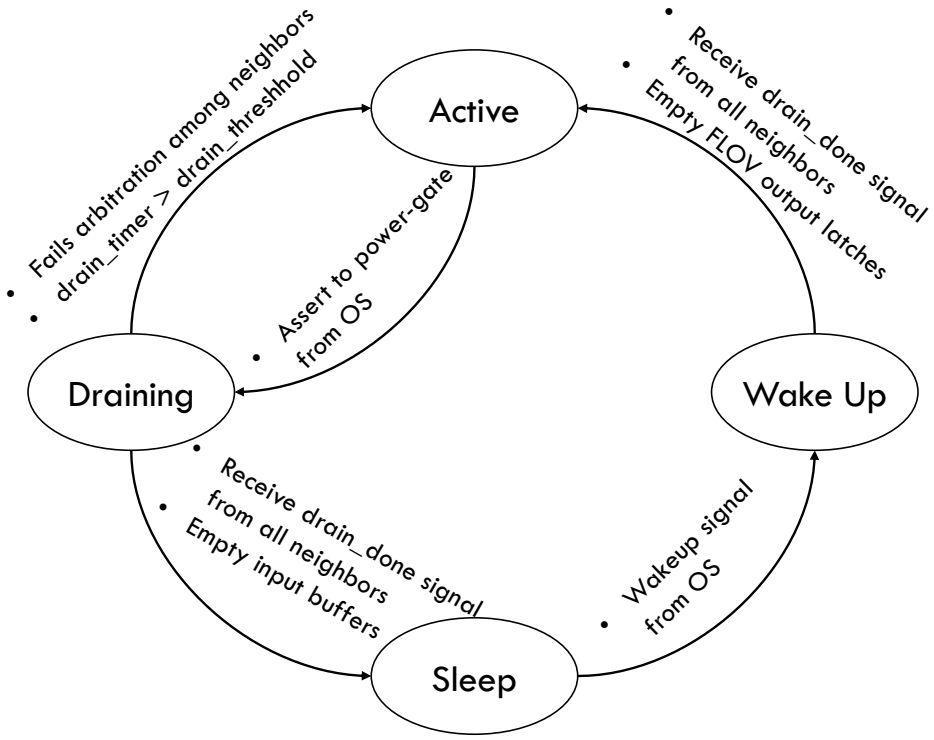


Figure 3.3: A state transition diagram for the power status of a router.

A router in *Draining* checks for any residing flits in its input buffers, and continues to forward them to downstream routers as normal. Once emptying all its input buffers and receiving `drain_done` signals from all its neighbors, the router is power-gated by shutting down the baseline router portion and enters *Sleep* state. At the same time, all the muxes/demuxes are controlled to select the forward data path (path F), and the router sends all its neighbors a signal to initiate new packet transmissions, and to update their immediate neighbor PSRs.

If a router is power-gated, a flit coming into the router is stored in the FLOV output latch without any routing/arbitration. Then, it is delivered to a designated virtual channel (VC) in the downstream router since the VC was already determined by the upstream router. From the downstream router, the packet delivery becomes normal. When a router is in the *Sleep* state, the credit counts of its downstream router are copied to the upstream router so that the upstream router can obtain the correct credit information of the downstream router.

A power-gated router in R-FLOV mode wakes up again when its core becomes active or it is

voted towards NO-FLOV mode for better performance. When a sleeping FLOV router wakes up due to the aforementioned conditions, it sends signals to its neighbors to stop new packet transmission and enters *Wakeup* state. When it completes current packet transmissions and emptying its output latches, the router powers on the baseline router portion and switches to select the normal data path R. During *Wakeup* process, the FLOV router may still relay credit signals from its downstream router to its upstream router. Once it becomes *Active*, it only processes credit information from downstream routers for its own, and its upstream router sets the corresponding credit to fully available.

Figure 3.4 shows a set of snapshots of a working example to demonstrate R-FLOV mode in time sequence. For simplicity, draining of the packets and credit control are shown only for one direction, but a router has to perform these actions for all its neighbors before state transitions.

(a) In Figure 3.4a, three routers are *Active*. Router A holds the body (`B1`) and tail (`T1`) flits of packet 1 as well as the head flit (`H2`) of packet 2. Router B holds the head flit (`H1`) of packet 1 and Router C is empty. The PSR entries of the routers show the power states of the immediate neighbors in the East (Routers A and B) or West (Router C). The current credit status of VC1 of the downstream routers is also shown. The shaded portion indicates the power-gated components that are the output latches.

(b) In Figure 3.4b, both Routers B and C send `Drain` signals to their neighbors to indicate their willingness to go into the *Draining* state. Since Router B has the lower router ID, it wins the arbitration and Router C has to revert to *Active* state. The PSR entries in Routers A and C are updated to *Drain* due to Router B. Router A transfers flit `B1` to Router B and B transfers flit `H1` to Router C. The corresponding credit counters are also updated.

(c) In Figure 3.4c, Router A sends the `drain_done` signal to Router B as it finishes transmitting packet 1 to B. Similarly, Router C sends the `drain_done` signal to B. But since Router B has not finished draining its buffers yet, it has to wait before going into the *Sleep* state.

(d) Figure 3.4d depicts the scenario after Router B finishes draining packet 1 to Router C and

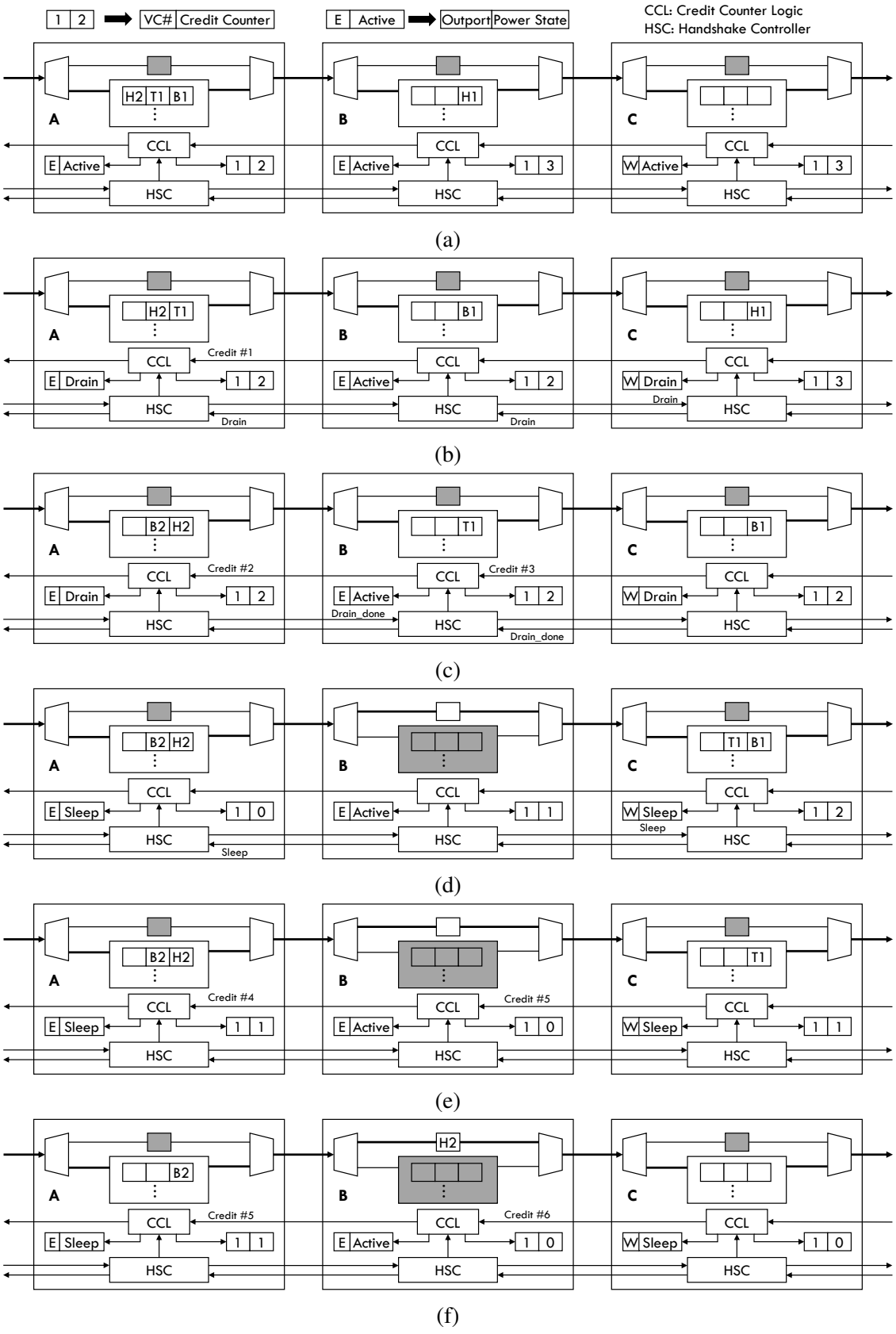


Figure 3.4: An example of R-FLOV with snapshots in timeline from (a) to (f).

goes into *Sleep* state. The shaded VC buffer indicates that the baseline router has been power-gated and the FLOV links (output latches) have been activated. Router B sends the `Sleep` signal to its neighbors so that they can update their corresponding PSR entries, and the credit counters are initialized as shown in Router A. Note that although Router A has a flit (`H2`) to send Router B, it still has to wait until B finishes its power state transition.

(e) Figure 3.4e shows the credit control and maintenance between Routers A and C while Router B is power-gated. After Router B goes into the *Sleep* state, Router A initialize its credit counter entry and the credit information is copied from Router B to A (`Credit #4`). This is because Router C is the logical neighbor of Router A, so A has to keep track of the buffer availability (credits) in C. `Credit #5` carries the newly available credit in Router C to Router B.

(f) In Figure 3.4f, `Credit #5` is relayed by the power-gated Router B to Router A. Then, it updates its credit counters. This relaying scheme maintains the correct flow control between Router A and Router C.

The wake-up procedure is similar to the draining procedure, the *Wakeup* router sends the `wake-up` signals to its neighbors and starts to drain packets from its output latches. The router also waits for all its neighbors to finish any intermittent transmissions and sends `drain_done` signals. The router then receives the credit information from the downstream router and sends a signal to notify the upstream router to make its corresponding credit counter fully available. Once it happens, the router controls muxes/demuxes to resume baseline operations.

3.2.2 Generalized FLOV (G-FLOV)

R-FLOV tends to achieve better throughput but limits power saving because none of a sleeping router's neighbors is allowed to sleep regardless of the power states of their attached cores. In this section, we introduce *generalized* FLOV *(*G-FLOV*)*, where a router can be power-gated even if any of its neighbors is power-gated, thereby two or more consecutive routers in a row/column can be power-gated simultaneously. During handshaking, the power-gated routers in the middle should

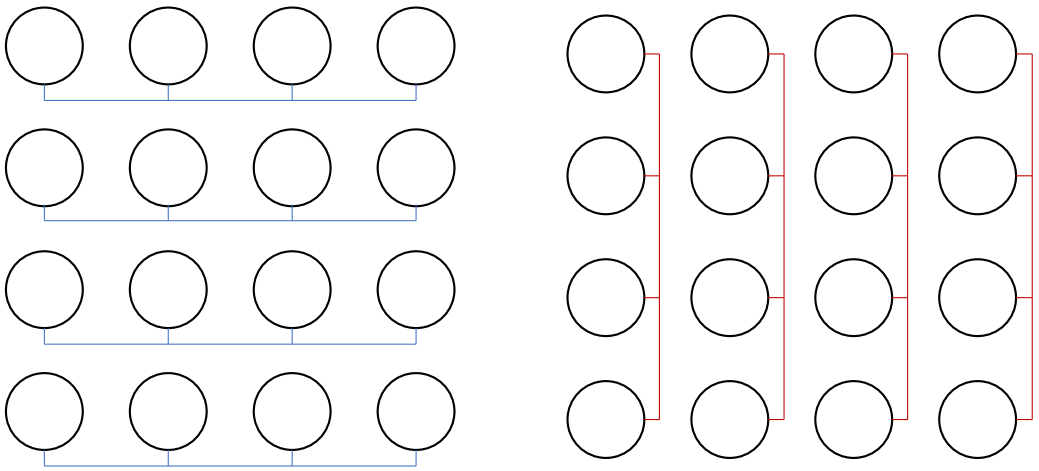
relay the handshake signals, in addition to update the corresponding logical and physical neighbor routers' power states in the PSRs. The flow control in G-FLOV is similar to R-FLOV, except that credit relaying may across several sleeping routers.

When a router is in R-FLOV mode, its neighbors are all *Active* if it is power-gated. In contrast, a power-gated router in G-FLOV mode may have neighbors in a chain that are also power-gated. Therefore, we introduce a few handshaking protocol modifications and additional functionalities to avoid protocol deadlock and to aid routing decision, described as follows:

Firstly, after a router enters *Sleep* state, it sends the corresponding power state and router ID of its logical neighbors in each direction to its upstream router, in addition to its new power state. Then, the logical neighbor of the power-gated router becomes the logical downstream router for its upstream router. Thus, the logical PSRs of all the routers can be kept up-to-date. Moreover, the logical neighbor ID information helps design a better routing algorithm that is presented in Section 3.3.

Secondly, in wormhole switching, no two logical neighbor routers in the same row/column are allowed to stay in *Draining-Draining*, *Draining-Wakeup*, or *Wakeup-Wakeup* state combinations at the same time in order to avoid protocol deadlock or starvation. If one of the handshaking routers is trying to wake up and the other trying to drain, *Draining* has lower priority due to the fact that *Wakeup* is more crucial for performance. For the simplicity of handshaking, if a power-gated router has a downstream router in the *Draining* state, it cannot wake up until the draining router changes its state. When two handshaking routers are trying to drain or wake up at the same time, only the one with a smaller router ID can proceed. If virtual cut-through switching is applied, the above condition can be relaxed for the *Wakeup-Wakeup* case. Unlike the *Draining-Draining* combination, two waking up routers have no dependence on each other since they always bypass the flits. In addition, the buffer resource in a powered-on router between two *Wakeup* routers is sufficient to store a whole packet to finish intermittent transmissions for cut-through. In addition, *Wakeup* routers that are involving handshaking should relay the `drain_done` signal to the neighbor *Wakeup* router on the same direction if there is any.

3.2.3 Adaptive FLOV through Router Voting



(a) Row voting buses (b) Column voting buses

Figure 3.5: Two-bit voting buses for rows (a) and columns (b) of adaptive FLOV power-gating policy.

Router power-gating is attractive in low network load since packets can be delivered with low latency. However, in medium to high loads, the network may become congested that can incur high latency overhead and sacrifice throughput. Therefore, it is important to dynamically adapt the trade-off of power saving and performance. Therefore, we propose *adaptive* FLOV (FLOV) policy to dynamically change each router's FLOV power-gating mode among NO-FLOV, R-FLOV, and G-FLOV progressively through router voting. In a power-gating network, large packet latency is mostly due to either detours from the shortest path or long credit round-trip latency. Therefore, we adopt a voting approach that each router periodically votes for its row and column for more power saving or better performance. Then each router collects the votes of its row and column to decide its FLOV policy independently. We introduce a high and a low latency watermark *hw* and *lw* to compare with the average latency of received packets. If the average latency is lower than the low watermark, the router votes for more aggressive power-gating mode. If the average latency is higher than the high watermark, the router votes for more conservative power-gating

mode towards performance consideration. Otherwise, it votes for no change. Given an empirical zero-load latency $latency_{zero-load}$, we define the watermarks as:

$$lw = 1.2 \times latency_{zero-load} \tag{3.1}$$

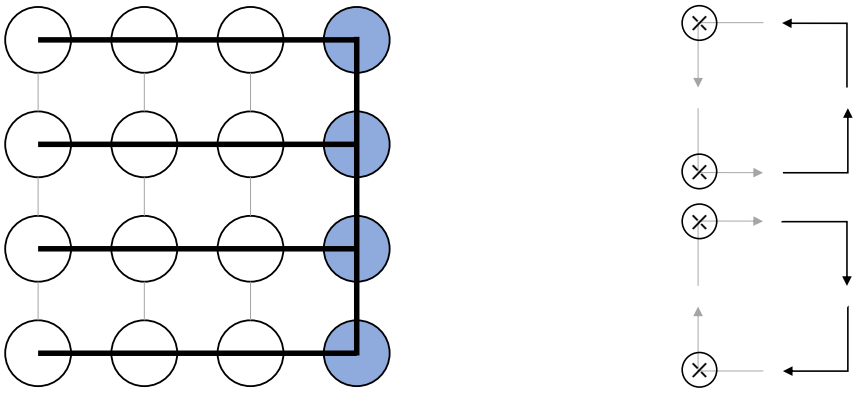
$$hw = 1.5 \times latency_{zero-load} \tag{3.2}$$

Figure 3.5 shows two-bit voting buses for rows and columns in the mesh network. These buses are time multiplexed for voting by each router in the corresponding row and column. Upon the voting time of a router, it compares the average latency of the packets received since last voting time to the watermarks. It votes -1 if the average latency is higher than *hw* and votes 1 if the average latency is lower than *lw*. Otherwise, it votes 0. All the routers snoop the buses and collect the votes. If the accumulative vote is greater than zero, the router change itself to a more aggressive power-gating mode, from NO-FLOV to R-FLOV or from R-FLOV to G-FLOV; if the accumulative vote is less than zero, it regresses to a more conservative power-gating mode, from G-FLOV to R-FLOV or from R-FLOV to NO-FLOV. Otherwise, it remains unchanged. As a result, FLOV policy decides a router's power-gating mode jointly by the routers in the same row and column through voting.

3.3 Dynamic Routing Algorithms

The FLOV NoC baseline architecture is a two dimensional mesh topology with one column or row of routers (on the edge) powered on all the time. In this paper, we assume the routers at last column are *always on* (*AON* routers) so as to ensure the network connectivity across the topology with the facility of FLOV links, forming an escape sub-network as shown in Figure 3.6a. One VC of each powered-on router is reserved for deadlock-free routing subfunction, called an escape VC. The routing algorithms include routing for packets in the regular VCs and routing for packets in the escape sub-network. Previously, we proposed a FLOV routing algorithm that adopts a deadlock recovery mechanism, where a suspected deadlocked packet in a regular VC is sent to an escape VC to recover from deadlock [127]. In this paper, we propose a new algorithm, FLOV+, by introducing more adaptation to achieve better throughput. Instead of deadlock recovery, FLOV+ adopts

Duato's Protocol to avoid deadlocks [39]. Note that routing computation is performed in powered-on routers, while power-gated routers only forward packets without changing the direction. In the following sub-sections, we describe the details of the FLOV and FLOV+ routing algorithms.



(a) Escape sub-network (worst case) (b) Turn model

Figure 3.6: Worst case of escape sub-network is shown in (a), which uses always-on routers at the last column; and turn model is shown in (b).

3.3.1 FLOV Routing Algorithm

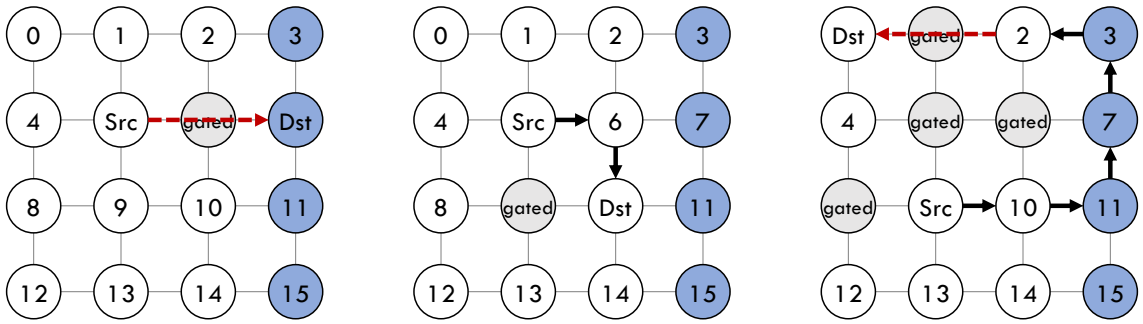
Algorithm 1 describes the FLOV routing algorithm, which is based on dynamic YX/XY routing, in addition to the consideration of power states of neighbor routers and deadlock-free escape routing subfunction.

For packets whose destinations are in the same dimension (X or Y) as the current router, the router sends them directly to the directions towards their destinations. Even in the case of power-gated downstream routers, the FLOV links ensure their delivery to the destinations. For packets whose destinations are at different dimensions (X and Y) from the current router, the paths incur turns towards their destination. If the Y neighboring router on the minimal route is powered-on, the packet is sent to the downstream router using YX routing. If it is power-gated, the power state of the X neighboring router on minimal route is checked, and if it is powered-on, the packet is forwarded to it.

When both the neighboring routers on the minimal path are power-gated, a viable path to the destination cannot be guaranteed since the current router may not be aware of the power states of the downstream routers in the further path. In this case, the packet is forwarded to the East neighbor towards *AON* routers and confined to the escape path and escape VCs. If the East router happens to be power-gated, the FLOV link is used for bypassing. The packet is not sent to the router in the Y direction because, in the worst scenario, if all the downstream routers in the Y direction are powered-gated, the packet is not able to make a turn and hence cannot be routed to the destination. However, if the packet is directed to the East direction, we can guarantee that the packet is able to make a turn toward the destination using the *AON* router of the corresponding row. Note that no u-turn is allowed so as to avoid livelock situations, where a packet keeps bouncing between two neighbors.

Since the algorithm has both YX and XY decisions, it is not necessarily deadlock-free. We adopt a timeout mechanism for suspected deadlock recovery [38]. If a packet has been waiting in a buffer for an extended time, it may exceed a certain threshold and be directed to the escape VC in the downstream routers to reach the destination using the deadlock-free escape sub-network. In the escape subfunction, as described in line 16–21 in Algorithm 1, a packet with destination in the same dimension (X or Y) of the current router is directly sent to their destinations and use FLOV links if needed. If the above condition is not satisfied, it is forwarded to East to the *AON* router and make a turn towards another *AON* router that is located at the row as the destination. Then the packet is sent to the West to its destination. Once a packet enters the escape path, it is confined to escape routing and escape VC. Based on the turn model [128], the escape subfunction is deadlock-free since it only allows four turns as shown in Figure 3.6b.

Figure 3.7 shows three FLOV routing examples. In Figure 3.7a, the destination is in the same dimension as the source router. Even though the router in between is power-gated, the packet can be forwarded to the East using FLOV link. In Figure 3.7b, the destination is in different dimensions from the source router. First, the routing algorithm checks the status of Router 9. Since Router 9 is power-gated, the packet can be sent to Router 6 that is powered-on. Then the packet makes a



(a) Example 1 (b) Example 2 (c) Example 3

Figure 3.7: FLOV routing algorithm examples 1 (a), 2 (b), and 3 (c). The gray 'gated' circles indicate power-gated routers. 'Src' and 'Dst' represent source and destination.

turn and reaches its destination. In Figure 3.7c, both neighbor Routers 5 and 8 on the minimal path are power-gated, therefore, the packet is sent to Router 10 and confined to escape sub-network so that it can at least make a turn at the *AON* column. Router 10 computes the escape route to East towards Router 11 since the destination is not in the same dimension. Router 11 then routers the packet to Router 3 where it makes another turn toward the destination.

3.3.2 FLOV+ Routing Algorithm

The FLOV routing algorithm works well when only low to medium fraction of routers are power-gated. When the power-gated routers continue to increase, more packets are directed to the escape sub-network which can cause low regular VC utilization and high congestion in the escape sub-network, especially in the *AON* column. It also incurs detours and is not able to route packets through the shortest path in some cases. In addition, only one routing option is available for selection, which lacks adaptation and may block the packet for a long time when the only output port and VC set are busy.

Figures 3.8a and 3.8b show the sub-optimal and optimal routing examples, respectively. In Figure 3.8a , a packet is sent from Router 9 to Router 0 using the FLOV routing algorithm. Since both physical neighbors of the source router are power-gated, the packet is directed to East to use the escape network towards the *AON* column and makes turns to reach its destination, resulting in 7 hops in total. Note that there exists a shortest path from Router 9 to Router 0 by going through

Algorithm 1: FLOV routing algorithm.

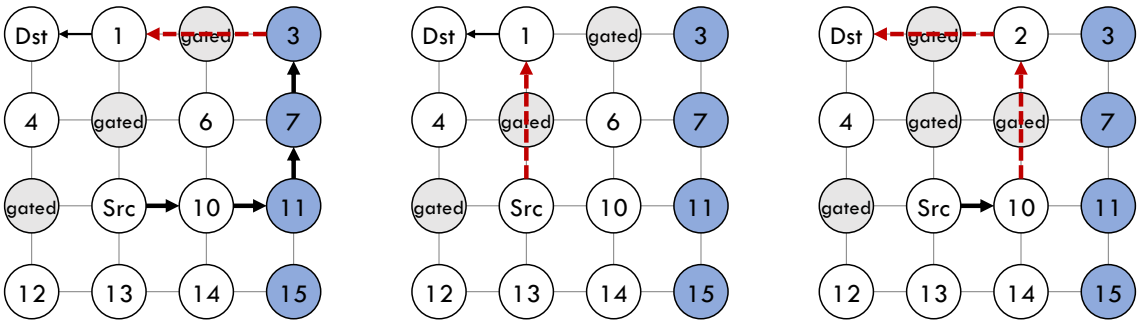
Input: $cur, dest, in\_port, in\_vc, inqueue\_time$
Output: out_port, vc_set

```
bool escape = false;
(xy_port, yx_port) = GetMinimalPorts(cur, dest);
if IsEscape(in_vc) || inqueue_time > threshold then
    escape = true;
else
    // Use YX or XY if either neighbor
    // router is not power-gated
    if neighbor[yx_port] is power_gated then
        if neighbor[xy_port] is power_gated then
            escape = true;
        else
            out_port = xy_port;
    else
        out_port = yx_port;
    if IsUTurn(in_port, out_port) then
        escape = true;
// Escape subfunction
if escape == true then
    if yx_port == xy_port || at AON column then
        out_port = yx_port;
    else
        out_port = East;
    vc_set = escape_vc;
else
    vc_set = regular_vcs;
return (out_port, vc_set);
```

Algorithm 2: FLOV+ routing algorithm.

Input: $cur, dest, in\_port, in\_vc$
Output: $routes$

bool $escape$ = false;
if *IsEscape*(in_vc) **then**
    $escape$ = true;
(yx_port, xy_port) = GetMinimalPorts$(cur, dest)$;
$(yx\_credits, xy\_credits)$ = GetFreeCredits$(cur, dest)$;
// Add escape option with lowest priority
if $yx\_port == xy\_port$ || *at AON column* **then**
    $escape\_port = yx\_port$;
else
    $escape\_port = East$;
$routes$.Add$(escape_port, escape_vc, lowest)$;
// Add regular routing options
if $escape ==$ *false* **then**
    // Prioritize routing options
    if $yx\_credits >= xy\_credits$ **then**
        $yx\_pri = highest$;
        $xy\_pri = high$;
    else
        $yx\_pri = high$;
        $xy\_pri = highest$;
    // Determine route availability
    if $logical\_neighbor[yx\_port]$ *on minimal path && NotUTurn*(in_port, yx_port) **then**
        $route\_yx$ = true;
    if $logical\_neighbor[xy\_port]$ *on minimal path && NotUTurn*(in_port, xy_port) **then**
        $route\_xy$ = true;
    // Add routing options
    if $route\_yx ==$ *true* **then**
        $routes$.Add$(yx_port, regular_vcs, yx_pri)$;
    if $route\_xy ==$ *true* **then**
        $routes$.Add$(xy_port, regular_vcs, xy_pri)$;
    if $route\_yx ==$ *false* && $route\_xy ==$ *false && NotUTurn*$(in\_port, escape\_port)$ **then**
        $routes$.Add$(escape_port, regular_vcs, low)$;
return $routes$;



(a) Example 1 (b) Example 2 (c) Example 3

Figure 3.8: Routing algorithm examples: example 4 (a) uses FLOV Routing while example 5 (b) and 6 (c) use FLOV+.

power-gated Router 5 to reach Router 1 to turn to the destination, traveling only 3 hops as shown in Figure 3.8b. This path is not considered by FLOV routing because it ignores the relative position between downstream *Active* router and destination.

To tackle the aforementioned problem, we have improved the routing algorithm by leveraging the information of destination's position relative to downstream router's position. Note that during handshaking, the router switching to *Sleep* state sends its corresponding logical downstream neighbor's power state and ID in each direction to its upstream router. Therefore, the downstream routers' relative positions to the destination can be calculated to make better routing decisions. Furthermore, more options are provided for routing selection to exploit the path diversity and adaptation. In this new algorithm, instead of deadlock recovery, we use deadlock avoidance by applying Duato's Protocol where the escape route option is always provided for selection with lowest priority [39].

The new routing algorithm is described in Algorithm 2, called FLOV+ routing algorithm. With these optimizations, we can relax the burden of the escape network, especially the *AON* column. For the FLOV escape routing subfunction, a packet is forwarded to the *AON* column to make turns as shown in Figure 3.7c, which can lead to congestion. In contrast, when using FLOV+ routing, as shown in Figure 3.8c, the routing option in line 27 of Algorithm 2 makes a detour by sending the packet from Router 9 to 10. Thus, the packet can make a turn at Router 10, then flies over

Router 6 to reach Router 2, and finally turns to West towards the destination. This routing path mitigates the pressure of *AON* column and is the minimal path in the irregular power-gated network. Since a packet is always sending closer to the destination row and u-turn packets are directed to escape routing afterwards, the algorithm is also livelock-free. The adaptive routing options also provide higher throughput than FLOV in power-gated networks, translating to more power-gating opportunities, thereby more power savings.

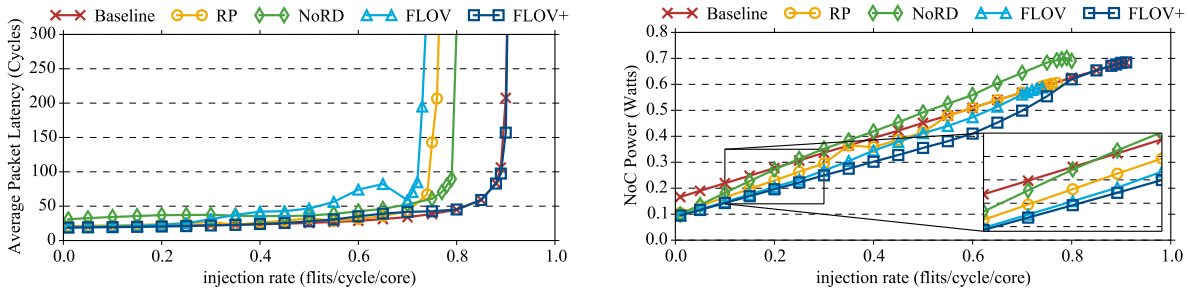
3.4 Experimental Evaluation

Table 3.1: FLOV System Configuration

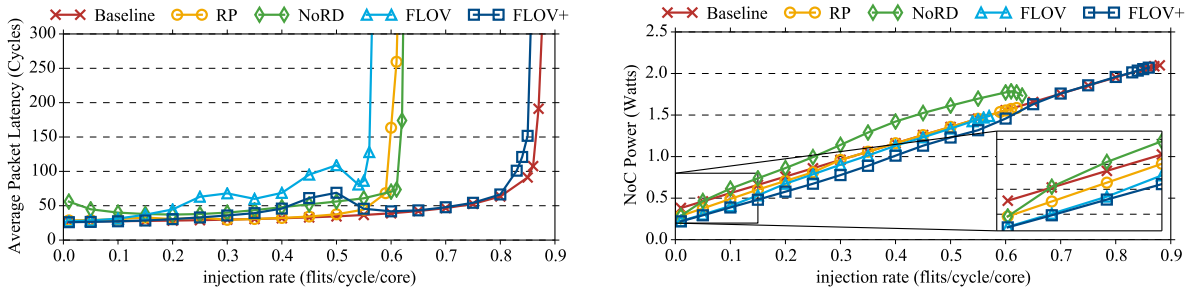
Parameter	Configuration
Network Topology	4×4, 6×6, 8×8 (default) and 10×10 Mesh
Input Buffer Depth	5 flits
Router	3-stage (3 cycles) router
Virtual Channel	3 regular VCs and 1 escape VC per virtual network 1 vnet for synthetic and 3 vnets for full system
Packet Size	5 flits/packet for synthetic workload 1-flit control and 5-filt data packet for full system
Memory Hierarchy	32 KB L1 I/D Cache, 8 MB L2 Cache MESI, 4 MCs at 4 corners
Technology	32 nm
Clock Frequency	2 GHz
Link	1 mm, 1 cycle, 16 B width
Power-Gating Parameters	Power-Gating overhead = 17.7 pJ wakeup latency = 10 cycles
Baseline Routing	Minimal Adaptive Routing

3.4.1 Experimental Methodology

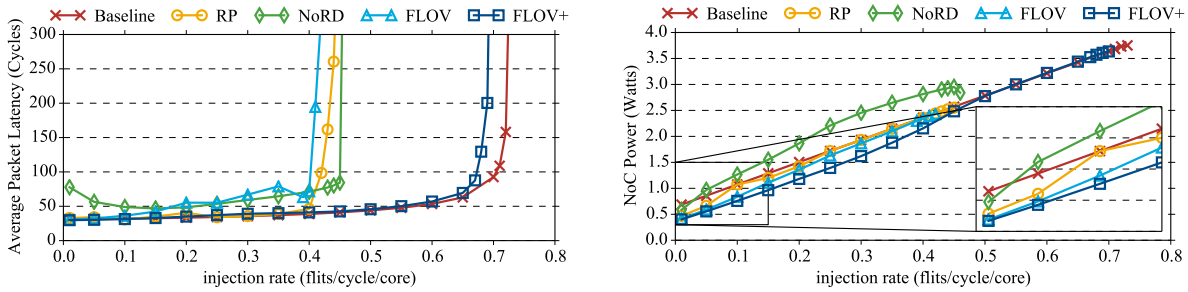
We use BookSim [129] for synthetic workload experiments, and integrate it with `gem5` [130] for full system simulations. In addition, we use DSENT [48] to estimate power consumption of



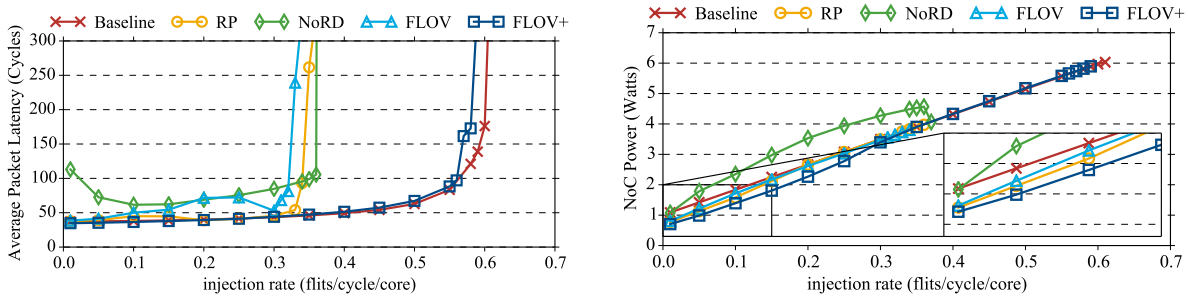
(a) 4×4 mesh



(b) 6×6 mesh



(c) 8×8 mesh



(d) 10×10 mesh

Figure 3.9: Load-latency curves (left) and power consumption (right) under uniform random traffic with 50% cores power-gated for 4×4 (a), 6×6 (b), 8×8 (c), and 10×10 (d) mesh networks.

the interconnect components with 0.5 switching activity in 32-nm technology. We assume a 2-GHz clock frequency for the routers and links. Table 3.1 summarizes the simulation configuration parameters. Both synthetic and real workloads are evaluated for performance and power-saving comparisons of FLOV and FLOV+ against the baseline interconnects with no power-gating (Baseline), Router Parking (RP) and Node-Router Decoupling (NoRD). We compare both the original FLOV routing algorithm [127] and the proposed FLOV+ routing algorithm in adaptive power-gating setting. Otherwise stated, we assume 50% of the cores are power-gated that have no injections, where the power-gated cores are randomly decided. Uniform Random and Tornado traffic patterns (among powered-on cores) for synthetic workloads as well as eight benchmarks from PARSEC benchmark suite [131] are used for evaluation. For synthetic workloads, simulations are warmed up with the first 10,000 cycles and run for 100,000 cycles in total. For PARSEC benchmarks, the ROI is evaluated with small input data size.

3.4.2 Throughput and Power Consumption

Figure 3.9 shows the load-latency curves and power consumption under Uniform Random traffic with 50% cores power-gated, where Figures 3.9a, 3.9b, 3.9c and 3.9d show for 4×4, 6×6, 8×8 and 10×10 mesh networks, respectively. Since static power is dominant in the evaluated technology, RP may stick to aggressive mode most of the time during medium to high traffic load and lead to high performance overhead. Therefore, we run aggressive, conservative and no power-gating modes, and switch to a more conservative mode towards performance when the average latency increases more than 25%, unless it is already in no power-gating mode.

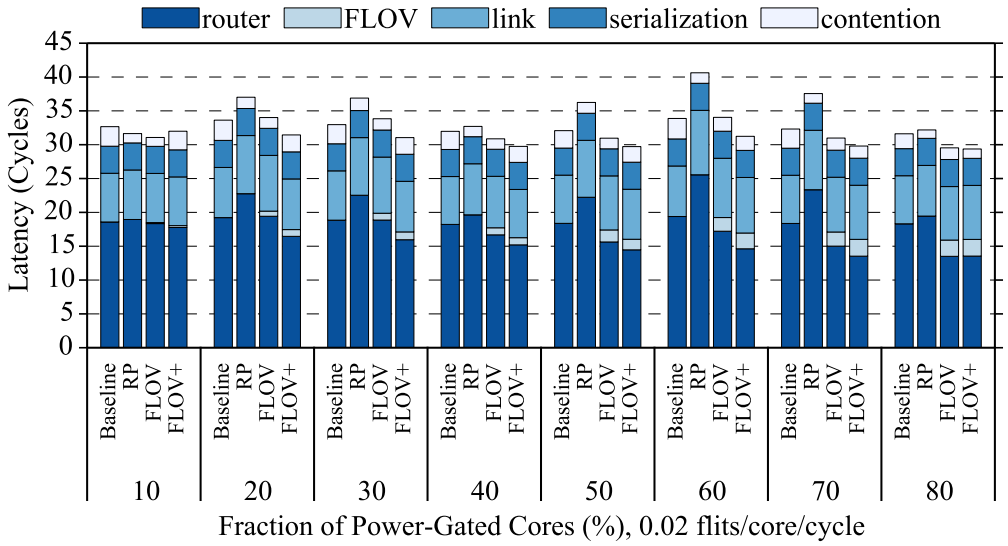
As shown in Figure 3.9, NoRD tends to have higher latency compared to other techniques in low traffic load. While in low to medium load, both FLOV and NoRD have higher latency than the others. This is due to the limitation of FLOV routing that directs a packet to escape whenever its neighbor routers on the shortest path are power-gated, making the escape network congested. NoRD may not wake up the power-gated routers early enough and suffers latency penalty, since power-gated routers are not woken up by its attached off core but by the packets that go through the ring network. In terms of throughput, the original FLOV saturates earliest due to lack of adaptation

and heavy reliance on the escape network. RP is better than FLOV and worse than NoRD since RP's deterministic routing is not as flexible as NoRD's adaptive routing. And FLOV+ outperforms these three and is the closest to Baseline. Since NoRD uses two VCs for escape and FLOV+ only has one escape VC, FLOV+ have more adaptive VC resources for better throughput. In addition, the escape routing subfunction of FLOV+ also reduces the hop counts compared to NoRD. While comparing to Baseline, FLOV+ escape subfunction is worse than XY deterministic the escape subfunction of Baseline, leading to an offset in throughput. All three network scales have similar trend in latency and throughput, showing FLOV+ scales well to different network sizes, improving throughput over state-of-the-art by 40% – 70%.

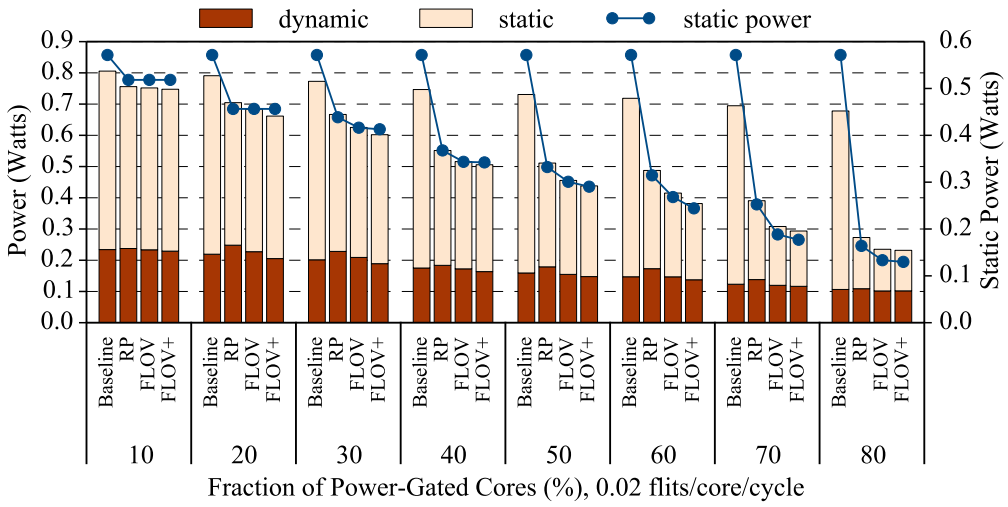
Figure 3.9 also shows the power consumption of the corresponding injection rates. The low load region is zoomed in for analysis, as shown in the lower right corner in the bottom flov. When traffic load is between 0 to 0.05 flits/cycle/core (except for 10×10 network), NoRD saves static power since the network is mostly idle and it can power-gate more routers. However, as traffic load increases, NoRD starts to consume a bit more power than Baseline, mainly due to the long detour of the escape ring network. This is similar to the discovery in the literature [47]. This overhead tends to be higher for larger network size. Other techniques save more power when load is low and save less as load increases. Among them, FLOV+ saves the most static power under the same load and RP saves the least static power. When the load increases to the high region, FLOV+ and Baseline consume similar amounts of power.

3.4.3 Power-Gating Case Studies

In this subsection, we discuss cases of power-gating under different traffic patterns in low injection rates by varying the portion of power-gated cores. Figures 3.10 and 3.11 summarize the results in low load using Uniform Random traffic under different core power-gating scenarios. Similarly, Figures 3.12 and 3.13 show the results for Tornado traffic. The subplots 3.10a, 3.11a, 3.12a and 3.13a show average packet latency with breakdowns into router latency (number of hops $\times$ router pipeline latency), link latency (total link traversals), serialization latency (number of flits per packet) contention latency, and FLOV latency (number of FLOV routers/links traversed), while



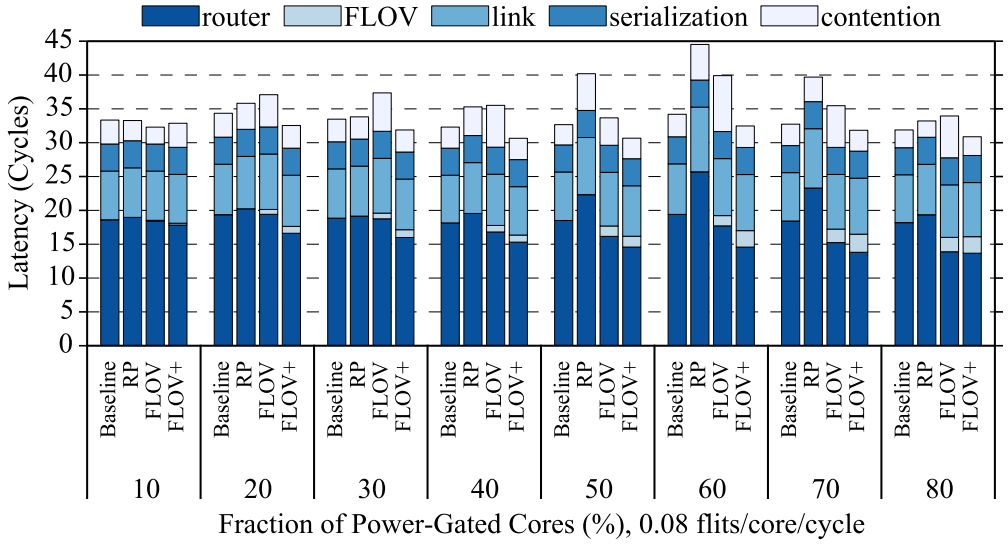
(a) Average latency breakdown



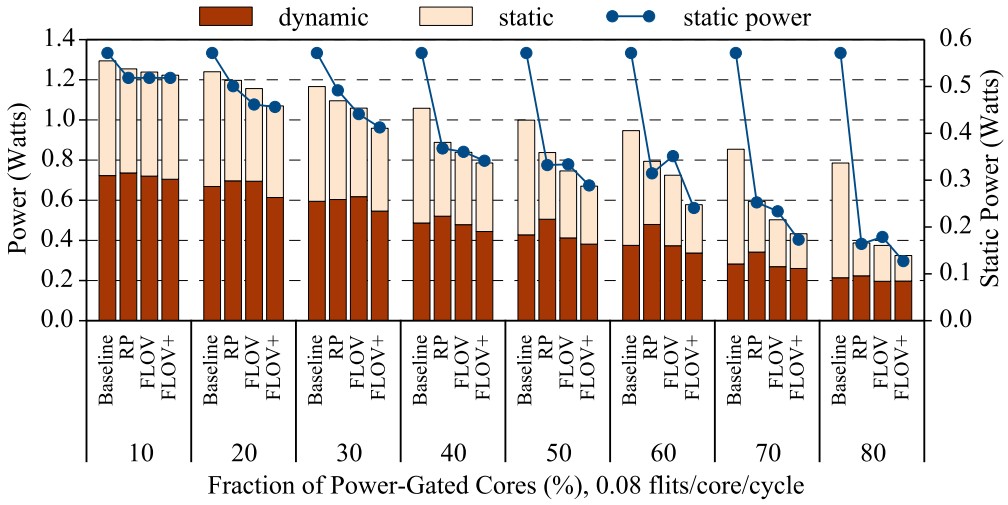
(b) Power consumption breakdown

Figure 3.10: Average latency breakdown (a) and power comparison breakdown (b) for injection rates of 0.02 flits/node/cycle with Uniform Random traffic pattern.

the subplots 3.10b, 3.11b, 3.12b and 3.13b show power consumption with breakdown into dynamic and static components. As NoRD incurs high latency overhead in low load, we only compare to Baseline and RP to reveal details in this subsection. For RP, we have run both aggressive and conservative modes and selected the mode that consumes less power.



(a) Average latency breakdown

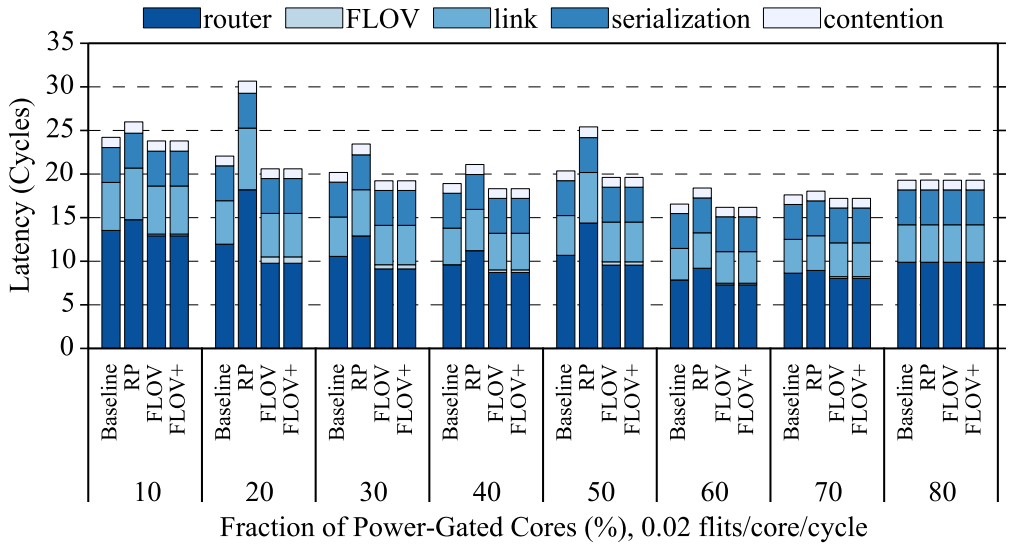


(b) Power consumption breakdown

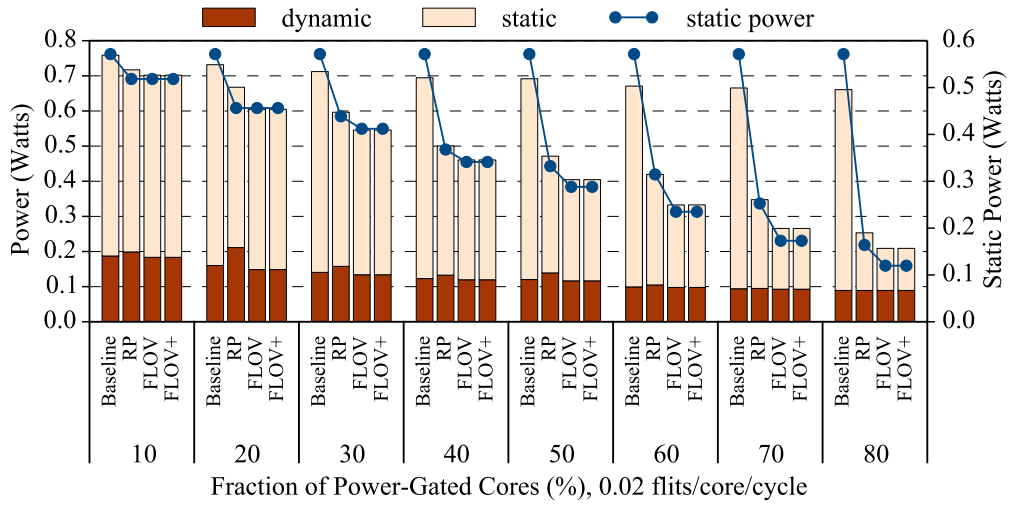
Figure 3.11: Average latency breakdown (a) and power comparison breakdown (b) for injection rates of 0.08 flits/node/cycle with Uniform Random traffic pattern.

3.4.3.1 Performance

Average latency comparison of FLOV, FLOV+ with RP and Baseline under Uniform Random and Tornado traffic patterns are shown in Figures 3.10a, 3.11a, 3.12a and 3.13a, respectively. FLOV and FLOV+ outperform RP across different traffic patterns and injection rates except that FLOV is worse than RP in cases of 20% and 30% of power-gated cores under Uniform Random traffic in



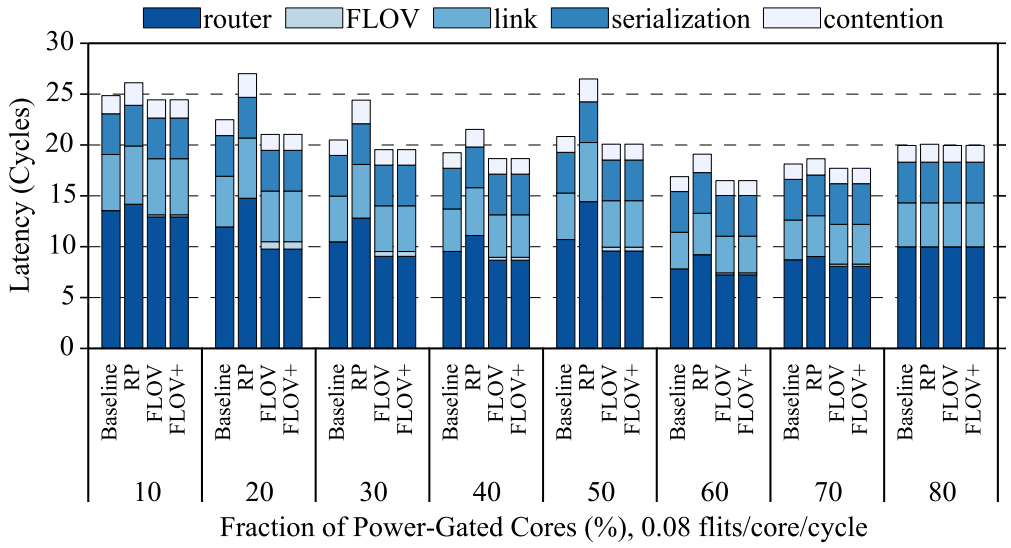
(a) Average latency breakdown



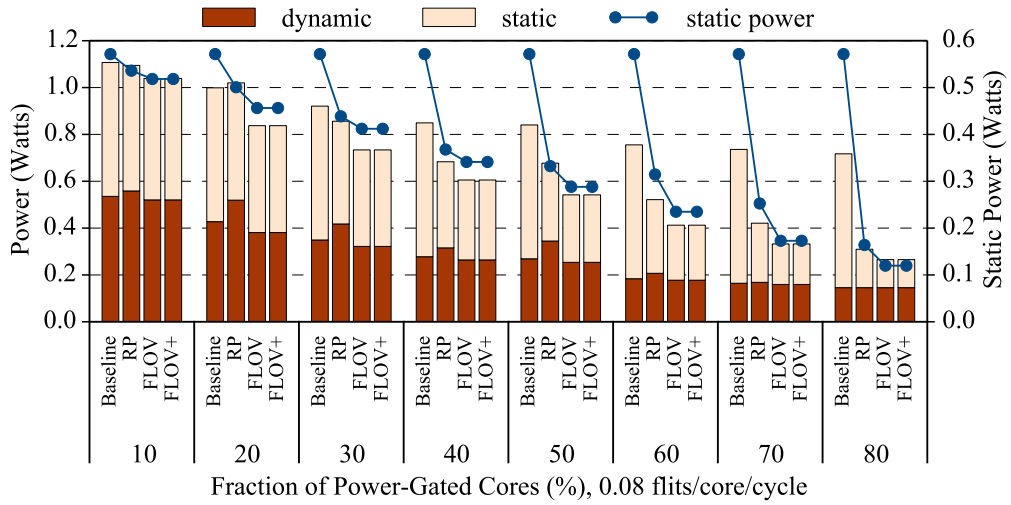
(b) Power consumption breakdown

Figure 3.12: Average latency breakdown (a) and power comparison breakdown (b) for injection rates of 0.02 flits/node/cycle with Tornado traffic pattern.

higher injection rate as shown in Figure 3.11a. In RP, a packet always routes through powered-on routers, which may be non-minimal, thereby increasing the path length. In contrast, FLOV and FLOV+ take advantage of all the links and route a packet through a minimal path in the best effort using FLOV links. Even when minimal routing is impossible for some cases in the escape sub-network, the average packet latency can be reduced since the FLOV links do not incur the 3-cycle baseline router per-hop latency, as the flit is only temporarily held in the FLOV latch for one



(a) Average latency breakdown



(b) Power consumption breakdown

Figure 3.13: Average latency breakdown (a) and power comparison breakdown (b) for injection rates of 0.08 flits/node/cycle with Tornado traffic pattern.

cycle. This can be observed clearly that the router latency for RP is larger than that of FLOV and FLOV+ due to detours. Under Uniform Random traffic, the FLOV latency increases as more cores are power-gated for the FLOV and FLOV+, which show increased FLOV link utilization. Under Tornado traffic, the communication occurs between two power-on nodes in the same row/column, and the routers in the rightmost column are always active. Therefore, less number of FLOV links are used, which leads to reduced FLOV latency. In Figure 3.11a, when 20% and 30% of cores

are power-gated, RP is slightly better than FLOV since it transits to conservative mode as dynamic power penalty is higher than static power saving, which also improves performance. This can also be observed in the configuration with 30% power-gated cores, where static power consumption for this injection rate (Figure 3.11b) is higher than 0.02 flits/cycle/core injection rate (Figure 3.10b). When 20% of cores are power-gated under Tornado, RP has lower latency in higher injection rate (Figure 3.13a) compared to in lower injection rate (Figure 3.12a). This is because it switches to conservative mode that improves latency. In the above cases, RP trades off static power savings for latency benefits. However, FLOV and FLOV+ achieve both power saving and performance guarantee in these cases.

Another observation is that as the injection rate increases from 0.02 to 0.08, the performance impact on RP is higher than on FLOV+. Figures 3.10–3.13 show higher contention latency for RP than FLOV+ when injection rate increases from 0.02 to 0.08. This is because certain routers, connecting different network partitions to ensure network connectivity, become network hotspots in RP. In contrast, The proposed FLOV+ routing algorithm avoids such network hotspots.

In Figures 3.12a and 3.13a, both FLOV and FLOV+ outperform Baseline with Tornado traffic. This is because in Tornado, the traffic injected to each router is destined to a router in the same row/column. Thus, FLOV and FLOV+ can use FLOV links with minimal paths and avoid the 3-cycle router latency. FLOV+ also shows better performance than Baseline even in Uniform Random traffic as shown in Figures 3.10a and 3.11a. This is due to the fast FLOV link and the proposed routing algorithm, which considers logical neighbor positions to provide best-effort shortest path.

In Figures 3.10–3.13, both FLOV and FLOV+ have relatively higher contention latency at higher injection rate. One reason is that packets have higher probabilities of being routed to the *AON* router column for guaranteed paths to the destinations, which may create congestion in the *AON* router column. Also, when packets are routed through consecutive FLOV links in a row/column, packet transmission may be delayed due to the longer credit round-trip latency across consecutive gated routers. However, the higher utilization of FLOV links compensates for the contention latency, which can be explained by the router and FLOV latency. Note that RP also tends to have

higher contention latency compared to the FLOV and FLOV+ because of the high probability of hotspot creation.

As the number of power-gated cores increases, FLOV, FLOV+ and RP all power gate more routers. However, only FLOV+ has stable latency while RP is the worst. From our study, we have observed that NoRD can be even worse in such low loads, since the attached core of the power-gated router is also off, thereby it may take longer detour in order to wake up routers when more cores are power-gated.

3.4.3.2 FLOV *Routing versus* FLOV+ *Routing*

FLOV+ routing algorithm has similar trend with FLOV routing algorithm as compared to RP. In Figures 3.10a and 3.11a, FLOV+ can achieve lower latency than FLOV. This is mainly due to the fact that FLOV+ has a higher chance to route packets through the shortest path instead of sending to the *AON* column to make turns, reducing the number of hops traversed with respect to FLOV routing. Such benefit is confirmed by the lower router latency in FLOV+ than that in FLOV. Interestingly, FLOV+ routing achieves better latency performance than Baseline. This is owing to the fast FLOV links. Since FLOV+ can take advantage of the one-cycle FLOV links on the minimal path, it can avoid the router pipeline stages as in Baseline for faster packet delivery.

3.4.3.3 Power Consumption

Figures 3.10b, 3.11b, 3.12b and 3.13b show the power consumption breakdowns into dynamic and static power. The static power is also depicted as secondary results. For both injection rates, the dynamic power consumption of FLOV and FLOV+ are lower than RP, since in RP every hop in the rerouted packet traversal requires the full router pipeline execution, whereas in FLOV and FLOV+ the intermediate power-gated routers use FLOV links that consume significantly lower power. RP also consumes more dynamic power than Baseline due to its non-minimal path rerouting of packets as the number of power-gated cores increases. At higher fractions of power-gated cores, FLOV and FLOV+ consume less dynamic power than Baseline by avoiding router pipelines.

For static power consumption, under Uniform Random traffic, FLOV and FLOV+ saves more

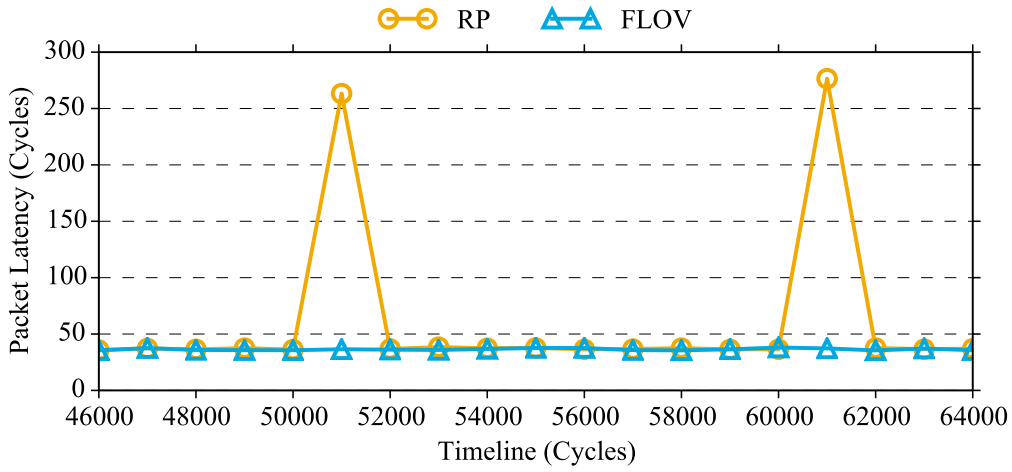
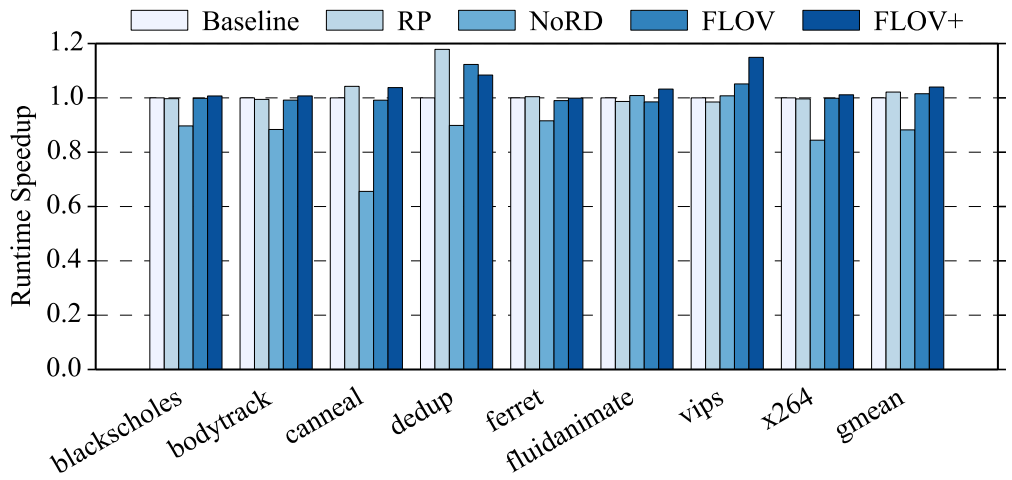
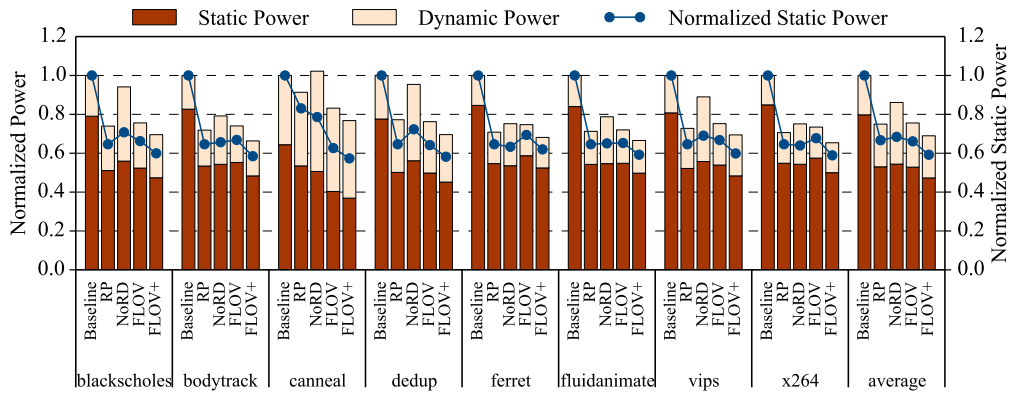


Figure 3.14: Reconfiguration overhead of RP and comparison with FLOV.



(a) Application runtime speedup



(b) Full system NoC power consumption breakdown

Figure 3.15: Full system simulation results: application runtim speedup (a), and normalized NoC power consumption (b) over Baseline.

than RP by power-gating more routers, which are required to be powered-on in RP to maintain network connectivity. FLOV+ consumes even less static power than FLOV, since FLOV needs to keep more routers in R-FLOV mode to guarantee performance due to routing limitations. While under Tornado traffic pattern, the only different trend compared to Uniform Random is that both FLOV and FLOV+ save the same amounts of power. This is because the communications only happen in the same row/column, which are handled in the same way in both routing algorithms.

3.4.4 Reconfiguration Overhead Analysis

We analyzed the impact of the network reconfiguration on packet latency by the comparison of RP and FLOV. Figure 3.14 shows average packet latency of FLOV and RP across the timeline of execution under Uniform Random traffic with 0.02 flits/cycle/node injection rate when 10% of the cores are power-gated. In RP, whenever the configuration of power-gated cores changes (at 50,000 and 60,000 cycles), the network has to be reconfigured by the Fabric Manager who recomputes and distributes the routing tables to the routers that are active in the next epoch (Phase I of reconfiguration protocol in RP). During reconfiguration, the network has to stall and no new injections are allowed except reconfiguration packets, which incurs additional queuing delays in packet latency. Our evaluations show that the reconfiguration time in RP Phase I is more than 700 cycles. We observe that the newly injected packets during this time experience significant queuing delays in RP. In FLOV, there is negligible handshaking overhead since the routers are power-gated in a distributed manner. So new packet transmissions can be initiated while some routers either power-gate or wake up independently.

3.4.5 Real Workload Evaluation

We also ran PARSEC 2.1 using `gem5` [130] with BookSim integrated for full system evaluation. For RP, we ran both aggressive and conservative mode, only the one with lower energy-delay product is selected for better efficiency. The system parameters are described in Table 3.1.

Figure 3.15a shows the runtime speedup of RP, NoRD, FLOV and FLOV+ over Baseline. It shows that FLOV+ has the least negative impact on performance, and even has a small improvement

(3.9%) compared to Baseline. FLOV and RP have negligible difference compared to Baseline while NoRD degrades performance by around 10%. Note that PARSEC applications have low network traffic loads, making them beneficial for power-gating. Since FLOV+ has a better routing algorithm compared to FLOV, and the one-cycle FLOV link compensates for the detour and round-trip credit loop latency, FLOV+ helps reduce network latency and improve performance. On the other hand, NoRD incurs high detour at low traffic load, leading to high latency. This phenomenon is more severe in larger network scales such as the evaluated 8×8 mesh network. Since the network traffic is low, RP has little negative effect on latency, thereby has similar performance as Baseline.

Figure 3.15b shows the dynamic and static NoC power breakdown normalized to Baseline. It shows that RP, NoRD, FLOV, FLOV+ reduces the total power consumption by 25%, 15%, 25%, and 31%, respectively. For static power consumption, RP, NoRD, FLOV, and FLOV+ achieve power reductions of 34%, 32%, 34% and 41%, respectively. Although NoRD saves the second most static power, the detour incurred by the ring introduces extra dynamic power consumption, offsetting the benefit and ending up saving the least total power. RP and FLOV have similar savings for both dynamic and static power, while FLOV+ saves the most power for both components. FLOV+ reduces power consumption by 8% and 20% compared to RP and NoRD, respectively. Note that in this work, routers in FLOV and FLOV+ adaptively change their power-gating modes independently through router voting. If every router is configured to G-FLOV mode, it can save much more power than RP but may not adapt to applications that have higher traffic as the adaptive one in this paper.

3.4.6 Area Overhead Analysis

In the proposed router microarchitecture, the modifications include 4 muxes and 4 demuxes in addition to the four output latches. The mux and demux selection signals are only toggled when the router wakes up or is power-gated, so the logic needed for the select signals is minimal. Every router has two sets of PSRs, where each entry incurs a 2 bit overhead (for power state). Hence the total overhead for the PSRs accounts to 16 bits (2 sets of 4-entry registers). The credit control logic is modified to be connected to the HSC so that the credit counters can be reset or zeroed based on the signals from the HSC. The additional overhead incurred due to this change

is mainly the connecting wires and minor modifications to the CCL logic for decoding the two HSC signals. The HSC requires 6-bit wires to connect the adjacent neighboring routers (4 bits for current and logical neighbor router power state change notifications, 1 bit for draining notification and 1 bit for physical neighbor assertion). In addition, the voting mechanism needs a 2-bit bus for voting snoop. This is approximately 0.1% of the baseline router area according to DSENT [48]. The HSC also includes the power state transition FSM implementation (4 states), which incurs minimal area overhead. The overall area overhead for the above components for a single router in 32-nm technology is estimated about $2.8 \times 10^{-3}\ mm^2$ which is 3% of the baseline router area. The power consumption of the HSC is also minimal due to the handshaking occurring only after long intervals of time (reconfiguration times) as shown in Section 3.4. None of the modifications incur any significant critical path delay and do not affect the frequency of the NoC operations.

3.5 Summary

In this chapter, FLOV is designed with a voting approach and adaptive routing algorithms to enable efficient and adaptive NoC power-gating using Fly-Over links. By router voting, each router collects votes from both dimensions of the network and adapts the power-gating mode among G-FLOV, R-FLOV and NO-FLOV with performance awareness, making it flexible for all different traffic loads. In addition, the proposed FLOV+ routing algorithm introduces more adaptability as well as logical neighbor information for routing decisions, achieving the best-effort minimal route. As a result, FLOV improves the throughput by more than 40% compared to the state-of-the-art, similar to the Baseline, while saving more power. Additionally, it scales well for different network sizes and outperforms other approaches. Our full system evaluation shows even performance improvement over Baseline by 3.9%, and power reduction by 31% and 20% compared to Baseline and state-of-the-art, respectively. In summary, FLOV not only reduces power consumption but also adapts to maintain performance, improving both power- and energy-efficiency.

4. APPROXIMATE COMMUNICATION FOR HIGH-THROUGHPUT NOC [1]

With the onset of the big data era, NoC as the communication backbone is under significant pressure due to the large data movement in data-intensive applications. Therefore, mechanisms that can reduce the communication traffic load in state-of-the-art NoCs become critical to cater to emerging data-intensive applications. Previous research [11, 12] has proposed to compress data in NoCs to reduce latency even at saturation load by exploiting the frequently repetitive patterns in applications. In addition, recent studies of approximate cache [132] also shows that there is high similarity among the data accessed by the workloads. Based on these two observations, this chapter develops an approximate communication framework, APPROX-NOC, that approximates similar data for compression-friendly patterns to achieve better compression ratio, thus reducing data movement and improving the effective network throughput.

4.1 Challenges in NoC Approximate Communication

Value approximation and compression are not cheap. Value approximation and data compression operations are on the critical path of the data packet preparation. The underlying compression techniques can incur considerable area and latency overheads. Dynamic compression requires storage for pattern tracking and data lookup while static compression incurs significant encoding and decoding logic. Furthermore, the approximation operation adds extra latency overhead on to the compression latency. Value range and error computation using complex multiplication is expensive and hence can offset the benefits from flit reduction achieved through approximation and compression. Thus, light-weight approximation design is required. Although the approximation engine can be treated as a plugin module, it is economical to tightly couple approximation and compression for lower-overhead implementation in terms of area, latency and energy.

[1] Reprinted with permission from "APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures" by Rahul Boyapati, Jiayi Huang, Pritam Majumder, Ki Hwan Yum and Eun Jung Kim, 2017. The 44^{th} Annual International Symposium on Computer Architecture (ISCA), pp. 666-677, doi: 10.1145/3079856.3080241,

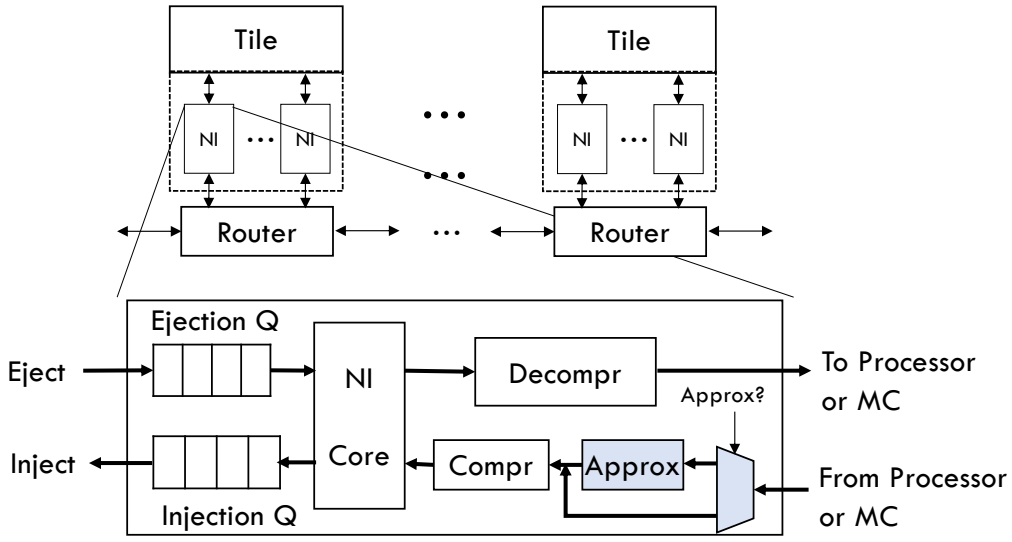


Figure 4.1: APPROX-NOC architectural overview.

Quality control is important. Approximable applications still require some Quality of Service (QoS) guarantees for output results. As mentioned in Rumba [133], it is also critical to differentiate overall quality control versus controlling errors in individual elements. Therefore, the approximation mechanism should be able to control data error rate individually in each cache block similar to Doppelganger [82] and also across the whole program execution. We assume that the programmer can annotate the QoS requirements, which is translated by compilers into error thresholds that are embedded in extended instructions.

4.2 APPROX-NOC Framework

In this section, we first describe the baseline multicore architecture and then detail the APPROX-NOC framework. The baseline architecture includes a collection of tiles connected via an NoC. Each tile may consist of core/accelerator, FPGA/ASICs, private caches, a slice of the last level cache and/or on-chip memory controllers (MCs). These tiles are connected to routers of the NoC, in either a one-to-one or many-to-one (concentrated) fashion depending on the NoC design. Each router connects to different components of a tile via Network Interfaces (NIs). The NI is responsible for message packetization and de-packetization, as well as flit fragmentation/assembly for

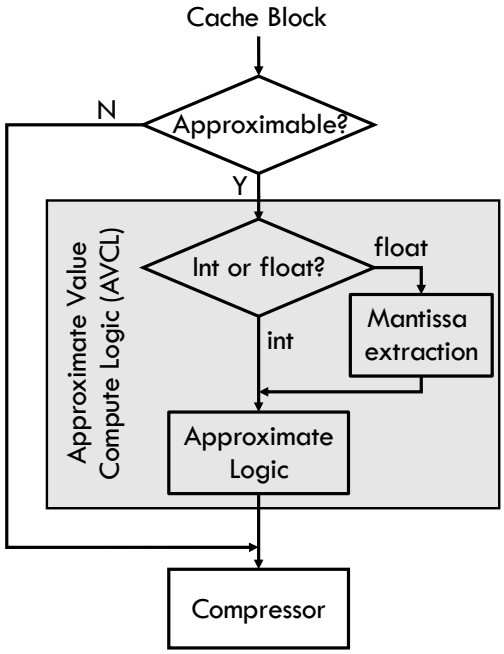


Figure 4.2: APPROX-NOC operation flowchart.

flow control. The NoC traffic consists of control packets for message passing/coherence and data request/reply packets. The size of the packet varies depending on whether it is an address/control packet or a data packet.

4.2.1 Architectural Overview

Figure 4.1 depicts the architectural overview of APPROX-NOC framework. Conventionally, data transmitted through NI from the tile is packetized and fragmented into flits in preparation for transmission. The packet is then injected into the router via the NI flit by flit. When the packet reaches its destination, the flits are assembled to restore the packet. APPROX-NOC consists of a value approximate module, namely VAXX, and an encoder/decoder pair for data compression in the NI. The encoder, of the underlying compression technique, tries to compress each word in the cache block to be transmitted and sends a small encoded index with metadata instead of the full value, thereby reducing the size of the packet. Before compression, the VAXX module facilitates value approximation to further improve the compression rate.

Figure 4.2 shows the flow of value approximation. For a cache block waiting to be transmitted,

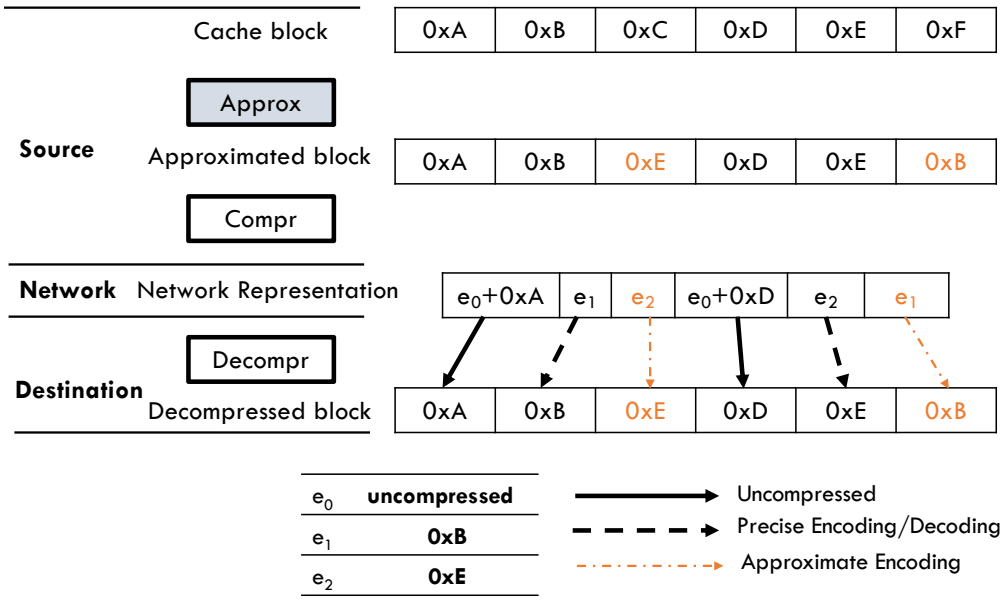


Figure 4.3: Compression and decompression of a 6-word cache block.

metadata containing the approximable flag and data type are firstly checked. If the cache block is not approximable, it bypasses the VAXX engine and starts compression. Otherwise, the data type is checked and the block is directly sent to the approximation logic if it is an integer. For floating-point data, we approximate only the mantissa fields and the approximation logic for integer values is reused to minimize the area and power overheads. The error range compute unit is also included in the approximation logic and the VAXX technique guarantees that the approximated data differs from the precise word within the preset error threshold. The approximated data blocks are then sent to the encoder for compression operation.

Figure 4.3 shows an APPROX-NOC working example, where a cache block (24 bytes with 6×4-byte words) is compressed at the source and decompressed at the destination. In this example, the encoder has two recorded patterns `0xB` and `0xE`, which can be encoded. The words `0xC` and `0xF` in the cache block are determined to be approximately similar to `0xE` and `0xB` by VAXX module, respectively. When the cache block is ready to be packetized, the encoder compresses the approximated block to an intermediate network representation (NR) by replacing the candidate data patterns with encoded code. The cache block, now in the NR form, is fragmented into flits

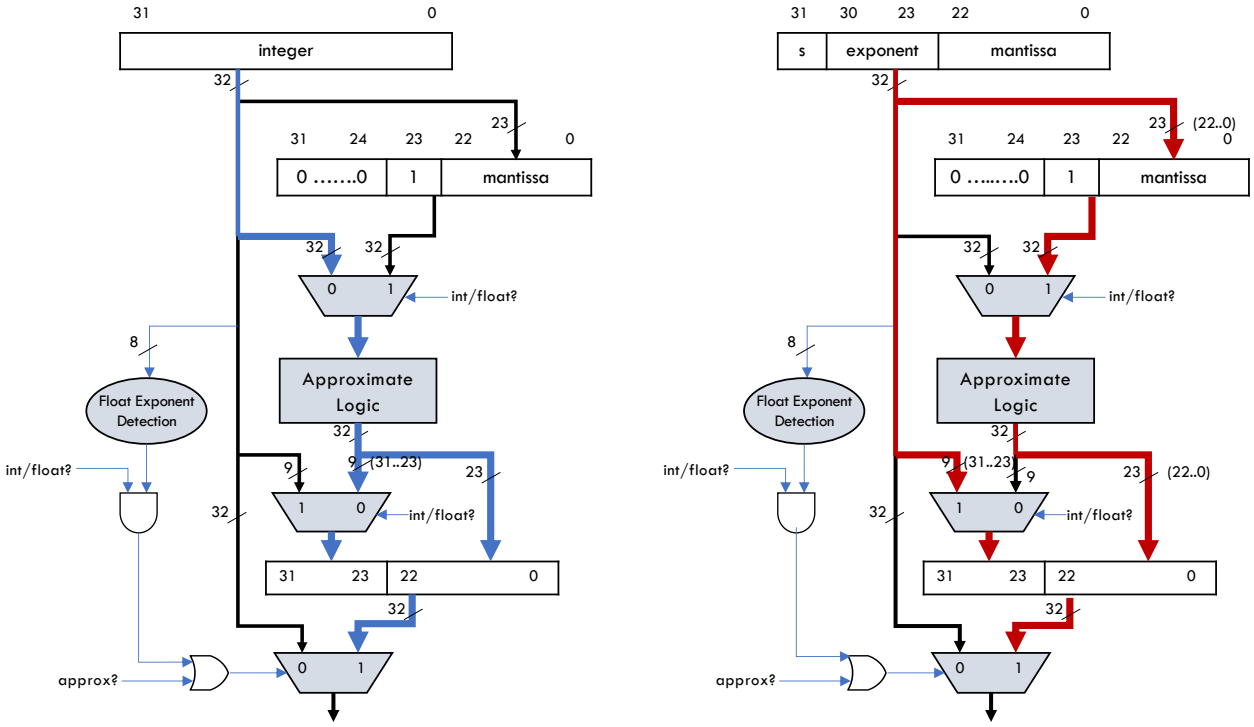


Figure 4.4: Approximate value compute logic.

and injected into the attached router. Note that the patterns `0xC` and `0xF` are compressed approximately only if the compiler annotates the data to be approximable. When the packet reaches its destination, the decoder at the destination uses the encoded code to recover the cache block from the NR. The decoded block now is an approximated version of the original cache block, with words `0xC` and `0xF` replaced by similar words `0xE` and `0xB`, respectively.

4.2.2 Approximation Logic Design

We propose the VAXX value approximate technique to compute an approximate value for a given data block within a predefined error threshold. In this work, we consider integer and floating-point value approximation. We approximate the cache block that are to be transmitted only if all the words in the block are approximable and this information is assumed to be carried with its access request. The core of VAXX is implemented in the Approximate Value Compute Logic (AVCL), which consists of floating-point mantissa extraction, error range compute and approximate logic.

For a given value, VAXX computes the variance by which the approximate value can deviate

from the precise value. For example, for a data pattern `0b1001` (value 9) and an error threshold of 25%, the range of values 8 (`0b1000`), 9 (`0b1001`), 10 (`0b1010`, 11 (`0b1011`)) can be potential matches. In this example, the 2 least significant bits can be don't cares (`0b10xx`) for the approximate matching pattern. This computation can be performed using multiplication/division operations, but such a design is too expensive.

To calculate the error range, we first compute the number of bits to represent the largest error a value can tolerate given the predetermined threshold. We simplify the logic by precomputing the number of shift bits $\lceil \frac{100}{e} \rceil$, where *e* is the error threshold ($e\%$). Then the value is shifted right to compute the error range:

$$error\_range = given\_value \times \frac{e}{100} = given\_value \div \frac{100}{e} \approx given\_value \gg \lceil \frac{100}{e} \rceil$$

For example, for an error threshold of 25%, the number of shift bits is 4 ($\lceil \frac{100}{25} \rceil$). Hence, when the given value is 128, the *error_range* can be easily calculated to be 32.

Floating-point value approximation is more complicated than integer due to the representation. A floating-point value is represented as: $(-1)^{sign} \times (1 + .mantissa) \times 2^{(exponent-bias)}$. We propose to approximate only the mantissa field of floating-point values. The mantissa part is extracted and transformed to scale to the size of an integer value, by padding the most significant bits with zeros. To transform and scale the value of a floating-point value, we extract the 23-bit mantissa part and concatenate it with a higher bit 1 to form the significant, where the exponent part is scaled out. In this manner, both integer and transformed floating-point variables can share the same approximate logic to maintain low overhead. Figure 4.4 shows the AVCL design in detail, where the datapaths taken by the integer and floating-point variables are represented separately for ease of understanding. The float exponent detection logic determines whether to bypass the approximation unit, for floating-point variables, whenever the exponent is 0 or all 1's, which represent special numbers such as zero, denormalized numbers, infinity and NaN. For variables that are annotated to be non-approximable, the AVCL logic is bypassed.

The proposed APPROX-NOC framework can use the VAXX on top of any data compression mechanisms. However, trivially adding VAXX module on top of NoC data compression can be

Index	Pattern encoded	Data Size
000	Zero run	3 bits
001	4-bit sign-extended	4 bits
010	One byte sign-extended	8 bits
011	Halfword sign-extended	16 bits
100	Halfword padded with a zero halfword	16 bits
101	Two halfwords, each a byte sign extended	16 bits
111	Uncompressed Word	32 bits

Figure 4.5: Frequent pattern compression [134].

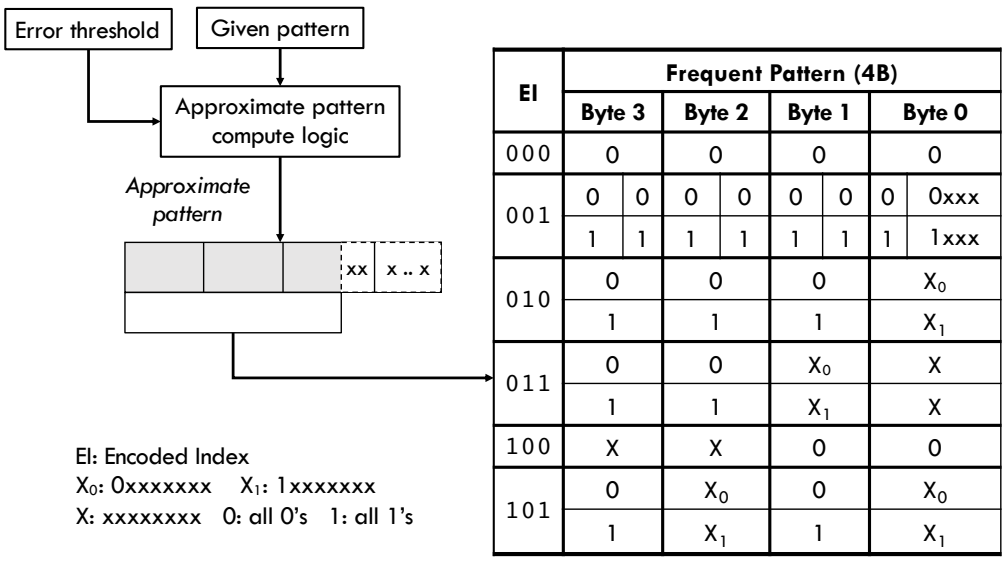
expensive and unscalable due to the computation as well as latency overhead. Therefore it is critical to design microarchitectures that optimize the functionality of approximation and compression as a whole in terms of area/latency/power. To this extent, we showcase two microarchitectural implementations of APPROX-NOC with two state-of-the-art NoC data compression mechanisms in the next section.

4.3 APPROX-NOC Implementation

In this section, we first present the VAXX implementation for an underlying static frequent-pattern compression, namely FP-VAXX. Next we describe the implementation for a dynamic dictionary-based compression, namely DI-VAXX. Then we discuss the latency overhead due to approximation.

4.3.1 Frequent-Pattern Mechanism

First, we briefly describe the Frequent-Pattern Compression (FP-COMP) technique and then propose microarchitectural implementation for FP-VAXX. Previous research [134] has proposed an FP-COMP mechanism for data compression and [83] has extended it for NoCs with low-overhead decompression, which we adopt in this work. It compresses a static set of frequent patterns as shown in Figure 4.5. FP-COMP detects a match on one of the patterns and sends adjunct data along with the encoded index. Therefore, FP-COMP incurs additional decompression complexity due to variable length compression.



EI	Frequent Pattern (4B)			
	Byte 3	Byte 2	Byte 1	Byte 0
000	0	0	0	0
001	0 0	0 0	0 0	0 0xxx
	1 1	1 1	1 1	1 1xxx
010	0	0	0	X_0
	1	1	1	X_1
011	0	0	X_0	X
	1	1	X_1	X
100	X	X	0	0
101	0	X_0	0	X_0
	1	X_1	1	X_1

Figure 4.6: FP-VAXX microarchitecture.

*4.3.1.1 FP-*VAXX *Implementation*

Figure 4.6 shows the microarchitectural overview of the VAXX implementation for FP-COMP. For each data word, we first compute the approximate pattern using the AVCL. Once the don't care bits of the word are determined, the rest of the data word (shaded portion in the figure) is matched with the corresponding portion of the frequent patterns in the pattern matching table (PMT) to find a match and compress on a frequent pattern hit. We propose to utilize a content addressable memory based (CAM) structure to implement the PMT structure for fast matching. By doing this only the bits that can be approximated, according to the value error threshold, are candidates for approximation and the rest of the pattern must be a complete match to a frequent pattern for compression. For data that is not annotated to be approximable, the AVCL is bypassed to enable lossless compression for given data words.

4.3.2 Dictionary-Based Mechanism

Dictionary-based Compression (DI-COMP) keeps track of recurring data patterns dynamically and maintains an encoded-index consistency between senders and receivers. To track recurrent data patterns and maintain the dictionary, we propose to use a table-based mechanism similar to

the one proposed in [86].

Figures 4.7a and 4.7b show the microarchitecture of an example encoder and decoder pattern matching tables (PMTs) with size of 4 entries, respectively, in a (3×3) NoC. In the encoder PMT each entry contains a data pattern, frequency counter and a vector of encoded indices, each corresponding to one destination router (decoder), i.e. in an *N*-node NoC, each entry has a vector of *(N-1)* encoded indices. For a data pattern in the encoder PMT, the vector of indices indicates whether it can be compressed for a particular destination in the network. In addition, the encoder PMT can have different encoded index values for different destinations, for the same data pattern, since each decoder works independently. The decoder PMT entries consist of the data pattern, frequency counter, encoded index and a vector of *(N-1)* valid bits, one for each of the *N-1* encoders. The decoders detect recurrent data patterns and place them in decoder PMTs while sending an update notification to the encoder, with the new encoded index. The vector of valid bits indicates all the encoders that have this data pattern in their PMTs and is used when replacements happen to invalidate the pattern at all encoders. In the example shown in Figures 4.7a and 4.7b, the encoder PMT at node 3 stores the indices for patterns `0b0000` and `0b1111` for destination 6 while the decoder PMT at node 6 has valid bits set for the respective patterns for node 3.

*4.3.2.1 DI-*VAXX *Implementation*

In order to optimize the microarchitectural cost of VAXX implementation with DI-COMP mechanism, we modify the operational flow of the approximation with a tight integration, instead of passing a given data block through the AVCL before compression. We propose to compute the approximate pattern for every reference pattern, at the time of the pattern being recorded, in the DI-COMP scheme and save the approximate versions of the reference patterns. Therefore, any given pattern can be compared to a set of approximate patterns for fast matching, thereby removing AVCL from the critical path of the packetization.

We propose to use a Ternary Content Addressable Memory (TCAM) structure to optimize the approximation and compression latency. TCAMs function similar to a CAM. In addition to 0 or 1, a third state of "`x`" (don't care) is allowed, i.e., in a TCAM we can actually store `0b10xx`

Data pattern	Frequency counter	Vector of indices							
		0	1	2	4	5	6	7	8
0000	--			11			00		
0101	--				00				
1011	--					00			
1111	--						01	11	

(a) Encoder PMT at Node 3

Data pattern	Frequency counter	Index	Vector of valid bits							
			0	1	2	3	4	5	7	8
0000	--	00	0	0	1	1	0	0	0	1
1111	--	01	0	0	0	1	1	0	0	0
1100	--	10	0	0	0	0	0	0	1	0
1110	--	11	1	0	0	0	0	0	0	0

(b) Decoder PMT at Node 6

Figure 4.7: The encoder PMT at Node 3 and the decoder PMT at Node 6.

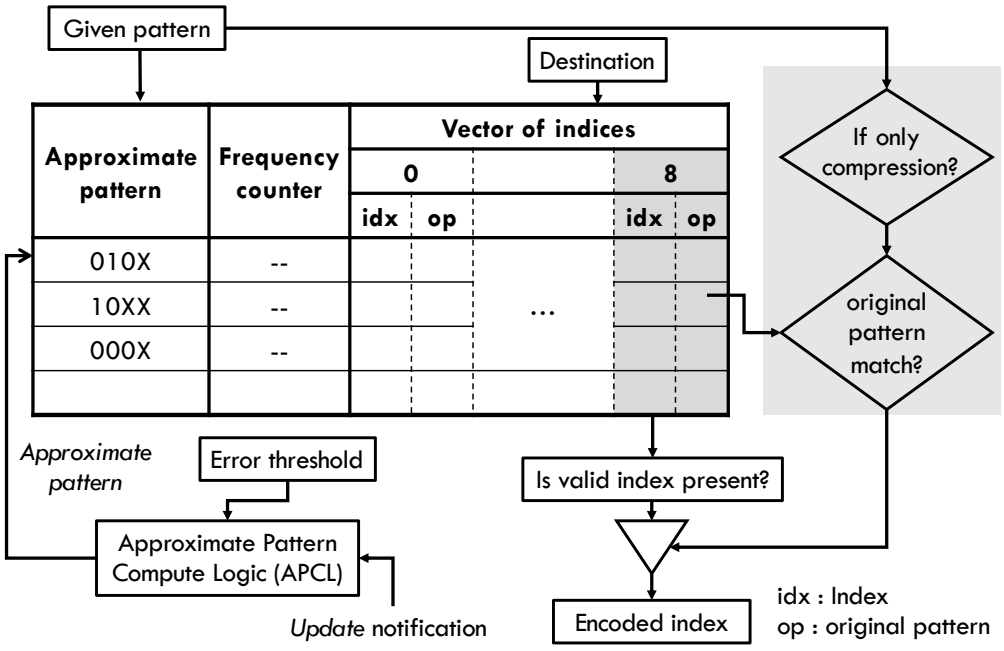


Figure 4.8: DI-VAXX microarchitecture.

for a pattern `0b1001` and the table entry results in a match for the patterns `0b1000`, `0b1001`, `0b1010` and `0b1011`. The decoders utilize a regular CAM structure to recover the original pattern `0b1001` based on the index. The microarchitecture of the TCAM-based encoder PMT is shown in Figure 4.8 and the operation is explained as follows.

- The receivers (decoders) detect frequent data patterns and send an update to the encoders to reflect in the PMT.
- When the encoder receives an update, instead of just storing the original pattern, it computes the approximate pattern with don't care bits (e.g. `0b1001` to `0b10xx`) based on the error threshold using APCL. Then the encoder records the approximate pattern in the TCAM and stores the index for the corresponding receiver. If there is a matching TCAM entry, the encoder just updates the index.
- When a data pattern arrives at the encoder, the TCAM is accessed. In case of a hit, the encoded index is used for compression. Thus, we can reduce the latency overhead on the critical path of compression.

For data packets that are not annotated for approximation, this TCAM-based mechanism is insufficient to provide exact compression, because a match has no guarantee that the recovered pattern at the receiver is the same pattern the sender intended to transmit (e.g. 8 can match in TCAM and be recovered as 9). To facilitate exact matching along with approximate matching, we propose to add storage capability in the encoders for the original patterns in addition to the TCAM entry (approximate pattern). Figure 4.8 shows the encoder PMTs with the original pattern storage. Each TCAM entry can have multiple original patterns because different receivers (decoders) could have detected different patterns in the range of values. We propose to store multiple original patterns for each entry. When a data pattern that cannot be approximated arrives at the encoder, first the TCAM entry is matched and then an exact match on the corresponding original pattern (based on receiver) is checked before compressing it. The storage overhead can be optimized by storing only the bits of the original pattern that were made don't cares in the approximate pattern.

4.3.3 Latency Overhead

We assume a three-cycle compression latency (two cycles matching and one cycle encoding) and two-cycle decompression latency overhead for each cache block as mentioned in [83]. To ensure that the DI-VAXX and FP-VAXX matching can happen within the provisioned compression latency, we propose parallel hardware matching units based on the latency overhead evaluations. In case of DI-VAXX and FP-VAXX, we have 8 parallel TCAM matching units since two matches per cycle in each unit is possible based on the model from [135]. Additionally, FP-VAXX requires 8 APCL units.

We also propose to use two latency hiding optimizations to reduce the compression overhead. First, we propose to perform the virtual channel arbitration of the packet, using the header flit which is not compressed, in parallel with the compression. We amortize the compression overhead with the NI queuing time, i.e, if there are previous packets waiting in the queue, the compression overhead would not add to the critical path of packet latency.

4.4 Evaluation

In this section, we present the experimental setup followed by the evaluation of the APPROX-NOC framework.

4.4.1 Methodology

Experimental Setup. We evaluate APPROX-NOC using a cycle-accurate, in house NoC simulator and the `gem5` full system simulator [130]. We implement the DI-VAXX and FP-VAXX mechanisms in addition to the DI-COMP and FP-COMP mechanisms [86, 83] in both the simulators. For detailed network impact evaluations, we use the NoC simulator where the default error threshold is set to 10% and the percentage of approximable data packets is set to 75%. We also perform sensitivity studies to show the impact of varying these parameters. To evaluate the impact of APPROX-NOC on the overall application output error, we extend a Pin-based [136] coherent multiprocessor cache simulator [137] for instrumentation. We hand-annotate the benchmarks mentioned below, similar to Doppelganger [82], to identify the data regions which can be approximated. The

Table 4.1: APPROX-NOC System Configuration

Parameter	Configuration
Processor	32 Out-of-Order Cores @ 2GHz 32KB L1I cache and 64KB L1D cache, 2-way 2MB L2 cache and 16 directories MOESI hammer cache coherence protocol
NoC	4×4 2D concentrated-mesh 2GHz three-stage router 4 virtual channels with 4-flit buffer, 64-bit flit wormhole switching and XY routing
Error threshold	5%, 10% (default), 20%
Approximable data packet ratio	25%, 50%, 75% (default)
Dictionary-based mechanisms	8-entry pattern matching table

VAXX mechanism uses the knowledge of the data type (floating point or integer) of variables in each benchmark to determine the approximation operation. An important consideration while hand-annotating approximable data regions of benchmarks is the data type of the variables being determined to be approximable. We assume that the data type of the cache block being compressed is known to APPROX-NOC and conservatively only compress cache blocks in which all the words have the same data type. This is because knowledge of the data type of each word would require significant metadata overhead. We use `gem5` to evaluate the performance effect of our approximation mechanism on the overall system. The APPROX-NOC configuration and the NoC parameters used for our evaluation are listed in Table 4.1.

Workloads. We evaluate the PARSEC [138] benchmark suite with *simlarge* input data set, which have been previously used for evaluating approximation mechanisms [139]. In addition, we explore the approximation opportunities in big data analytics by modifying *SSCA2* [140], a data-intensive graph benchmark, to evaluate betweenness centrality (BC) in real-world graphs [141]. BC is a popular graph analysis technique to identify important entities in large-scale networks.

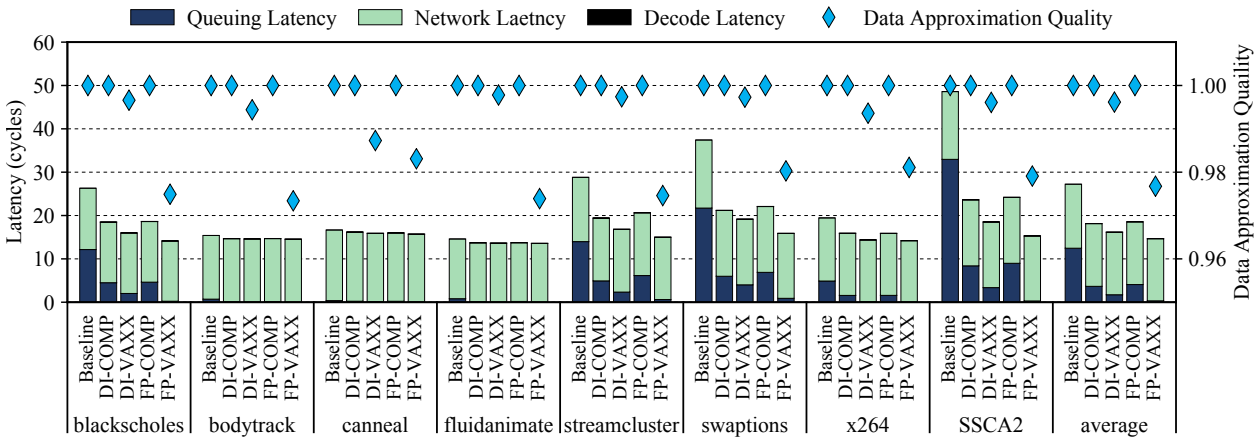


Figure 4.9: Average packet latency breakdown and overall approximation quality.

We approximate the floating-point pair-wise dependencies that are used for centrality calculation. Such applications in big data analytics can leverage approximation in data segments (e.g. weights in graphs) within a tolerable error margin since most algorithms approximate the result by only evaluating on a subset of the data with sampling. We run the benchmarks using `gem5` [130] to evaluate the impact of our mechanisms on the system performance and to collect the communication traces for the region of interest, which are then fed into our NoC simulation environment and simulated for 100 million cycles for detailed NoC evaluations. To evaluate the throughput impact, we use synthetic workloads. We collect the data injected at each node from the `gem5` benchmark traces and utilize the data traces to create data packets in the synthetic workloads. In this way, the synthetic workloads can be used to vary the traffic pattern/injection rate but the data being communicated can be kept constant and correlated with data locality in the benchmarks.

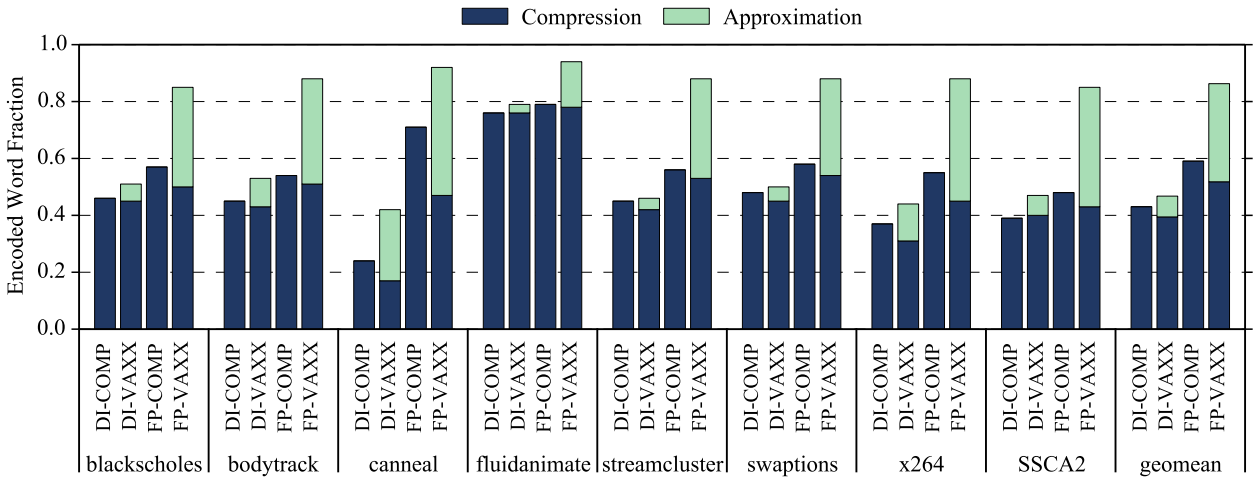
4.4.2 Results and Analyses

In this section, we present the NoC performance evaluation of APPROX-NOC using benchmarks from different application suites and synthetic workloads. We first analyze the performance impact of APPROX-NOC on the average packet latency, compression ratio, then use synthetic workloads to evaluate the impact on network throughput.

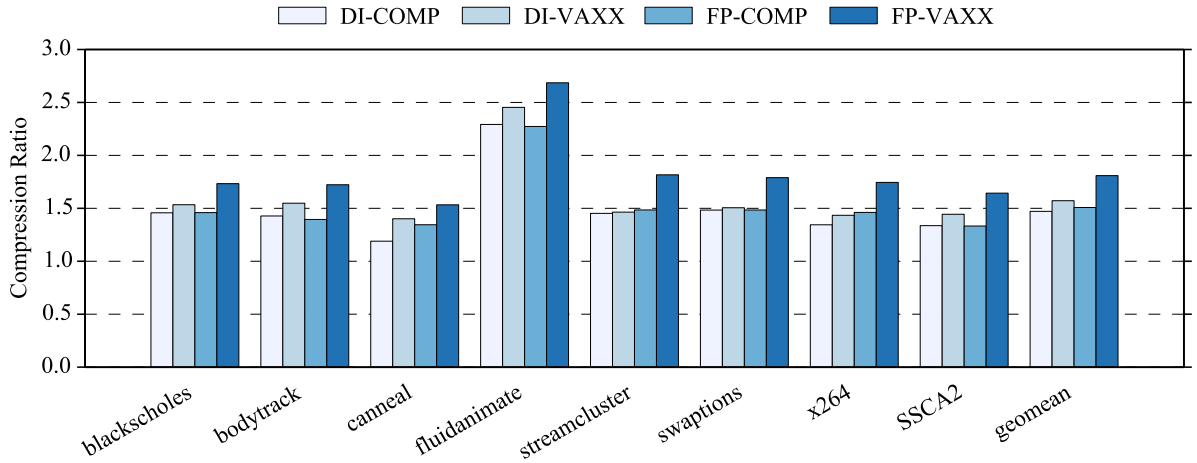
4.4.2.1 Performance Analysis

Average Packet Latency. The average packet latency comparison, in a 4×4 2D concentrated-mesh NoC, for the two implementations of APPROX-NOC is shown in Figure 4.9. Across the benchmarks, DI-VAXX reduces the average packet latency by 11% with respect to DI-COMP and 40.7% compared to Baseline. FP-VAXX achieves up to 21.4% and 46.5% latency reduction compared to FP-COMP and Baseline, respectively. This is mainly due to the fact that approximation allows for more reduction in the number of injected flits leading to performance benefits, especially when the network is congested during the bursty phases. The large packet latency reduction in *SSCA2* graph benchmark is owing to the data intensive nature of the application. With a large data set, the limited cache size cannot hold the whole working set of the benchmark, and hence its irregular data accesses incur large volume of data movement. We expect that data-intensive applications that have a high ratio of data movement to computation traffic will benefit from APPROX-NOC.

Note that the queuing delay decreases significantly by introducing approximation, which reduces the blocking delays of single-flit control packets caused by the long data packets. The decoding latency of the average packet latency is negligible because it is amortized over the large number of control packets, and also compensated by the reduced queuing latency. In addition, it is interesting that the VAXX techniques have a larger impact on packet latency with the FP-VAXX mechanism compared to the DI-VAXX. This is because the DI-VAXX mechanism needs to learn the data locality at the beginning of each new communication phase by tracking and updating its locality tables, thereby losing approximation opportunities. In contrast, the FP-VAXX can use the static patterns across the whole program execution. For *bodytrack, canneal* and *fluidanimate*, VAXX only achieves moderate improvement. This is because packets in these benchmarks have low queuing and network latency and the flit reduction translating to lower serialization latency is offset by the approximation/compression/decompression overheads. In addition, the percentage of data packets injected is very minimal compared to control packets, and hence the reduction in data flits does not show a significant impact on overall packet latency. The low queuing latency also supports the argument of low data to control packet ratio.



(a) Fraction of encoded words



(b) Compression ratio

Figure 4.10: Fraction of encoded words breakdown to exact compression and approximation (a) and compression ratio improvement of VAXX (b).

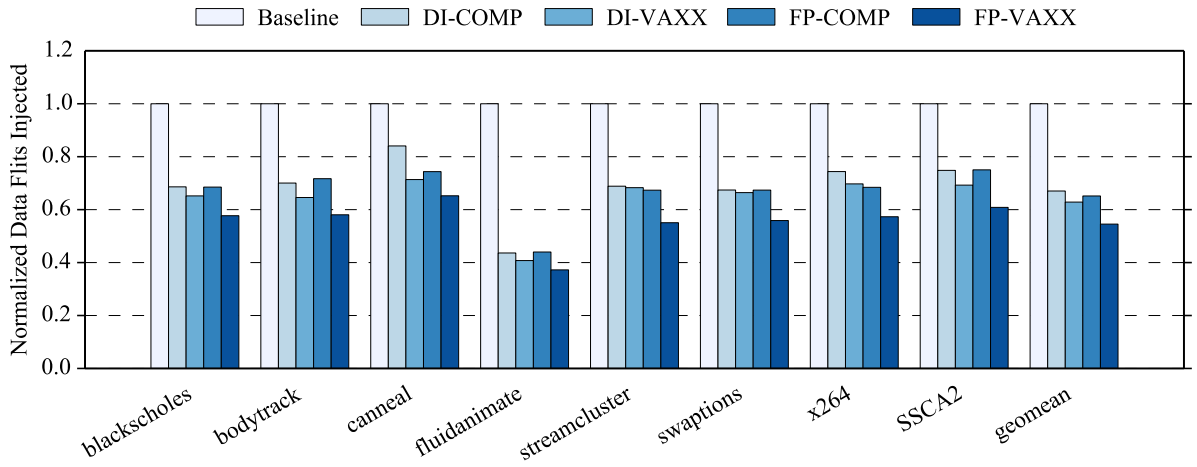
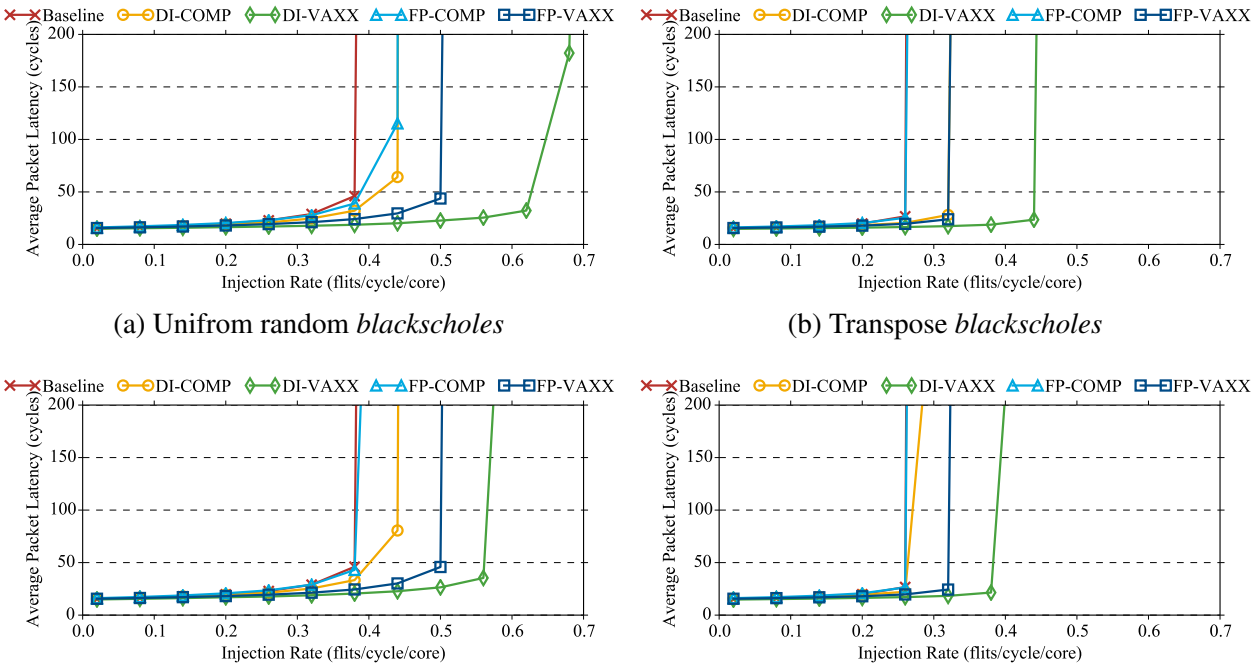


Figure 4.11: Reduction in number of injected flits.

Approximation Effectiveness. Figure 4.11 shows the reduction in traffic load by plotting the number of data flits injected under each APPROX-NOC mechanism. DI-VAXX reduces the number of injected data flits by 3% and 38% compared to the DI-COMP and Baseline, respectively. Similarly, FP-VAXX reduces data flit volumes by 19% and 45% with respect to FP-COMP and Baseline, respectively. The moderate traffic reduction in *streamcluster* and *swaptions* benchmarks when juxtaposed with the large latency improvement seems to be counter intuitive. This can be explained by two reasons. Firstly, the value approximation enables injection acceleration for critical data, thereby translating to reduced queuing latency for the many short packets that are blocked. Therefore, even though the reduction in injected flits is small, the effective resulting latency reduction can be amplified. In addition, in dynamic compression, approximation may change the learning of DI-COMP, which might affect the compression chance of data that is required to be precise. Hence, the overall flit reduction might be smaller due to changes in the operation of the DI-COMP learning. But overall we observe that the network traffic reduction translates to average packet latency improvement.

This is further supported by Figure 4.10a, which shows the breakdown of the fraction of encoded words to exact compression and approximated compression. We observe that VAXX increases the encoded word fraction by up to 18% for DI-VAXX compared to DI-COMP and up to 37% for FP-VAXX over FP-COMP. Figure 4.10b shows compression ratio of compression and approximation. DI-VAXX and FP-VAXX enhance the compression ratio by up to 21% and 41% compared to the corresponding compression schemes, respectively. On average, the two VAXX implementations increase compression ratio by 10% and 30%. Figures 4.10 and 4.11 show that the reduction in number of injected flits does not scale proportionally to increase in compression ratio due to approximation. This is because of internal fragmentation in the 8B flits where a large portion of the tail flit can be empty, since the compressed data might not be a multiple of 8B.

Data Value Quality. Figure 4.9 also depicts the data value quality for each benchmark. Even though the error threshold is checked for approximating each word, the incurred error differs from word to word. So we compute the actual overall data error incurred across the benchmark execution



(a) Unifrom random *blackscholes*

(b) Transpose *blackscholes*

(c) Uniform random *streamcluster*

(d) Transpose *streamcluster*

Figure 4.12: Throughput analysis with different benchmark data traces under uniform random and transpose traffic patterns.

and show the overall data value quality. Across the benchmarks, though we allow for a 10% error rate, the effective data value quality is higher than 97%, which is due to a portion of the words being compressed without error and most of them matching with close proximity. Note that this is the quality of the integer and floating-point data values. We analyze how this variance translates to overall application output error later.

4.4.2.2 Throughput Analysis

We use synthetic workloads to analyze the impact of APPROX-NOC on the network throughput. Figure 4.12 plots the throughput of APPROX-NOC compared against the Baseline, DI-COMP and FP-COMP. We select data traces from *blackscholes* and *streamcluster* benchmarks for both *uniform random* and *transpose* traffic patterns. The simulations are run for 1 million cycles and we assume a 25:75 data to control packet ratio to emphasize the significance of APPROX-NOC when a large amount of data is communicated. When compared to the compression schemes, VAXX improves

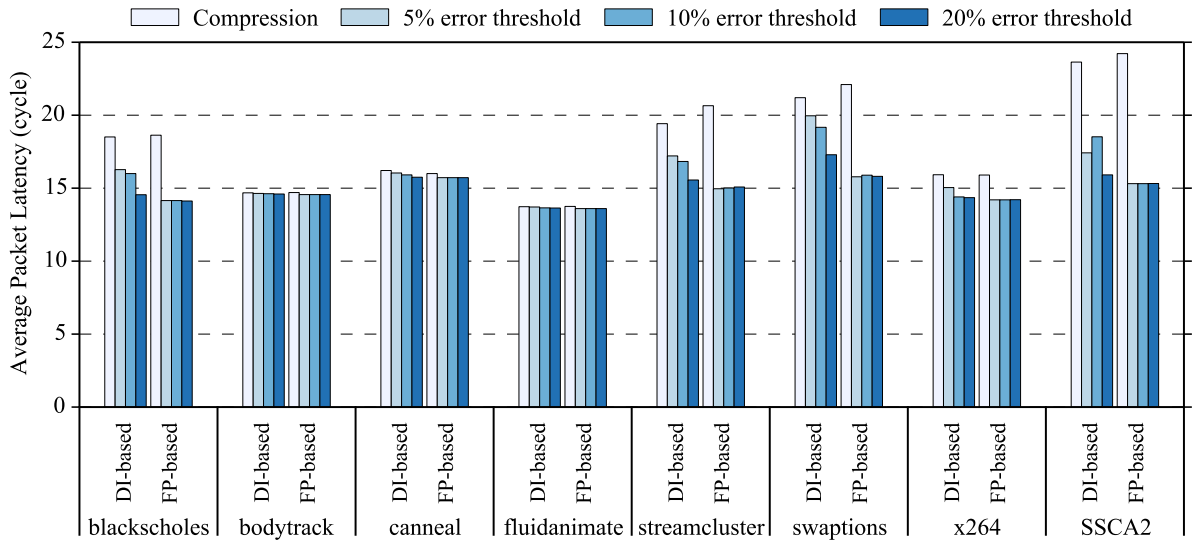


Figure 4.13: Error threshold sensitivity analysis.

the throughput by up to 40% for *uniform random* and 69% for *transpose* traffic patterns. This gain is achieved by reducing the effective injection load due to approximating data. The huge increase in throughput compared to the latency benefits observed from benchmarks can be attributed to the larger ratio of data packets being injected. Another interesting observation is that DI-VAXX performs better than FP-VAXX. This is because of higher data value and temporal locality in the synthetic workloads at higher injection rates with larger data packet ratio. From our observations, the dynamic dictionary-based scheme tends to work well for applications with high data locality and intensive data movement due to its learning capability, while the static frequent pattern scheme tends to work well for applications with many frequent patterns and short communication phases without learning.

4.4.3 Sensitivity Studies

In this section, we show the sensitivity of APPROX-NOC to the error threshold and the percentage of approximable data packets.

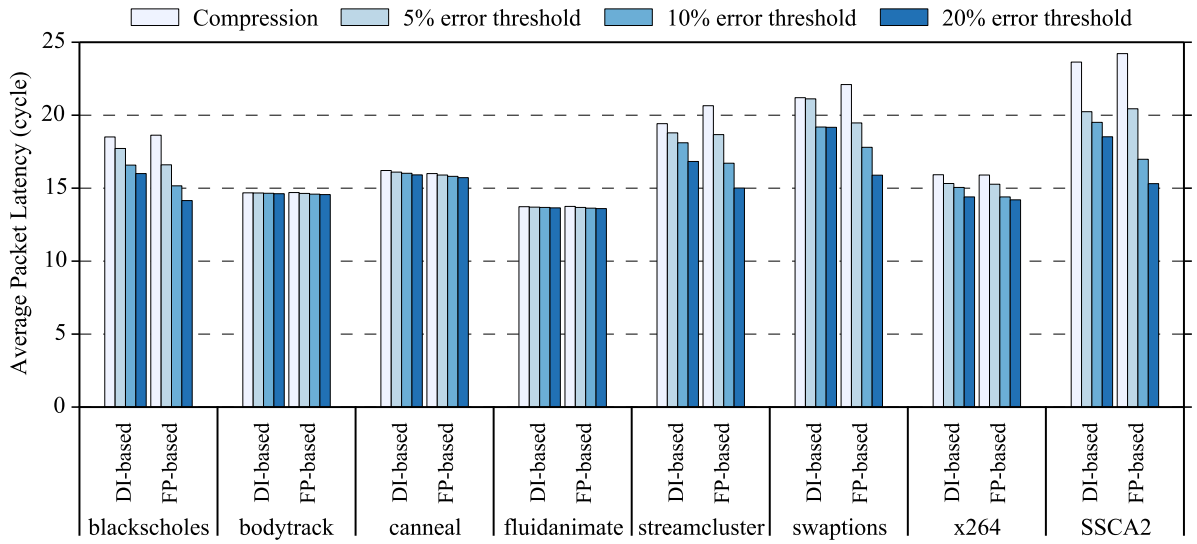


Figure 4.14: Approximable packets ratio sensitivity analysis.

4.4.3.1 Error Threshold

Figure 4.13 shows the average packet latency across the APPROX-NOC mechanisms for all the benchmarks by varying the error threshold. As the error threshold increases from 5% to 10% (default) to 20%, the impact of APPROX-NOC on packet latency amplifies due to the better chance of approximate matching. One interesting observation is that the FP-VAXX mechanism does not seem to have a significant impact on the packet latency even though with a higher error threshold. This is because our approximation technique can translate the approximate value into a higher compression ratio even with a small error threshold. It is well matched with the static frequent pattern compression. Despite the moderate latency improvement, FP-VAXX also incurs more overall error compared to DI-VAXX. This is because FP-VAXX always tries to match with the highest priority frequent pattern in the pattern matching table even though an exact match is available at lower priority. Therefore, some of the exact matches, when error threshold is lower, might be converted into approximate matches as the error threshold is increased. These scenarios can lead to additional error without latency benefits.

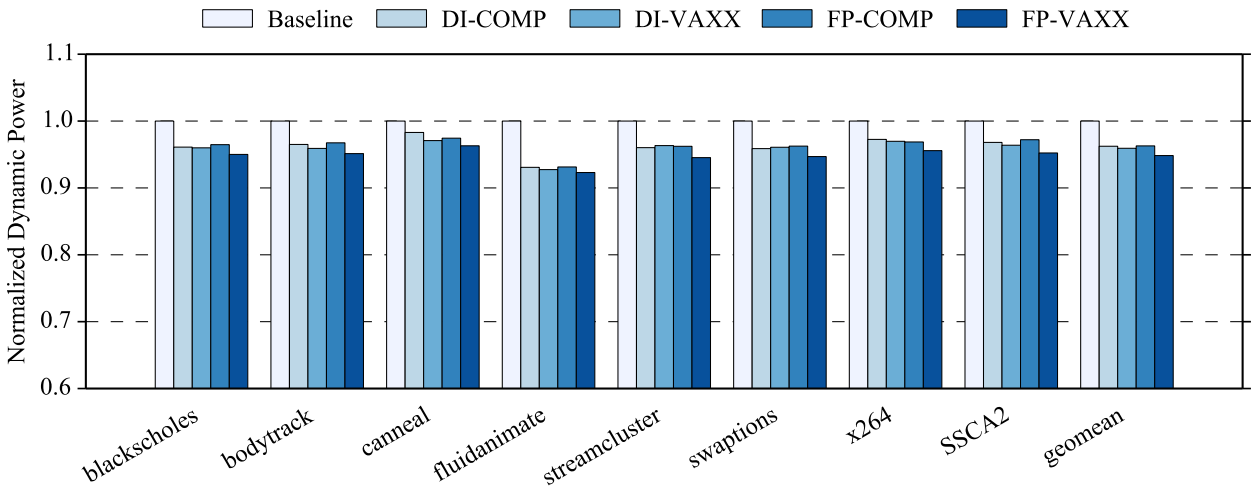


Figure 4.15: Dynamic power consumption normalized to Baseline.

4.4.3.2 Approximable Packets Ratio

Figure 4.14 shows the average packet latency for APPROX-NOC across benchmarks as the percentage of approximable packets varies. The packet latency benefits improve as the percentage of approximable packets increases due to the enhanced chances of approximate matching. This can be observed significantly in *SSCA2, swaptions, streamcluster* with both DI-VAXX and FP-VAXX, while the other benchmarks do not show compelling latency reduction as the percent of approximable packets is increased. This is due to the low queuing latency and small data-to-control packet ratio for these benchmarks, leading to minimum impact of data flit reduction on the overall network latency.

4.4.4 Full System Impact Analysis

In this section, we use Pin-based [136] functional model and `gem5` [130] performance model to evaluate the impact of APPROX-NOC on the overall system. We present the overall application output errors followed by the overall runtime impact due to approximation on different benchmarks.

Overall Application Output Error. We analyze the impact of our mechanism on the overall application output quality in addition to the data quality using the Pin-based [136] instrumentation

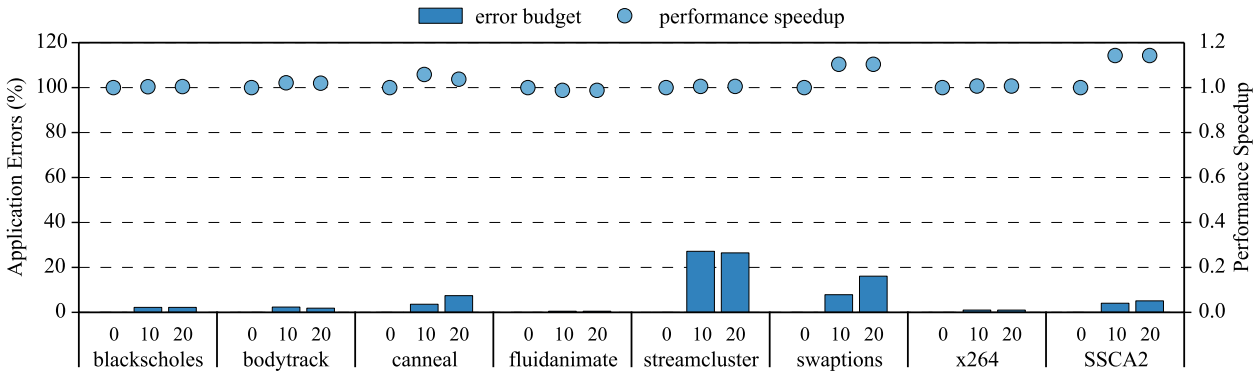


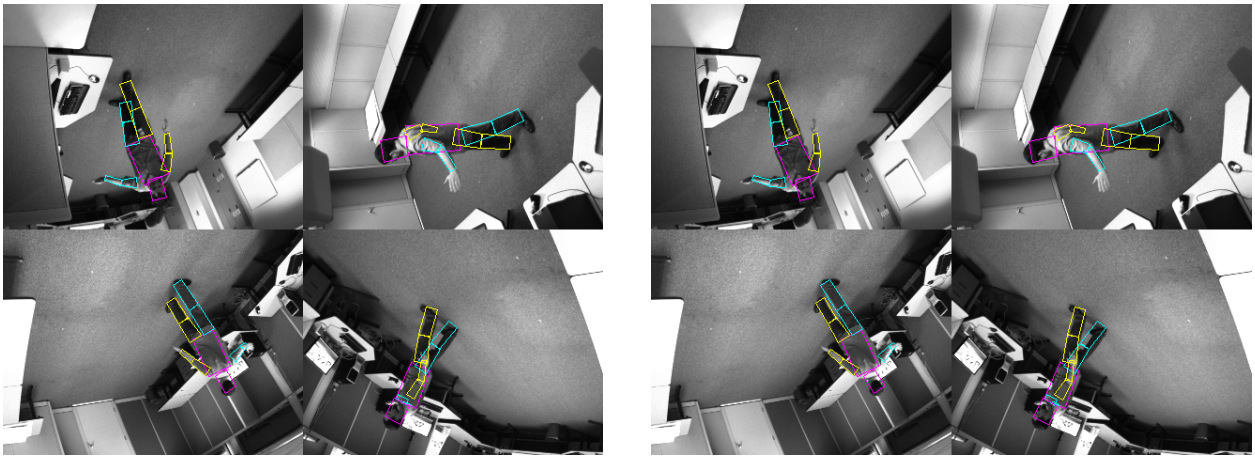
Figure 4.16: Application output accuracy and normalized performance.

framework. We implement our approximate functionalities on top of a coherent cache simulator [137]. We model a system with 16 cores and each core has a 64 KB two-way L1 private data cache of cache line size of 64 Bytes. We emulate packet response from another cache slice whenever a miss happens.

To evaluate the applications output quality, we extend application-specific accuracy metrics based on prior approximate computing research [142, 139, 82, 75]. In addition, we exploit value approximation opportunities in the big data domain by studying a graph processing benchmark *SSCA2*. *SSCA2* calculates the betweenness centrality scores of the nodes in a small world network to identify the key entities. So we evaluate the pair-wise betweenness centrality difference between the approximate output and its precise counterpart for error calculation.

Figure 4.16 shows the applications output accuracy for all benchmarks. With the predefined 10% data error budget, all the benchmarks are well controlled within the error bound except for *streamcluster*. This is because that by approximating the coordinates, the cost between points and centers might deviate from the precise one and lead to mismatch of centers between the approximate version and precise version. As mentioned in previous work, through approximate space exploration or training during compilation, we can improve the accuracy while maintaining the performance benefit [142, 75].

In Figures 4.17, we show the application output of the approximated and precise pair for *bodytrack*. The two figures are very similar and the difference is hardly captured through human vision.



(a) Precise output (b) Approximate output

Figure 4.17: Approximate versus precise output of *bodytrack*.

In this experiment, we allow for a 10% error threshold in the data and observe that the overall output vectors differ by 2.4%.

Figure 4.16 also shows the output accuracy with different error thresholds. Even with 20% error budget, the applications output errors are close to 5% except for *streamcluster* and *swaptions*. With the bounded data error control, APPROX-NOC can achieve high throughput and low latency by exploiting approximate communications while maintaining acceptable output quality.

Overall Application Performance. We also evaluate the impact of APPROX-NOC on the overall system performance. We configure a 64-core CMP connected by an 8x8 mesh network, and run the benchmarks for 100 million instructions with *simmedium* input data size. Figure 4.16 also shows the normalized performance of APPROX-NOC over exact compression for different benchmarks with various error thresholds. We observe that the performance is improved by up to 10% and 14% in *swaptions* and *SSCA2*, respectively. While we see moderate improvements on the rest of the benchmarks. This is because *swaptions* and *SSCA2* have higher degree of sharing in the approximable region of interest in the application code compared to other benchmarks. Higher degree of sharing leads to a significant amount of similar approximable data being transferred across the NoC during the execution of these benchmarks, thereby improving the efficacy of our mechanism on the overall performance.

4.4.5 Power Consumption and Area Overhead

In this section, we evaluate the effect of APPROX-NOC on the network power consumption and area overhead, while taking into consideration the overhead of approximate matching and compression/decompression. The static power consumption does not vary across benchmarks and the static power overhead of all the APPROX-NOC mechanisms is minimal compared to the large baseline static power consumption. Therefore, we show the dynamic power consumption between APPROX-NOC mechanisms and benchmarks in Figure 4.15. The best performing FP-VAXX mechanism reduces the dynamic power consumption on average by 5.4% compared to Baseline and 1.3% compared to FP-COMP. Note that this can be primarily attributed to the reduction in the number of injected flits which compensates for the power overhead of VAXX techniques.

Based on the hardware requirements, we evaluate the area overhead of the APPROX-NOC encoders using CACTI [143] and Verilog based area analysis with 45nm technology node. DI-VAXX incurs 0.0037 mm^2 for each NI (router). Similarly, FP-VAXX requires an overhead of 0.0029 mm^2. The decoder design does not change between the schemes and the overhead is as mentioned in [83].

4.5 Summary

In this chapter, we introduce APPROX-NOC, a hardware data approximation framework for high throughput NoCs in the memory intensive big data era. We present a value based approximate matching technique to use in a plug-and-play fashion with any underlying data compression techniques. We also present low-cost microarchitectural implementations of VAXX with state-of-the-art dictionary-based and frequent pattern-based NoC data compression mechanisms. Our evaluation results show that the best APPROX-NOC mechanism reduces the average packet latency up to 21.4% over state-of-the-art NoC data compression mechanism. Additionally, the best APPROX-NOC mechanism improves throughput up to 60% compared to state-of-the-art compression mechanisms with synthetic workloads. We observe that FP-VAXX achieves higher approximation rate and more performance benefits across the benchmarks while DI-VAXX works better

when there is significant data repetition. On average, the data quality is always above 99% across the benchmarks even with a 10% error threshold since a large portion of the words are within close proximity. The results show promising potentials of approximate communication for High-Throughput data movement.

5. REDUCING DATA MOVEMENT VIA IN-NETWORK COMPUTING [1]

The explosion of data availability and the demand for faster data analysis have led to the emergence of data-intensive applications.These workloads, ranging from neural networks to graph processing, expose compute kernels that operate over myriads of data, exhibiting large memory footprint and low data reuse rate. Significant data movement requirements of these kernels impose heavy stress on modern memory subsystems and communication fabrics. To mitigate the worsening gap between high CPU computation density and deficient memory bandwidth, solutions like memory networks and near-data processing designs are being architected to improve system performance substantially. In this chapter, we examine the idea of mapping compute kernels to the memory network so as to leverage in-network computing in data-flow style, by means of near-data processing. An in-network compute architecture, *Active-Routing*, is proposed to enable computation on the way for near-data processing by exploiting patterns of aggregation over intermediate results of arithmetic operators.

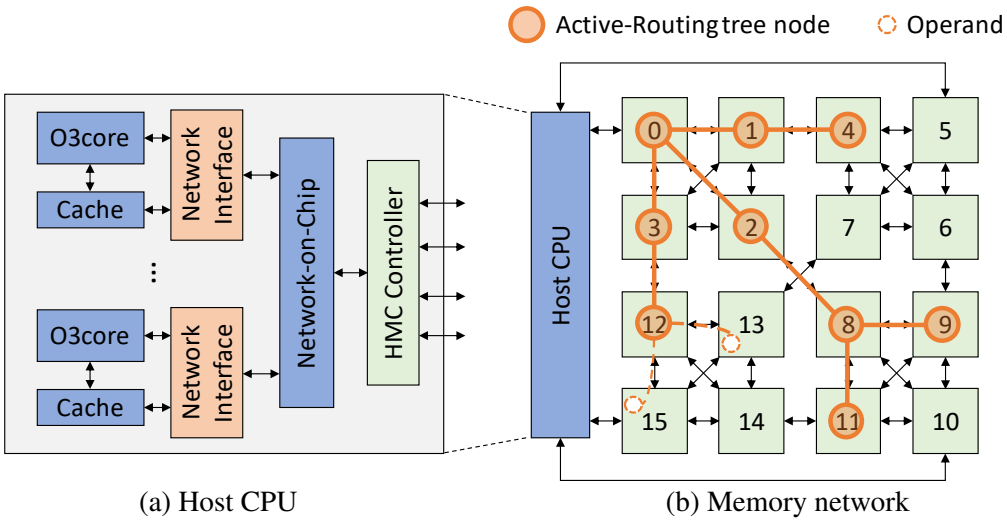
5.1 Active-Routing Architecture

In this section, we first illustrate *Active-Routing* by walking through an example. Then we describe its three-phase packet processing procedure. Lastly, we categorize the memory access patterns of the data to be processed and propose enhancements to reduce offloading overhead by leveraging their characteristics.

5.1.1 Architectural Overview

Figure 5.1 presents the system configuration, where host processors are connected to a memory network formed by chaining hybrid memory cubes (HMCs). In this system, we show an example of *Active-Routing* in the memory network that computes `sum += A[i]×B[i]` over a large loop with

[1]Reprinted with permission from “Active-Routing: Compute on the Way for Near-Data Processing", by Jiayi Huang, Ramprakash Reddy Puli, Pritam Majumder, Sungkeun Kim, Rahul Boyapati, Ki Hwan Yum and Eun Jung Kim, 2019. 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 674-686, doi: 10.1109/HPCA.2019.00018, Copyright © 2019 IEEE.



(a) Host CPU (b) Memory network

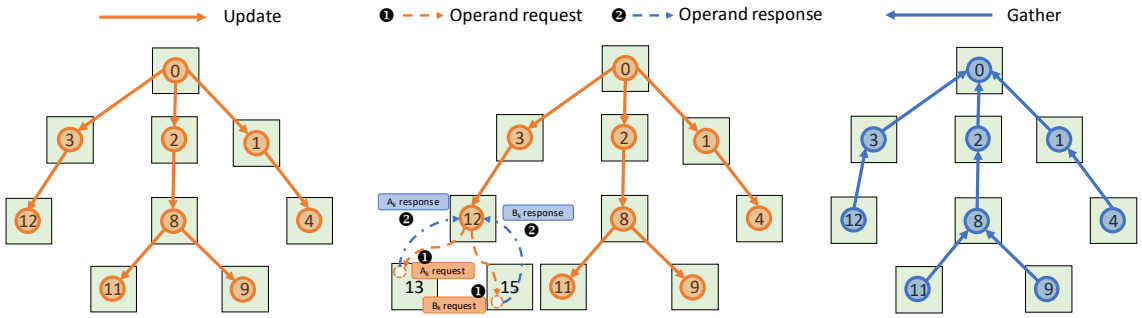
Figure 5.1: System configuration with (a) a host CPU connected to (b) a Memory network with an *Active-Routing* example.

loop index `i`. Each computation of `A[i]×B[i]` is offloaded from the host CPU to the memory network as an **Update** packet. **Update** packets are scheduled for computation at the memory cubes near to the operand locations to compute the partial sum through near-data processing (NDP). Following **Update** offloading, **Gather** packets are sent to collect the partial results from each cube, and reduce them in the network routers on the way back to the host.

The three phases of *Active-Routing* as it progresses in the timeline for this example is shown in Figure 5.2, namely *ARTree Construction*, *Update* and *Gather Phase*.

- While offloading **Update** packets, an *Active-Routing Tree (ARTree)* is being constructed along the packets' paths to the scheduled compute memory cubes. For instance, in Figure 5.1b, an **Update** packet is sent from the CPU through memory cube 0 to cube 8. It records the tree nodes and builds a tree branch along its path to cube 8. **Update** packets scheduled at different cubes construct different branches. These branches altogether form an *ARTree*, as abstracted in Figure 5.2a.

- The offloaded computations drive near-data processing during the *Update Phase* as shown



(a) Tree Construction (b) Update Phase (c) Gather Phase

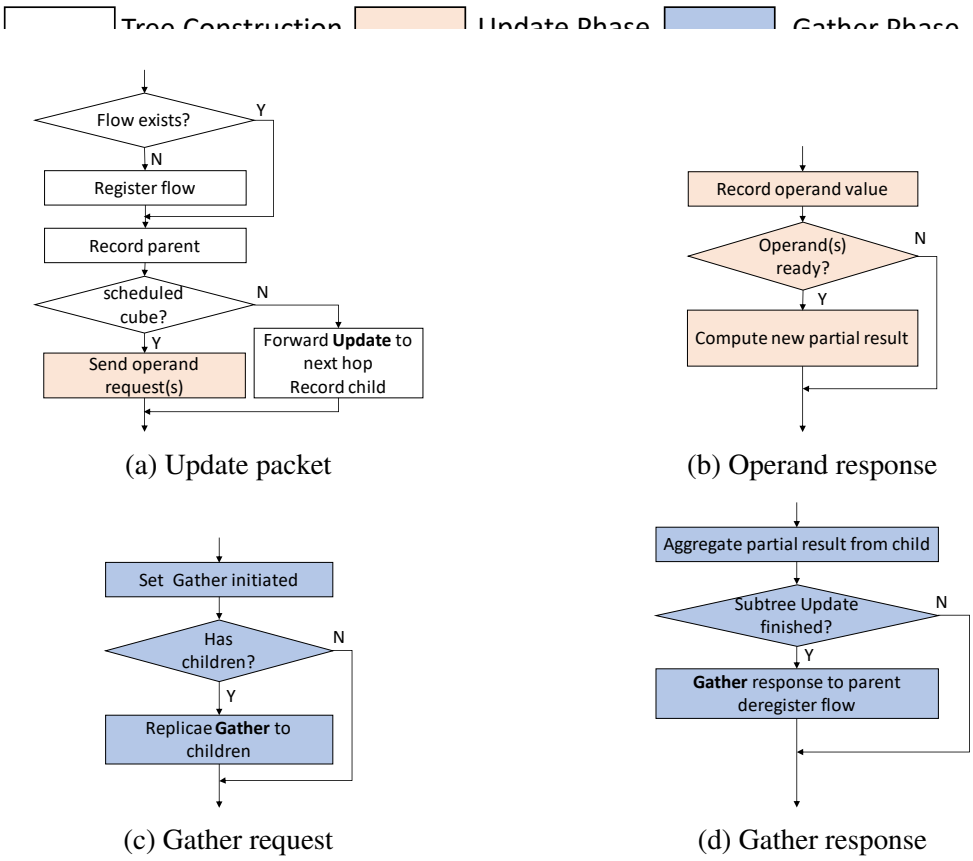
Figure 5.2: Active-Routing consists of three phases: (a) Active-Routing Tree Construction on-the-fly; (b) near-data processing in Update Phase; and (c) network aggregation along the Active-Routing Tree in Gather Phase.

in Figure 5.2b. Each operation `A[i]×B[i]` requests its source operands `A[i]` and `B[i]` to finish the computation and updates the partial sum in the scheduled cube. Figure 5.2b also depicts a case where two operands reside in different cubes. In such cases, the **Update** packet will be sent to the scheduled compute point that is the last common cube (cube 12) on the minimum routes for both operands: ❶ it replicates to issue two operand requests for A_k and B_k to the resident memory cube 13 and cube 15, respectively. ❷ Then, the two operand responses are sent to cube 12 to complete the computation. All the intermediate results in the same compute cube are reduced to a partial sum in the cube during this phase.

- Figure 5.2c shows the *Gather Phase* when **Gather** packet is issued following all the **Update** packets. It is replicated from the root to each node of the *ARTree*. Then **Gathers** at leaf nodes initiate network reduction of partial sums computed in the previous phase in a dataflow manner to the root along the *ARTree*.

5.1.1.1 Three-Phase Packet Processing

In general, *Active-Routing* maps a compute kernel in the memory network to optimize reduction over intermediate results of arithmetic operators. We name such a mapping as an *Active-Routing flow*. A unique identification (*flow ID*) is assigned for each *flow* and its corresponding *ARTree*.



(a) Update packet

(b) Operand response

(c) Gather request

(d) Gather response

Figure 5.3: Active packet processing flow chart for (a) update packet, (b) operand response packet, (c) gather request packet and (d) gather response packet.

Each *flow* involves a three-phase packet processing procedure as shown in Figure 5.3.

ARTree Construction. For each *flow*, an *ARTree* is built dynamically while processing its **Update** packets, as shown in Figure 5.3a. Upon receiving an **Update** packet, each cube registers its *flow ID*. If the **Update** packet is not scheduled to compute at the current cube, the packet is forwarded to its child based on its routing to the scheduled compute cube. Therefore, an *ARTree* is built by recording parent and children information at each node.

Update Phase. This phase starts in concurrent with the *ARTree* construction phase. It involves processing of **Update** packets, and operand request/response packets as shown in Figures 5.3a and 5.3b. While processing **Update** packets, operand requests are sent out to the memory from the scheduled compute node. When the operand responses arrive, the arithmetic operations are

scheduled to compute the partial aggregation result.

Gather Phase. Figures 5.3c and 5.3d show the packet processing in *Gather Phase* to commit *Active-Routing flow*. This phase has one forward pass to multicast the **Gather** requests from the root to leaf nodes, and a backward pass to reduce the results from leaf nodes to the root node. Once a node's subtree finishes *Update Phase*, it replies to its parent and deallocates the *flow* record. A parent receives **Gather** responses from all its children to indicate the completion of their *Update Phase*. When the root node finishes its own *Update Phase* and receives all its children's **Gather** responses, it commits the *flow*.

5.1.1.2 Memory Access Patterns

Instruction offloading and operand fetching incur overhead using packet switching due to meta-data in the packet header and packet internal fragmentation. Memory access patterns of operand fetching can be exploited to amortize the overhead by offloading multiple operations at a time. *Active-Routing* aims to optimize reduction on massive intermediate results of arithmetic operators, such as $\texttt{sum} = \sum_{i=1}^{n} \star\texttt{A}_\texttt{i} \times \star\texttt{B}_\texttt{i}$, where $\texttt{A}_\texttt{i}$ and $\texttt{B}_\texttt{i}$ store the operand addresses. Memory access patterns of operands can be regular when vector `A` stores array addresses. When it stores addresses of graph nodes or sparse matrix elements, the access pattern tends to be irregular. Therefore, the combined memory access pattern for the two operands can be categorized into three groups: *regular-regular*, *regular-irregular*, and *irregular-irregular*. Based on these three categories, we propose three different ways to leverage data locality.

For *regular-regular* access pattern, we offload the computation in cache block granularity as vector processing. While for *regular-irregular* access pattern, we fetch the irregular data and send them to regular data resident location for processing similar to PEI [18]. The above two methods maximize the locality benefit and reduce the memory accesses. For *irregular-irregular* memory access pattern, we simply fetch single operand pairs to the scheduled compute node as scalar operations. *Active-Routing* can cooperate with previous study [94] to further optimize *irregular-irregular* access pattern, which we leave for future work.

5.2 Implementation

In this section, we describe the programming interface and instruction set architecture (ISA) extension. Then we introduce the hardware components that work in synergy to realize *Active-Routing*, including Network Interface (NI) support and *Active-Routing Engine (ARE)*. Lastly, we discuss system integrity considerations and several enhancements in *Active-Routing*.

5.2.1 Programming Interface and ISA Extension

We provide simple programming interfaces (**Update** and **Gather**) to translate the program semantics into extended instructions. The ISA extensions are used to communicate with Network Interface to offload computations to the memory network for *Active-Routing* processing.

```
Update(void *src1, void *src2, void *target, int op);
UpdateRR(void *src1, void *src2, void *target, int op);
UpdateRI(void *src1, void *src2[], void *target, int op);
UpdateII(void *src1, void *src2, void *target, int op);
Gather(void *target, int num_threads);
```

The above **Update** and **Gather** APIs are defined to offload *Active-Routing flows*. The **Update** API carries two source memory addresses of an arithmetic operation. The postfix **RR**, **RI** and **II** of **Update** API are used for three memory access pattern categories, respectively. The `op` parameter is the opcode indicating the type of arithmetic and reduction operation (e.g. floating point multiply-and-accumulate). The `target` field in the APIs is the address of the reduced variable, which is hashed to a unique identification for each *flow*. In the **Gather** API, the `num_threads` parameter indicates the number of threads working on the *flow*. It is used for an implicit barrier at the root of *ARTree* to guarantee all the **Updates** have been initiated. We generalize the **Update** API with opcode `op` to support simple operations such as assignment. These APIs are translated to extended intrinsic instructions by the compiler. During execution, instruction fields are written to a set of dedicated registers in the Network Interface (NI). NI can assemble this information into an **Update** or a **Gather** packet and send it to the memory network.

Figure 5.4 shows the baseline and *Active-Routing* implementations of the thread worker pseu-

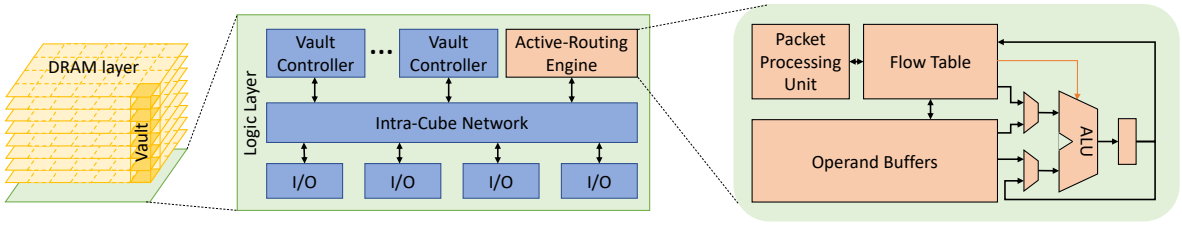
```
// baseline implementation
global diff = 0.0;
local loc_diff = 0.0;
for (v: v_start to v_end) {
  loc_diff += abs(v.next_pagerank - v.pagerank);
  v.pagerank = v.next_pagerank;
  v.next_pagerank = 0.15 / graph.num_vertices;
}
atomic diff += loc_diff;


// active optimization
global diff = 0.0;
local temp = 0.15 / graph.num_vertices;
for (v: v_start to v_end) {
  Update(&v.next_pagerank, &v.pagerank, &diff, abs);
  Update(&v.next_pagerank, nil, &v.pagerank, mov);
  Update(temp, nil, &v.next_pagerank, const_assign);
}
Gather(&diff, num_threads);
```

Figure 5.4: Pseudocode of thread worker for parallel *pagerank*.

docode of *pagerank* calculation kernel. In the baseline implementation, the **atomic** update for `diff` needs to fetch the `pagerank` and `next_pagerank` values for each vertex in the graph. This consumes a large amount of bandwidth due to irregular graph access patterns. It also needs to reduce the `diff` value atomically for each thread, which causes high overhead and limits thread scaling. In contrast, *Active-Routing* allows updates of `diff` near the data location to save bandwidth. In addition, the **Gather** requests from all the threads of same *flow* are synchronized at the root of *ARTree* as an implicit barrier. Then reduction is initiated along the *ARTree*. Note that the read-write dependencies between instructions are enforced as same as normal instructions. The read-write dependencies can be tracked and resolved by memory controllers similar to read-write requests dependencies handling with simple extension.

5.2.2 Network Interface

Programming interfaces are used in application for *Active-Routing* offloading. Compiler takes the API and translates it into extended instructions. Extended instructions are assembled to packets



(a) HMC logic layer for Active-Routing engine.

64-bit	6-bit	64-bit	64-bit	64-bit	2-bit	4-bit	1-bit
flowID	opcode	result	req_counter	resp_counter	parent	children flags	Gflag

(b) Flow table entry

64-bit	6-bit	64-bit	1-bit	64-bit	1-bit
flowID	opcode	operand1	op1_ready	operand2	op2_ready

(c) Operand buffer entry

Figure 5.5: Active-Routing microarchitecture: (a) engine implementation in HMC logic layer with (b) flow table entry and (c) operand buffer entry.

and offloaded to the memory network for processing. This functionality can be added to Network Interface (NI), connecting core and on-chip network, with marginal change. In NI, we add dedicated registers that can be written by extended instructions to convey the opcode and operand information. NI reads these registers to assemble an **Update** or a **Gather** packet and issue it into the memory network.

5.2.3 Active-Routing Engine

The *Active-Routing* facilities are implemented in *Active-Routing Engine (ARE)* on the HMC logic layer as shown in Figure 5.5a. It is integrated as an attached module to the router switch. ARE consists of 1) a packet processing unit to process and generate packets, 2) a *flow table* to keep track of *Active-Routing flows*, 3) a pool of *operand buffers* to store operands, 4) an ALU for computation.

5.2.3.1 Packet Processing Unit

Packet processing unit is responsible for decoding the **Update** and **Gather** packets and schedule actions correspondingly as shown in Figure 5.3. It can generate operand request packets to fetch the data and **Gather** response to commit the partial result to its parent.

5.2.3.2 Flow Table

Flow table keeps track of both the structure and states information of each *flow*. Figure 5.5b shows a *flow* entry. Each entry in the table includes a tree node record that maintains the structure of the tree by keeping a unique `flow ID`, an `opcode` for computation, and its `parent` and `children`. It also keeps the *flow*'s state, including the partial `result`, the `req_count` and `rep_count`, as well as `Gflag`. The `req_count` and `rep_count` counters are used to keep the number of issued requests and committed operations. A *Update Phase* is considered finished when these two counter values are the same. The `Gflag` is set by a **Gather** request indicating that the flow can start reduction once *Update Phase* completes.

5.2.3.3 Operand Buffers

Update packets are processed to generate request(s) to fetch operands and perform the computation with the response. Operand buffer is used as a temporary storage for the operands waiting to be processed, therefore maintaining the pending **Update** operations. We make a pool of operand buffers shared by different flows so as to improve the throughput and reduce the overhead. An operand buffer entry is reserved before sending out operand request(s) since co-existing *flows* can easily cause deadlock due to wait-and-hold condition, especially for two-operand operations. Figure 5.5c shows an operand buffer entry, which keeps the `flowID` and `opcode`, two `operand` fields and two `ready` flags to indicate the operand's availability. To reduce the operand buffer access time, we use a free and a ready queue to keep IDs of free and ready operand entries, respectively, for ease of direct lookup.

5.2.3.4 ALU

A light-weight ALU is implemented in ARE to compute arithmetic operations. *Active-Routing* supports various operations on different data types, including reduction operations such as sum, xor, and, min, and max, as well as multiply-accumulate on floating point and integer data. We plan to generalize our approach and implement more powerful logics to support complex program accelerations.

5.2.3.5 Putting It All Together

Upon receiving an **Update** request packet, ARE processes it in the Packet Processing Unit. If the corresponding *flow* has not yet registered in the *flow table*, an entry is allocated for the new *flow*. The flow is registered and fields are initialized by recording the *flow ID* and the packet's previous hop as parent in the entry. If the packet is not scheduled for the current cube, it is forwarded based on the computed route to next hop, which is recorded in the `children flags`. Otherwise, the `req_count` is incremented and an operand buffer entry is allocated from the free queue. Meanwhile, operand request packets embedding the operand address and buffer entry ID are also generated. If all operand buffer entries are busy, the Packet Processing Unit is stalled until an operand buffer entry is available. When a response for the operand arrives, the corresponding operand buffer entry is updated. If operands are ready, the operand entry ID is pushed to the ready queue for processing. ALU is directed by the ready queue for computation. After the computation finishes, the `resp_count` is incremented and `result` is updated in the corresponding *flow* entry. The operand buffer is deallocated for reuse by pushing back its ID to free queue. While processing **Gather** request packets, the `Gflag` of the corresponding flow table entry is set to initiate *Gather Phase* after the completion of the *Update Phase* of the subtree. If the cube has children cubes, the packet is replicated and multicast to its children. Upon receiving a **Gather** response from a child for partial result update, its corresponding child field is cleared. Note that every time the `result` is updated by either computation in current cube or **Gather** packet from a child, if `Gflag` is set and `children flags` are cleared, a **Gather** response is generated to send the partial result back to its

parent and release the flow table entry.

5.2.4 Integrity Considerations

There are two important design considerations to seamlessly integrate *Active-Routing* into current computer systems: virtual memory support and cache coherence.

5.2.4.1 Virtual Memory

Since *Active-Routing* is implemented by ISA extension, the offloaded instructions are treated as extended *active* loads/stores. Therefore, they can perform the same virtual to physical address translation as normal load/store instructions. With this design principle, we can avoid overhead for address translation units in the directories, or memory.

5.2.4.2 Cache Coherence

To offload instructions for *Active-Routing* optimization, it should ensure that the offloaded *flow* is using the up-to-date data in memory. A naïve way is allocating uncacheable memory for the data that may be used in the optimization. However, it may hurt the performance in other program execution phases which can use the deep cache hierarchy to exploit locality. To work around with coherence, offloaded packets are first sent to the directory based on their address, and query for back-invalidation if data is cached on-chip similar to [90]. Then it will be issued to the memory for *Active-Routing* processing. For two-operand computations, if the two addresses belong to different directories, two separate requests are created for back-invalidation, after which they are merged at the predetermined offloading memory port. Since **Update** packets are issued in parallel, the back-invalidation overhead is amortized across massive concurrent packets. We observe that back-invalidation rarely happens in our experiments.

5.2.5 Enhancements in Active-Routing

We observe two critical points that have significant impact on the *Active-Routing* performance: (1) the decision for choosing the root of a tree affects the network congestion, and (2) the overhead of offloading computations widely varies with the change in its granularity. To improve *Active-*

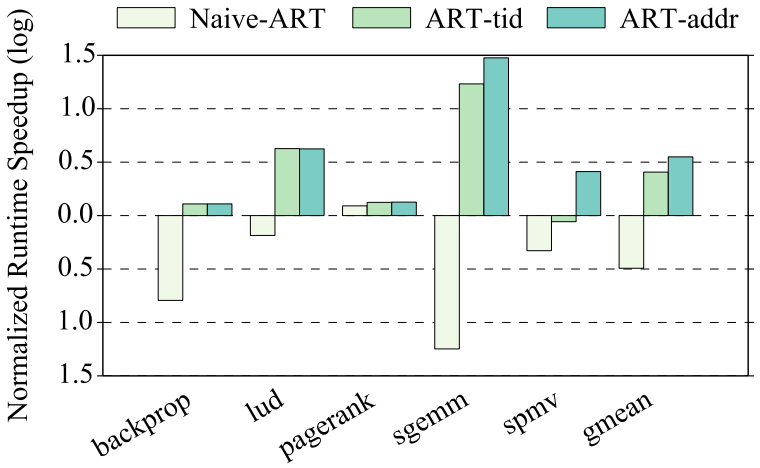


Figure 5.6: Runtime speedup of ART variants over HMC Baseline.

Routing performance further, we address each of these points as follows.

Since the computations are offloaded from the host CPU through the memory ports, we naturally consider the cubes that are attached to the four channel ports as root node candidates. We start with a static approach that we always assign the root node to be cube 0. In order to balance the load better in the network, we propose two enhancement techniques that can consider all four corner cubes as roots and are able to create multiple trees for one *flow*. The first one uses thread ID to interleave the candidate cubes so as to balance the trees rooted from four corners among multi-thread applications, named as ART-tid. Since the scheduling is oblivious to the data location, it can create deep trees and lead to more hop traversals for **Update** request packets. Another enhancement technique takes the operand addresses into account and sends the **Update** packet through the port nearest to its destination. This creates shallow trees with respect to ART-tid, we name it as ART-addr. Since these two schemes can create multiple *ARTrees* for one *flow*, the extended HMC memory controllers that manage the trees are coordinated to merge the subflows at the end of *Gather Phase*. On the contrary, Naïve-ART constructs only one *ARTree* for each flow.

To reduce offloading overhead and the number of memory accesses, we adapt the offloading granularity, to exploit the data locality of different memory access patterns discussed in Section 5.1.1.2. This optimization is applied to both ART-tid and ART-addr, whereas Naïve-ART does

not consider granularity, which simply offloads every single operand pair without considering data locality. This Naïve-ART may experience contention in operand buffer resources, and network contention in addition to high offloading overhead due to static manner for tree construction and simple offloading.

Figure 5.6 shows the improvement impacts of enhancements over Naïve-ART. We take the log scale of speedup that is normalized to HMC conventional system baseline (not shown). It shows that with naïve way of static tree formation and offloading, Naïve-ART is worse than HMC baseline, especially when there is some locality in accesses. In contrast, by constructing the trees dynamically and exploiting the memory access patterns, we can achieve better performance. In the following sections, we only present ART-tid and ART-addr for detail analysis.

5.3 Methodology

5.3.1 System Modeling and Configuration

We use an execution-driven simulator McSimA+ [144] with detailed microarchitecture models as the backend for cores and cache hierarchy. For HMC memory modeling, we integrated a cycle-accurate simulator CasHMC [145] with McSimA+ to replace its memory system. We leveraged McSimA+'s Pin [146] based front end to implement *Active-Routing* instruction extensions. The microarchitectural behaviors of *Active-Routing* were implemented on the crossbar switch in the HMC logic layer.

For power and latency modeling, we use CACTI [143] for on-chip cache power estimation, assume 5pJ/bit for each hop in memory network [147], 12 pJ/bit for HMC memory access. We implemented the *ARE* in verilog and synthesized it using TSMC 45 nm library. The multiplication takes the longest time, which is 6.61 ns, and the operand buffer takes 0.59 ns access time. As we use 1250 MHz for *ARE* and pipeline the arithmetic operations, it takes 9 cycles for each mult and 1 cycle for buffer access. At full load, *ARE*'s ALU can compute 1 FLOP/cycle. The area and power estimation is 0.02 mm^2 and 17.8 mW for ALU, 0.026 mm^2 and 16.9 mW for operand buffer, 0.05 mm^2 and 33.2 mW for Flow Table.

We configured the host CPU as a CMP with network-on-chip and two level cache hierarchy with MESI coherence protocol. The 16 off-chip HMCs were connected to form a Dragonfly topology [95]. The system configuration evaluated in this work is shown in Figure 5.1 and described in Table 5.1.

Table 5.1: *Active-Routing* System Configuration

Parameter		**Configuration**
CPU	Core	16 OoO cores @ 2GHz issue/commit width: 4, ROB: 128
	L1I/D Cache	Private, 32KB, 4 way
	L2 Cache	S-NUCA 16MB, 16 way, MESI
	NoC	4x4 mesh, 4 MC at 4 corners
Memory	DRAM Timing	$t_{CK} = 0.8$ ns, $t_{RAS} = 21.6$ ns, $t_{RCD} = 10.2$ ns $t_{CAS} = 9.9$ ns, $t_{WR} = 8$ ns, $t_{RP} = 7.7$ ns
	HMC	4GB/cube, 4 layers 32 vaults, 8 banks/vault
	HMC Network	16 cube DragonFly, 4 controllers Minimal routing, virtual cut-through 16 lanes link, 12.5 Gbps/lane CrossbarSwitch clock @ 1250 MHz
Active-Routing Engine	Flow Table	16 flow entries
	Operand Buffer	128 buffer entries
	Processing Element	1250 MHz clock frequency An arithmetic logic unit

5.3.2 Workloads

Active-Routing targets applications that have abundant reduction on data processing operations such as multiply-accumulate or pure reduction operations over a large memory footprint. We studied five kernels from several benchmark suites. These kernels are widely used in diverse application domains such as scientific computing, graph analytics, language modeling and deep learning. We also developed four data-intensive microbenchmarks for case study.

Table 5.2: Workloads

Workloads		Optimization Region	Input Data Size
Benchmark	*backprop* [149]	activation calculation in feedforward pass	2097152 hidden units
	lud [149]	upper and lower triangular matrix decomposition	4096 matrix dimension
	pagerank [150]	ranking score calculation	web-Google graph [141]
	sgemm [151]	matrix multiplication	4096x4096 matrix
	spmv [151]	matrix-vector multiplication loop	4096 matrix dimension (0.7 sparsity)
Micro-benchmark	*reduce*	sum reduction over a sequential vector	6400K dimension
	rand_reduce	sum reduction over random elements	6400K elements
	mac	multipy-and-accumulate over two sequential vectors	two vectors with 6400K dimension
	rand_mac	multiply-and-accumulate over two random element lists	two lists with 6400K elements

In order to support execution with McSimA+ frontend, all the applications were re-implemented using the Pthread library. We chose sufficient large input data so as to stress the last level cache and memory as well as to account for reasonable simulation time. The working set sizes varied from 80 MB to 0.5 GB. The memory requirements of these kernels used in various applications tend to grow significantly larger as data scales [148]. The workloads and applied optimization region as well as input data are summarized in Table 5.2.

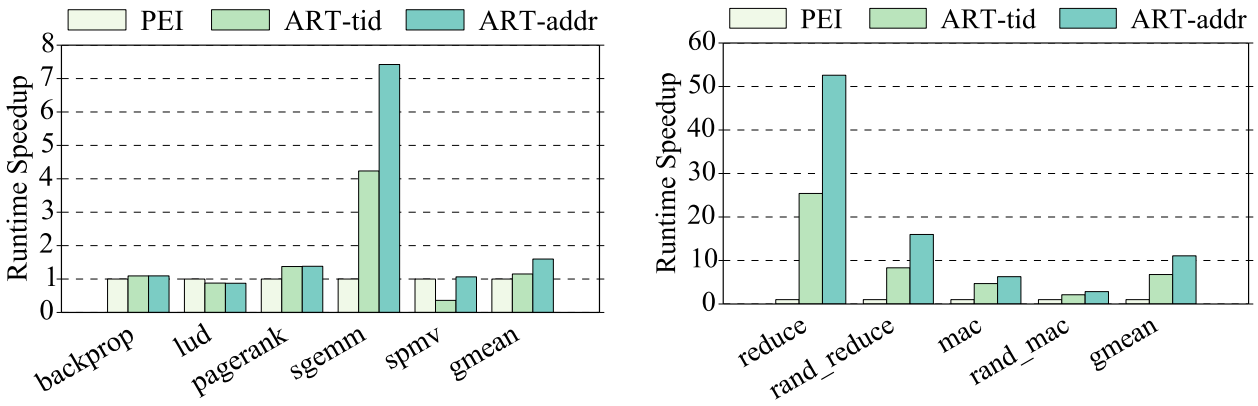
5.4 Evaluation

In this section, we evaluate ART-tid and ART-addr with respect to PEI [90], implemented by adding a computation unit at each vault controller supporting PEIs. It can compute a dot product of two 4D vectors in a cycle, one of the vector operands (either regular or irregular) are brought to cache and sent to the memory location of the other half (should be regular) for processing in memory. We first present performance evaluation followed by power and energy analysis. Then we show the potential of dynamic offloading through a case study.

5.4.1 Performance

5.4.1.1 Speedup

Figures 5.7a and 5.7b show the execution time speedup of benchmarks and microbenchmarks, respectively. Both ART-tid/addr schemes create multiple trees from all memory ports for massive



(a) Benchmarks

(b) Microbenchmarks

Figure 5.7: Runtime speedup over PEI.

flows in the benchmarks. The results show more than 6% performance improvement across various applications with respect to PEI except *lud*. Specifically, ART-addr improves *sgemm*, a dense matrix multiplication kernel up to 7× speedup. In *sgemm*, almost all the execution time is spent in matrix multiplication. During the kernel execution, PEI needs to fetch one of the source matrices and also update the target matrix, causing read and write contention on the limited cache, which results in cache thrashing. In contrast, ART has no contention between source matrices and target matrix since both source matrices are fetched and processed in memory, thereby outperforming PEI significantly. In geomean, ART-tid and ART-addr improve performance by 15% and 60% over PEI, respectively. For *lud*, PEI performs slightly better than both ART-tid and ART-addr. In case of *spmv*, PEI outperforms ART-tid but performs worse than ART-addr. This is because the computation distribution in these two applications is not balanced, which causes contentions in compute/buffer resources.

Note that the PEI implementation is optimistic since we have no limit on operand buffers. For *spmv*, ART-addr is better than ART-tid due to more balanced work distribution, which will be discussed in short. In microbenchmarks, the whole execution is the region of interest for optimization. Both ART-tid/addr alternatives work well across all microbenchmarks. Compared with PEI, ART-tid/addr achieves 7×/10× speedup, respectively.

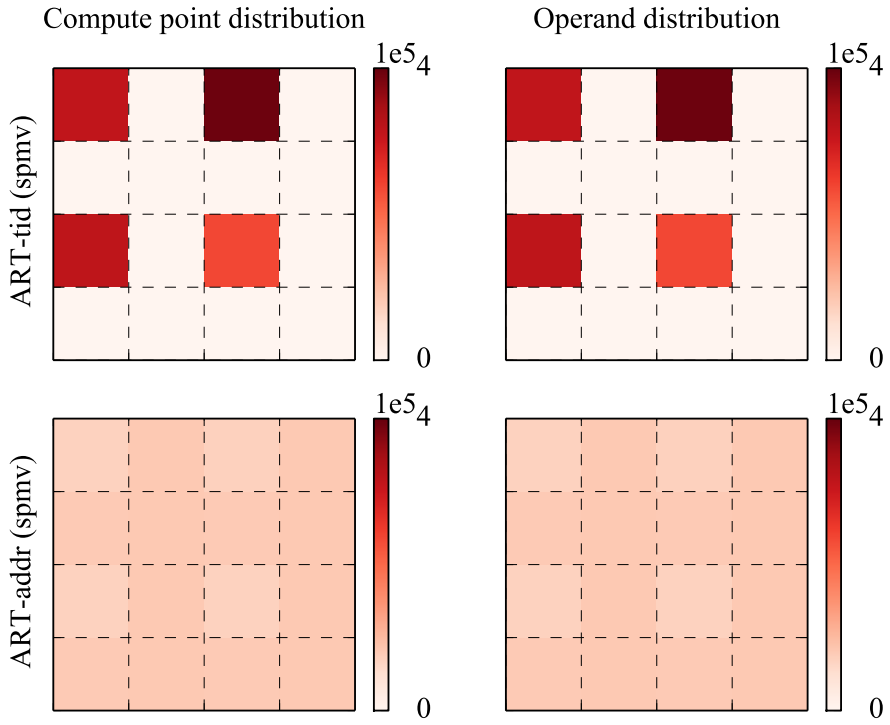


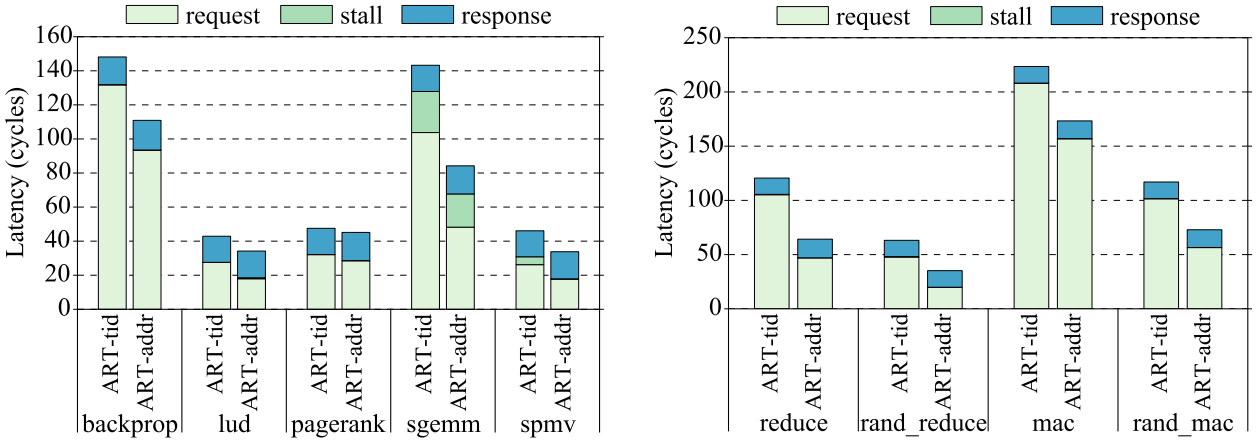
Figure 5.8: SPMV compute point and operand distribution.

Figure 5.8 shows a heatmap of *spmv* for ART-tid and ART-addr. In the heatmap, darker colors are used for denoting higher numbers of event occurrences. Each big square depicts the whole memory network and each small square block represents one cube in the memory network. In ART-addr, the work is evenly scheduled in each cube which can have better resource utilization. While in ART-tid, computations are centered in a few cubes, leading to compute/operand resources contention and less parallelism[2].

To evaluate scalability, we also ran experiments for *mac* on a 64-cube dragonfly memory network. With the same problem size for strong scalability, ART-tid and ART-addr achieve 4.6× and 6.3× speedup compared to PEI on the 16-cube memory network, whereas on the 64-cube memory network, ART-tid and ART-addr outperform PEI for 4.7× and 6.4× improvements, respectively. As we scale the problem size four times as the memory capacity scales for weak scalability, ART-tid and ART-addr improve the performance over PEI by 4.6× and 7.1×, respectively. When com-

[2]The operand distribution are different due to the dynamic memory allocation.

paring each technique's performance on the two different memory networks for the same problem size, PEI incurs 2% performance degradation on the 64-cube network compared to its performance on the 16-cube memory network. Whereas both ART-tid and ART-addr have less than 0.1% performance difference, either better or worse, on the two memory networks. Since PEI has more on/off chip data transfer than ART, it is more sensitive to the increased memory access latency due to higher average network latency in larger scale memory networks. On the contrary, ART benefits from both memory parallelism and network concurrency, therefore it tends to scale better for larger memory networks.



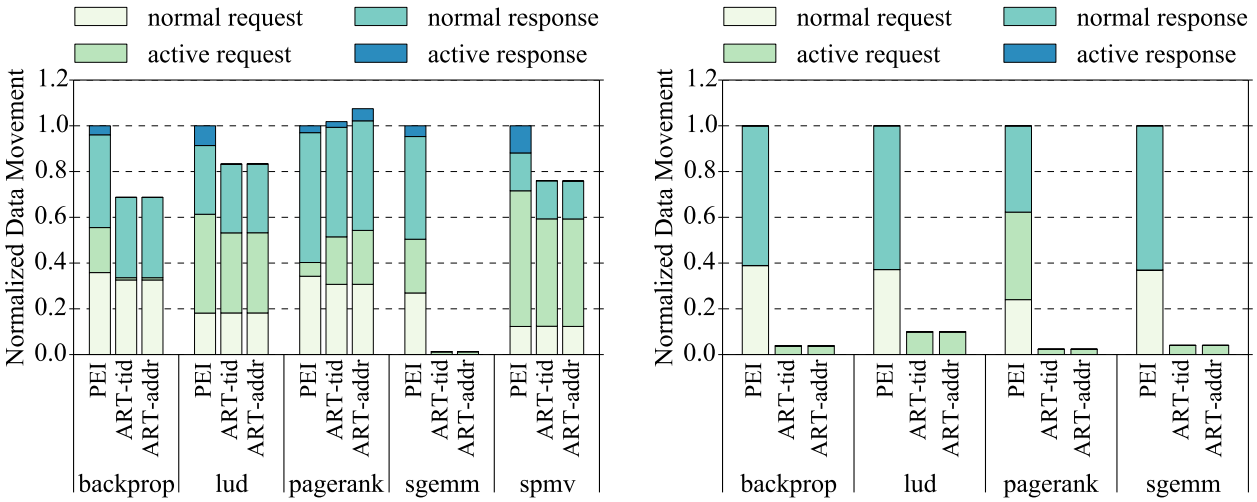
(a) Benchmarks (b) Microbenchmarks

Figure 5.9: Update round-trip latency breakdown into request, stall and response latency.

5.4.1.2 Update Offloading Roundtrip Latency

In Figure 5.9, round-trip latency is broken into request, stall and response to understand the contribution of different communication components for **Update** offloading. As expected, the total latency is inversely proportional to the performance shown in Figure 5.7. In general, ART-tid and ART-addr dynamically distribute the **Updates** across all available ports for tree construction. The ART-tid/addr schemes can balance the load evenly and utilize the memory network resources more

efficiently. Compared to ART-tid, ART-addr has lower round-trip latency across all benchmarks. ART-tid constructs trees by interleaving memory ports using thread IDs. Therefore, the tree root is not necessarily close to the directory where **Update** packets check for coherence. In contrast, ART-addr distributes **Updates** based on addresses, which makes the tree root physically close to the directory, thereby incurring less request latency. The stalls are mostly due to queuing in HMC controllers.

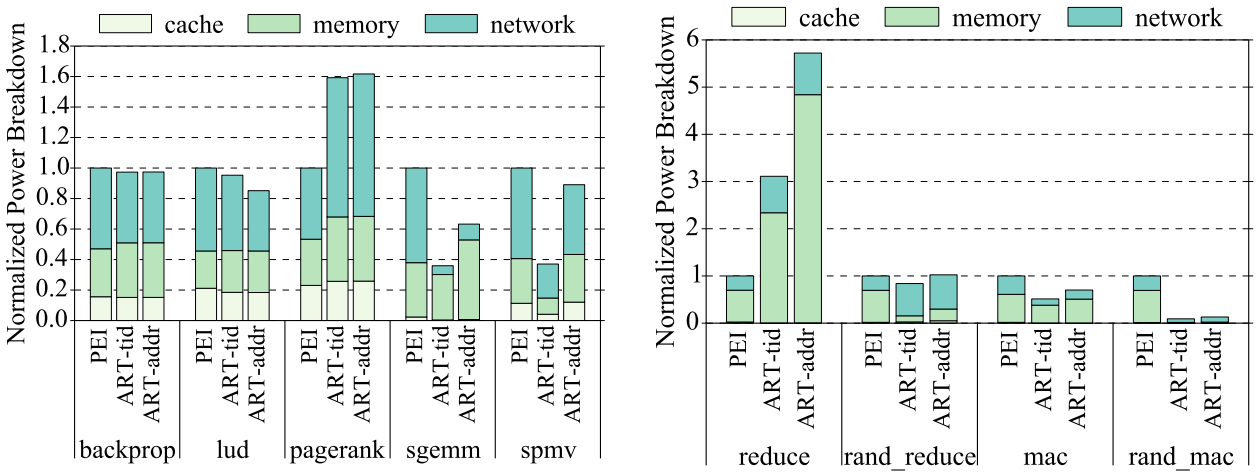


(a) Benchmarks (b) Microbenchmarks

Figure 5.10: On/off-chip data movement normalized to PEI.

5.4.1.3 Data Movement

We evaluate data movement as the data size transferred between the host processor and memory network. The data movement breakdowns for normal data and active data transfer are shown in Figure 5.10. For most applications, ART-tid/addr can reduce the memory requests fetching the data, mostly source operands, compared to PEI. In *pagerank*, the region of interest for optimization is the code segment that has reduction on large amounts of data processing tasks. In the benchmarks, only parts of the whole parallel phase are our optimization targets. The other phases still require data movement. Another overhead comes from massive fine-grained offloading in this



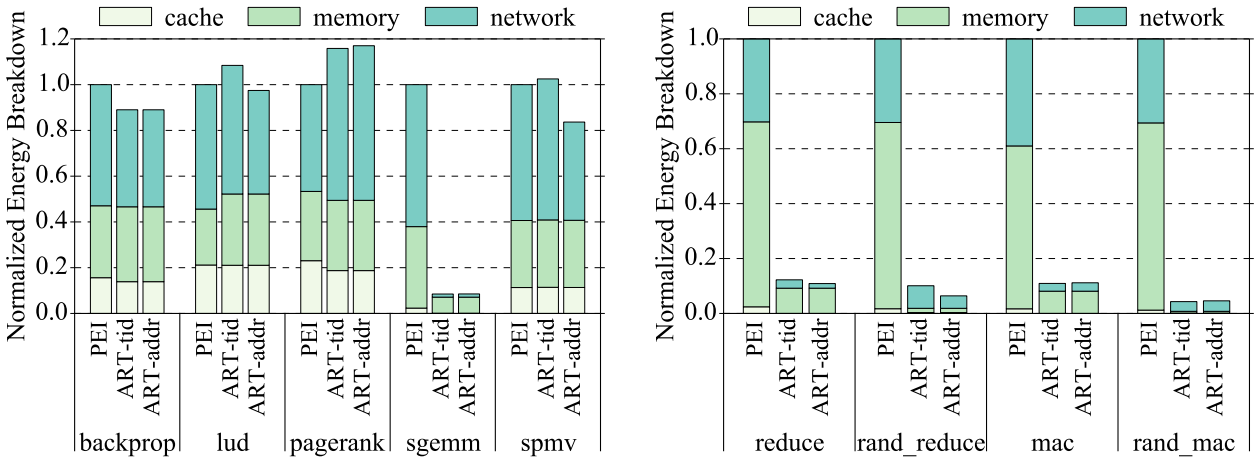
(a) Benchmarks

(b) Microbenchmarks

Figure 5.11: Normalized power consumption over PEI.

benchmark due to the irregular memory access pattern. Further preprocessing on the data [94] can solve this problem to gain performance and reduce data movement further.

In the microbenchmarks, the whole parallel phase can be optimized and hence the data movement decreases significantly. In *reduce*, the majority of its execution time is spent on summing up all the elements of a large array as it accesses the array elements sequentially. Similarly, *mac* operates multiply-and-accumulate over two large vectors. Both of them exhibit very good spatial locality in their memory accesses, which is exploited in cache-block grained offloading for vector processing. However in PEI, it needs to bring part of the data on chip and offload it with the instruction, causing data movements. For *rand_reduce* and *rand_mac*, ART-tid/addr have more data movements compared to the sequential accesses due to offloading overhead. Since PEI still needs to bring the data for random multiplication on chip before atomic write, it incurs more data movement.



(a) Benchmarks (b) Microbenchmarks

Figure 5.12: Normalized energy consumption over PEI.

5.4.2 Power and Energy

5.4.2.1 Power Consumption

We present the power consumption breakdowns into cache, memory and memory network in Figure 5.11. We observed that ART-tid/addr consume similar memory power and less network power than PEI except for *pagerank*. In ART-tid/addr, data is fetched from memory and communicate in the network. However in PEI, part of the operands need to be brought across the network to on-chip cache and be sent with the offloaded instruction, leading to cache contention even cache thrashing. For example, *sgemm* has cache contention between reading of large source matrix and writing to target matrix. The cache thrashing leads to more memory accesses. As a result, PEI and ART have similar memory access intensities. For regular memory accesses in terms of network power, ART feeds the data in the network with the minimum route while PEI brings data all the way to the CPU, thus PEI consumes more power. One exception is *pagerank*, which has irregular memory access patterns. ART offload computation *flows* in single operand granularity, causing high overhead in offloaded packets and operand packets, thus consumes more network power.

Microbenchmark *mac* has similar power characteristics as the benchmarks behaving regular

memory access patterns. For *reduce*, ART-tid/addr can massively process the reduction near-data in memory cubes without moving data around, which leads to more intensive memory accesses and more offloading. For irregular memory access patterns such as *rand_reduce* and *rand_mac*, PEI exhibits no reuse of the data and can only optimize atomic updates, leading to intensive memory accesses which consume more power.

5.4.2.2 Energy Consumption

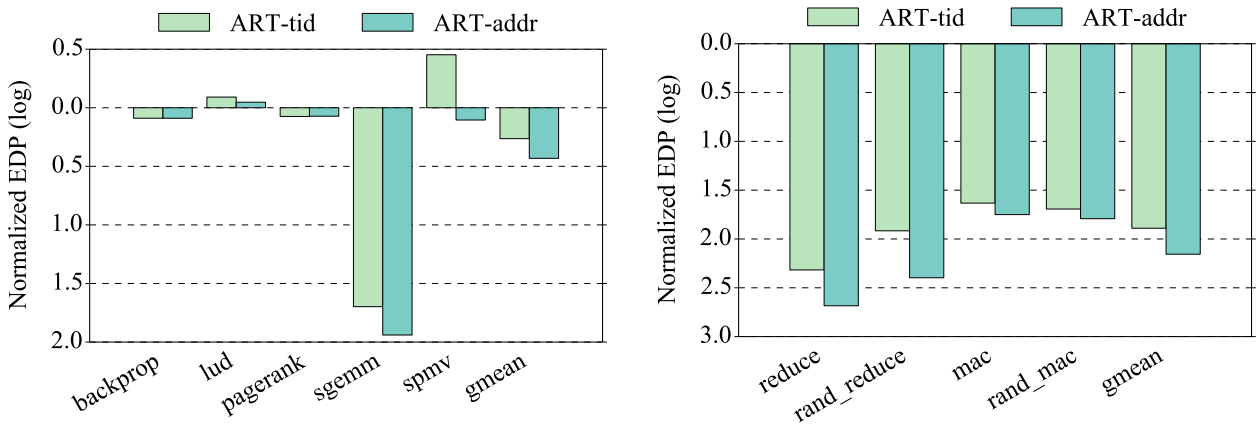
Figure 5.12 shows the energy consumption for cache, memory and memory network. ART-tid/addr reduces the energy consumption across all the benchmarks with regular memory access patterns and microbenchmarks. For applications that have irregular access patterns such as *pagerank*, the main contribution is from network energy that has high overhead due to fine-grained offloading. For *sgemm* and microbenchmarks, energy consumption is reduced dramatically due to significant running time speedup. We gain enormous benefit because most parts of these applications can be optimized by *Active-Routing*.

5.4.2.3 Energy-Delay Product

Figure 5.13 shows the normalized energy-delay product (EDP) over PEI in logarithmic scale to show the energy efficiency. We observed that ART-tid/addr has lower EDP for all applications except for *spmv* with ART-tid. The reductions in execution time as well as energy consumption contribute jointly to EDP reduction, achieving significant energy efficiency improvements. In *spmv* with ART-tid, the imbalanced work distribution leads to worse execution time. Since the energy saving is offset by the performance degradation, ART-tid on *spmv* has lower EDP. To summarize, ART-tid and ART-addr reduce the EDP by 80% on average compared to PEI.

5.4.3 Dynamic Offloading: A Case Study

In this section, with the help of an example we show that the performance can be further improved using a runtime knob. The runtime knob dynamically decides whether to offload computations (**Updates**) on the basis of memory access and communications patterns to achieve more performance gains. Execution phases that exhibit good locality of data accesses experience per-



(a) Benchmarks

(b) Microbenchmarks

Figure 5.13: Logarithmic scale of normalized energy-delay product (EDP) over PEI.

formance benefits by exploiting cache hits when scheduled on the host processor. In *lud*, it decomposes a matrix into upper and lower triangular matrices. Computations for these two matrices can be broken into two different phases. The first phase is for computations of the upper triangular matrix and the other is for those of the lower triangular matrix and they are executed iteratively. These two phases have different locality of data accesses. The second phase has a good data locality since its data access pattern is row-major order, whereas the data access pattern of the first phase is in column-major order.

For such a program behavior, the best execution model is to use *Active-Routing* for the first phase and process the second phase in the host processor. We analyze *lud*'s phase behaviors as shown in Figure 5.14. ARTtid always offloads computations to memory regardless of data locality, so the number of cycles for first and second phases in each iteration dramatically increases and decreases. However, when we run ARTtid-adaptive in which computations of the first phase are offloaded to the memory and that of the second phase is processed in the host processor, we achieve 2× speedup.

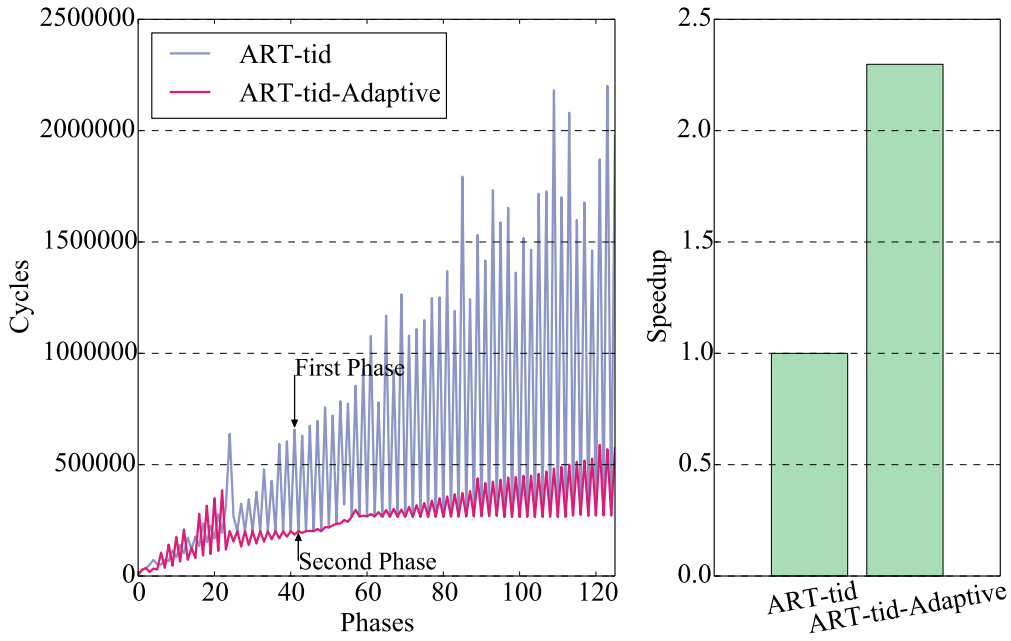


Figure 5.14: LUD phase analysis and dynamic offloading

5.5 Summary

This chapter proposes *Active-Routing*, an in-network compute architecture, to accelerate reduction on data processing operations in data-intensive applications for near-data processing. *Active-Routing* is implemented as a novel three-phase processing schedule, which offloads the computation near data in the memory network for execution and aggregates the results along their routing path. We categorize memory access patterns of compute kernels of interest and offload the computations in various granularities by exploiting their locality characteristics to reduce offloading overhead. Compared to the state-of-the-art PIM architecture, *Active-Routing* can achieve up to 7 $\times$ speedup with a geometric mean of 60% performance improvement and reduce energy-delay product by 80% on average, showing promising potential for in-network computing and data-flow processing in memory network.

6. ALGORITHM/ARCHITECTURE CO-DESIGN FOR DISTRIBUTED DEEP LEARNING

Large-scale distributed deep learning training has enabled developments of more complex deep neural network (DNN) models to learn from larger datasets for sophisticated tasks. In particular, distributed stochastic gradient descent intensively invokes *all-reduce* operation for gradient synchronization to update models, which dominates communication time during iterative training epochs. This chapter, we identify the inefficiency in recent all-reduce algorithms, and the opportunity of communication algorithm and architecture co-design. In addition, we propose MULTITREE All-Reduce algorithm with topology and resource utilization awareness for efficient and scalable all-reduce operation. Moreover, we specialize the interconnection network to match the injection/ejection bandwidth to network bandwidth in addition to flow control to cope with the algorithm in synergy.

6.1 MultiTree All-Reduce Algorithm

In this section, we first explain the rationale behind the MULTITREE approach. More specifically, we will shed light on why it is necessary to have multiple trees instead of rings and why network topology should be considered in the algorithm. Then, we describe the main algorithm with an example.

6.1.1 MULTITREE Insights

6.1.1.1 Why All-Reduce Trees?

In reduce-scatter and all-gather phases of all-reduce, each node leads a reduction and a broadcast of one chunk of the data. In Ring All-Reduce, each node communicates a chunk of data in a unidirectional ring, which takes $(n-1)$ steps during reduce-scatter and another $(n-1)$ steps during all-gather for n nodes. If each such communication can take place in a tree structure, it can reduce the algorithmic steps to $2\log_k n$ with a k-ary tree for n nodes. Moreover, a ring can be considered a generalization of a unary tree. Thus, the proposed algorithm to construct multiple trees instead of rings not only is bandwidth optimal but also reduces latency, thereby improving

all-reduce scalability.

6.1.1.2 Why Topology Awareness?

If trees are constructed without considering network topology and link utilization, it may lead to even worse performance than rings. For example, when all the reduction trees start from leaf nodes, there can be cases in which the same pair of parent and child in different trees are involved in the same communication step at the same time if trees are not constructed carefully. Furthermore, since tree levels closer to leaf level are denser than tree levels closer to roots, the communication in the reduce-scatter phase will experience from dense to sparse when reducing from leaves to roots, which may lead to high congestion near leaf nodes. Thus, MULTITREE exploits this insight to combine message scheduling and tree constructions, with link utilization awareness to schedule more communication near the roots to sparsify communication near leaves. Moreover, rather than serializing the tree constructions one by one, MULTITREE builds the trees concurrently, thereby making them balanced with global coordination.

6.1.2 MULTITREE All-Reduce Algorithm

MULTITREE aims at achieving optimal bandwidth and low latency, in addition to flexibility and scalability by exploiting the topology information and resource usage during the scheduling and tree construction phase. Unlike prior work that constructs trees from bottom-up [33], MULTITREE builds the trees from roots to leaves using a top-down approach. This approach can better leverage the network topology to build k-ary trees even with higher k values. As a result, it adds as many nodes as possible to make the predecessor levels denser and the tree shorter. Given a network $G(V, E)$ with nodes V and edges E, finally $|V|$ trees will be created, where each tree spans all the nodes. The pseudo code is listed in Algorithm 3.

6.1.2.1 Algorithm Description

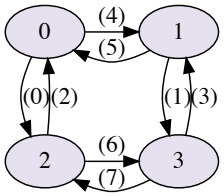
The algorithm first initializes each tree to start from each of the nodes in the network as well as the time step t, as listed at lines 1–3. Then it starts to construct the schedule trees for the all-gather phase (instead of reduce-scatter) since it is more natural to start from the root. This is listed

in lines 4–14. For every new time step t, a full topology graph $G'(V', E')$ is used, whose edges will be removed while adding new nodes to the trees. During this time step t, trees take turns to add one node c to connect to a node p that was added in previous time steps. Then the edge $p \rightarrow c$ is removed from the topology graph and scheduled for communication at the current time step t. Note that trees alternate by root ID in ascending order for simplicity, which works fine in most cases, especially for symmetric networks like torus. For asymmetric or irregular networks, trees with larger remaining height can be prioritized so that communication on the longest path is scheduled earlier. At line 9, nodes are examined breadth-first in their order of adding to the tree by previous time steps so as to make the predecessor levels denser. For selecting a neighbor of p at line 10, it first checks the neighbors in Y dimension then in X dimension for torus and mesh networks. Other structural information can be used for asymmetric and irregular networks, which we leave for future study. When the topology graph runs out of edges to connect new nodes for all the trees, it starts a new time step and repeats the algorithm until all the all-gather schedule trees are completed. After all-gather schedule trees are constructed, they are used to construct reduce-scatter trees and adjusted for communication time step. This procedure is listed in lines 16–18. Since reduce-scatter goes in the opposite direction with respect to all-gather communication, the algorithm simply reverses the communication pairs of all-gather schedule trees with adjusted time steps. The all-gather schedules are also adjusted in time to run after reduce-scatter schedules. In static systems, the algorithm only needs to run once and can be used for any DNN workloads. In dynamic and shared systems, it runs every time a new set of nodes is allocated for the workloads.

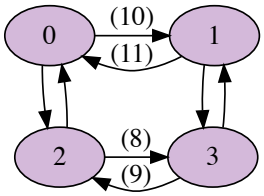
Algorithm 3: MULTITREE All-Reduce Algorithm.

Input: $topology\_graph\ G(V, E)$
Output: $reduce\_scatter\_schedule,\ all-gather\_schedule$

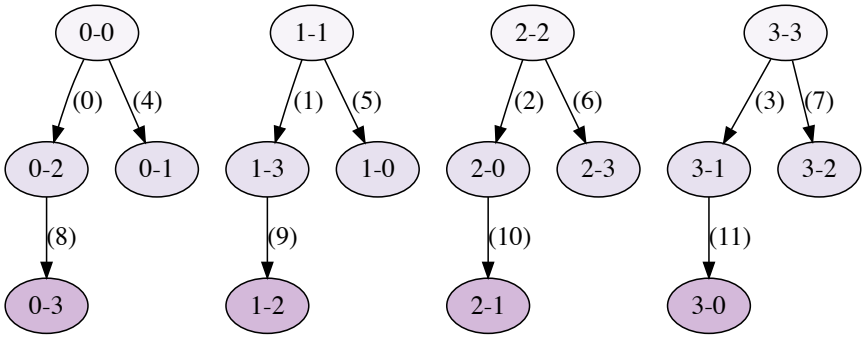
```
// Initialization
for each node i ∈ V of graph G(V, E) do
    Tree T_i adds node i to tree T_i as root;
t = 0;
// Compute all-gather schedules
while not all trees completed do
    // Start a new time step t with a new G
    t = t + 1;
    G'(V', E') = G(V, E);
    // Add new nodes to trees and schedule // communications for this
    time step
    while E' has free edges to add new nodes do
        // Trees take turns for balancing
        Select next tree T by root ID in ascending order;
        for p ∈ T's nodes added by previous time steps do
            if there is an edge (p → c) ∈ E' then
                Add node c to T and connect to p;
                Remove edge p → c from E';
                // Schedule message p → c at t
                Add (p → c, t) to T's all − gather_schedule;
                break;
Calculate total time steps tot_t = t;
// Compute reduce-scatter schedule, which
// is the reverse of all-gather
for (p → c, t') ∈ all − gather_schedule of each tree T do
    Add (c → p, tot_t − t' + 1) to T's reduce_scatter_schedule;
    // Adjust all-gather schedule
    Replace (p → c, t') with (p → c, tot_t + t');
```



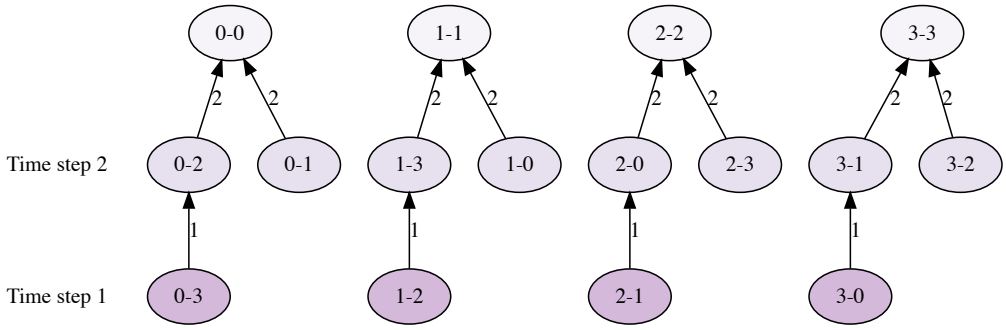
(a) Time step 1



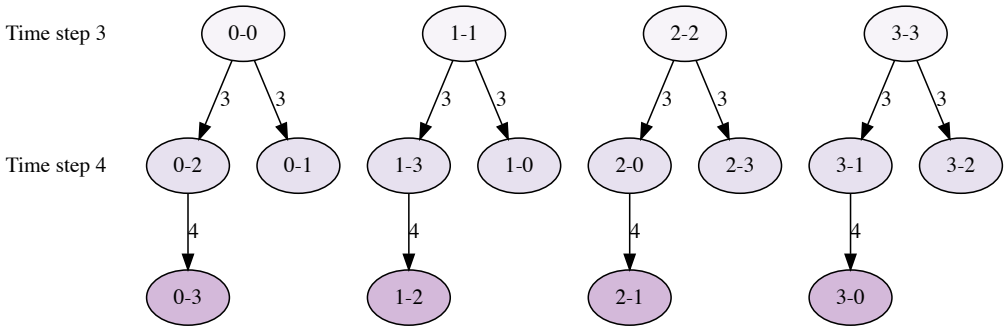
(b) Time step 2



(c) Tree construction (link allocation) at levels (time steps) 1 and 2

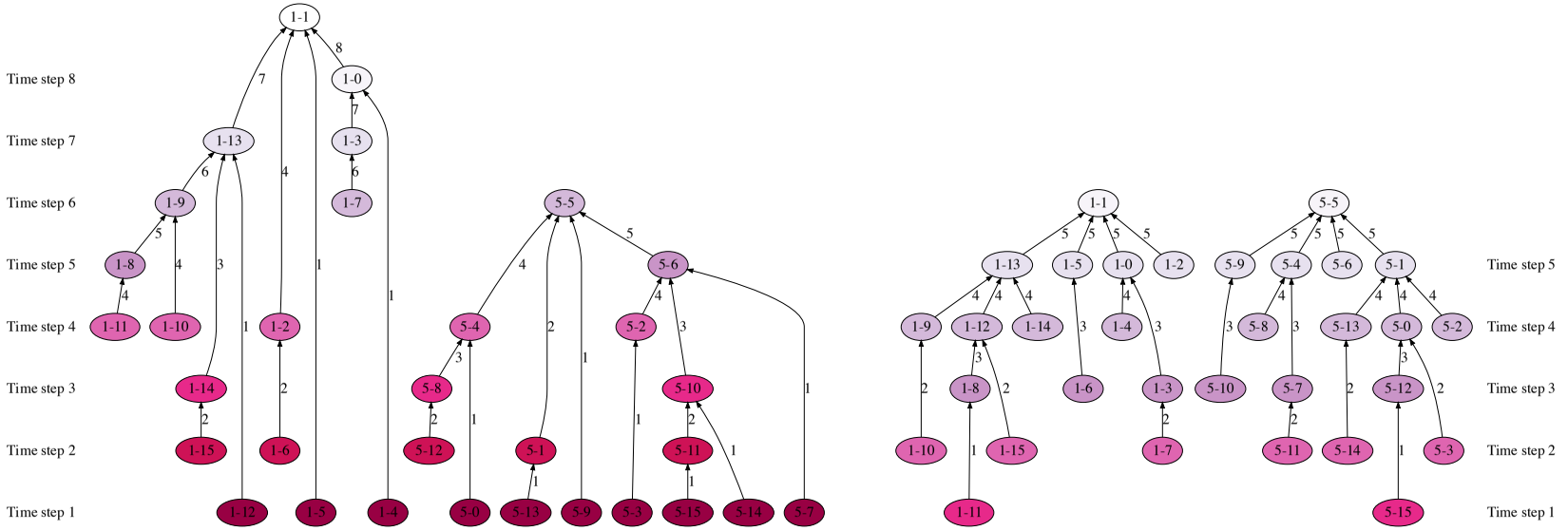


(d) Reduce-scatter schedule trees



(e) Allgather schedule trees

Figure 6.1: MULTITREE construction with link allocation and scheduling for all-reduce communication of a (2×2) mesh network. Node `n` in tree `T` is denoted as (`T-n`) and label `(i)` of an edge is the allocation sequence of that link while label `t` of an edge is the communication time step between the two nodes: link allocation sequence of the topology graph for level 1 (time step 1) (a); when no more links are available for the predecessor levels 0 and 1, a new link topology graph is used for allocation for level 2 (time step 2) (b); the tree construction process indicated by edge labels (c); the constructed reduce-scatter schedule trees (d) and all-gather schedule trees (e).



(a) MXNETTREE (b) MULTITREE

Figure 6.2: Reduce-scatter schedules of all-reduce trees 1 and 5 of a (4×4) 2D-Torus network constructed by MXNETTREE (a) and MULTITREE (b), where a tree node (T-n) indicates accelerator n in tree T and the edge label denotes the scheduled communication time step.

6.1.2.2 Complexity Analysis

The most expensive part of the algorithm is the loop for all-gather schedule tree constructions, as listed in lines 4–14. Let us assume a topology graph $G(V, E)$. The core part of adding new nodes to schedule trees is from lines 9–14. To add a new node, the algorithm checks whether the already added nodes of that tree still have edges connected to a pending node. In the worst case, it may check all the edges of the graph, which is $|E|$. In total, we have $|V|$ trees and each tree has $|V|$ nodes. So the worst case is $O(|V|^2|E|)$.

6.1.2.3 Concentrated and Indirect Networks Support

In Algorithm 3, $G(V, E)$ is the `switch-to-switch` adjacency list with the assumption that each router switch is attached with a node. In order to support concentrated networks as well as indirect networks, we extend the topology graph with additional `node-to-switch` and `switch-to-node` connection lists. To find an available child c for a node p, it follows breadth-first search on these three topology components as described in the following steps:

1. Get p's attached switch sw_0 from its `node-to-switch` list.

2. When multiple nodes are attached to the same switch, check whether sw_0's `switch-to-node` has connections to connect with p. If there is an available connection, pick a node as c and remove one connection ($p \rightarrow sw_0$) from p's `node-to-switch` list and one connection ($sw_0 \rightarrow c$) from sw_0's `switch-to-node` list, then return. If there is no available connection, go to step 3.

3. Get sw_0's neighbor switch sw_1 from its `switch-to-switch` list. Repeat the same process as step 2 with sw_1 until a node c is found or no connection is available. In this case, if a node is found, besides the connections removed in step 2, connections in traversed `switch-to-switch` lists should also be removed for the allocated links.

6.1.2.4 Bandwidth and Latency Comparisons

An ideal algorithm should be optimal in both bandwidth (each *router* communicating the minimum amount of data) and latency (minimum all-reduce completion time). Theoretically, MULTITREE aims at building multi k-ary trees for all-reduce, which has tree height of $\log_k n$ for n nodes, where ring and butterfly exchanges [152] are special cases whose k is 1 and 2, respectively. Hence, MULTITREE can achieve at least the same algorithmic steps as the butterfly. When the all-reduce data size is small and the network has sufficient bandwidth, butterfly can achieve contention-free communication with better latency than ring. However, for large data such as giant DNN models, serialization latency of big messages and contention play important roles in latency. Butterfly can face huge link contentions and perform worse than ring [31]. The multi-hop communication between a pair of nodes introduced by a butterfly-unfriendly topology can even worsen the situation. Multi-phase rings may reduce algorithmic steps, but not necessarily latency due to multi-hop communications between two nodes in the logical rings. In addition, bandwidth-waste algorithm that aggregates more data in a single reduction pass may reduce algorithmic steps by half, which works better with small data but not for large DNNs, since serialization latency of larger communicating data is more dominant. Building upon these, we propose topology-aware MULTITREE algorithm to reduce distance between pairs of communicating nodes to one hop, meanwhile couple tree constructions and link scheduling to effectively eliminate contention. Therefore, it not only achieves optimal bandwidth the same as ring but also reduces latency. It is worth mentioning that MULTITREE is well suited for various network topologies. In the future, we plan to extend this work for bandwidth trade-off to achieve optimal latency with more network information.

6.1.3 An Example

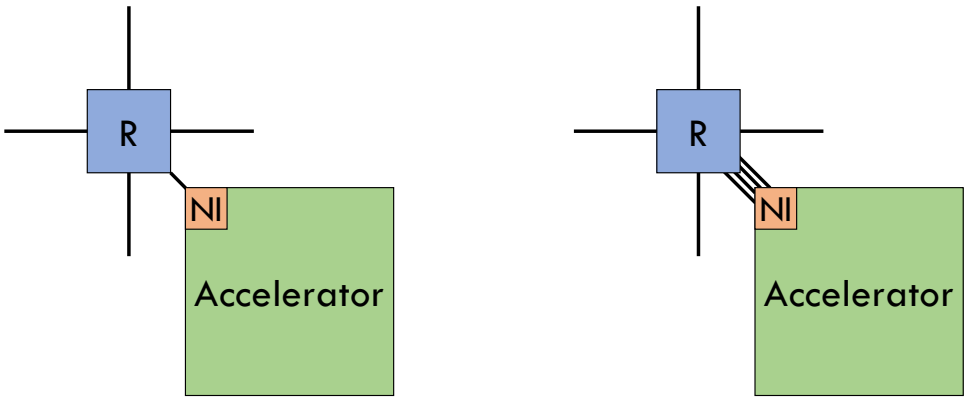
We demonstrate the algorithm steps by walking through an example that constructs all-reduce schedule trees of a (2×2) mesh network, as shown in Figure 6.1. Figures 6.1a and 6.1b show the topology graphs that are used for schedule tree construction for time steps 1 and 2, respectively. The edge label `(i)` in Figures 6.1a, 6.1b and 6.1c indicates the global sequence in which a node

is connected to its parent in a tree during construction. Figure 6.1c shows all the trees and their construction sequence, where the trees take turns to add one node at a time. This results in a balance among the trees. At sequence number 7, after the last edge ($3 \rightarrow 2$) is added to connect nodes 2 and 3 in tree 3, the topology graph runs out of edges for time step 1 to connect new nodes to any trees, indicating in the sequence numbers in Figure 6.1a. Therefore, a new topology graph in Figure 6.1b is used to start time step 2, which creates a new level for the trees. This reflects lines 4–14 of the Algorithm 3. These newly constructed trees are used to build the reduce-scatter schedule trees and finally, are adjusted to generate all-gather schedule trees, as shown in Figures 6.1d and 6.1e, respectively. Note that the trees are well balanced and symmetric in shape, but not necessarily structurally symmetric. Structural symmetry requires special representation of each node with respect to the remaining network and only applies to specific symmetric networks. Moreover, for networks like a (4×4) mesh where the longest distance from a source node varies depending on its position, the trees are asymmetric with different heights.

Figure 6.2 shows the comparison of reduce-scatter schedule trees 1 and 5 of a (4×4) 2D Torus network constructed using MXNETTREE [33] and MULTITREE approaches. Figure 6.2a shows a possible communication scenario using MXNETTREE, where the reductions are initiated from all the leaf nodes of all the trees at the same time, making the trees imbalanced and leaf levels dense. This is because the tree construction process has not considered the scheduling, thereby causing more link conflicts near leaf levels leading to a higher number of communication steps (i.e., 8). In contrast, MULTITREE couples tree constructions with link scheduling and builds the trees concurrently in a top-down manner, generating balanced trees with denser predecessor levels and sparser leaf levels, which effectively reduces the link conflicts near leaf levels and reduces the communication steps to 5, as shown in Figure 6.2b.

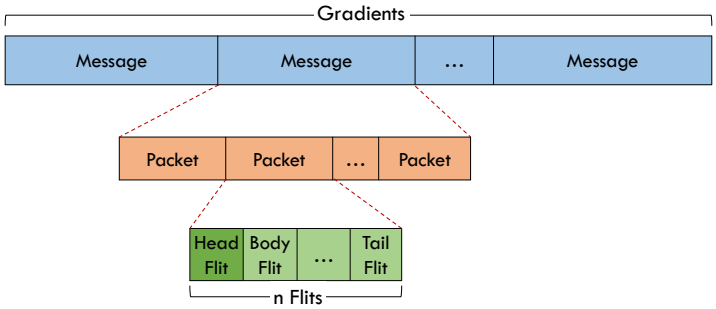
6.2 Architectural Supports

In this section, we co-design communication architecture to support MULTITREE All-Reduce. Additionally, we specialize flow control to exploit big gradient exchanges for communication acceleration.

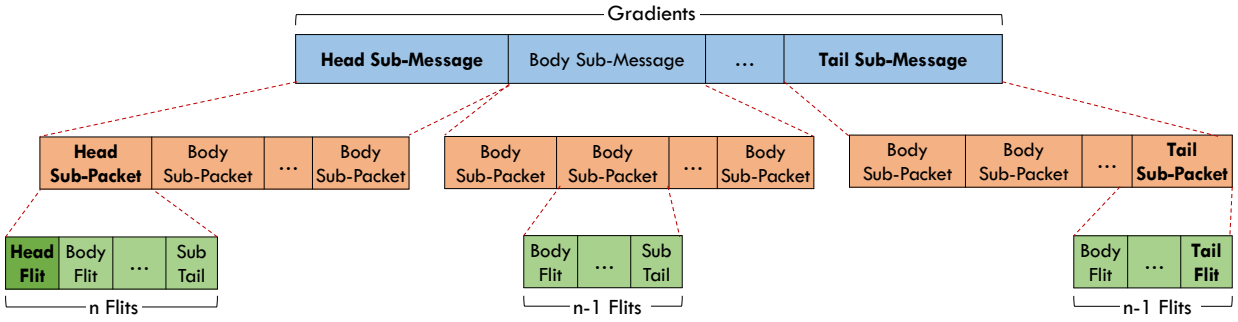


(a) Narrow network interface (b) Wide network interface

Figure 6.3: Conventional narrow network interface (a) and wide network interface dedicated for MULTITREE (b), where R is router and NI is network interface.



(a) Original gradient message



(b) Big gradient message

Figure 6.4: Original many messages with small packets of gradients (a) and big message with large packet of full gradients (b).

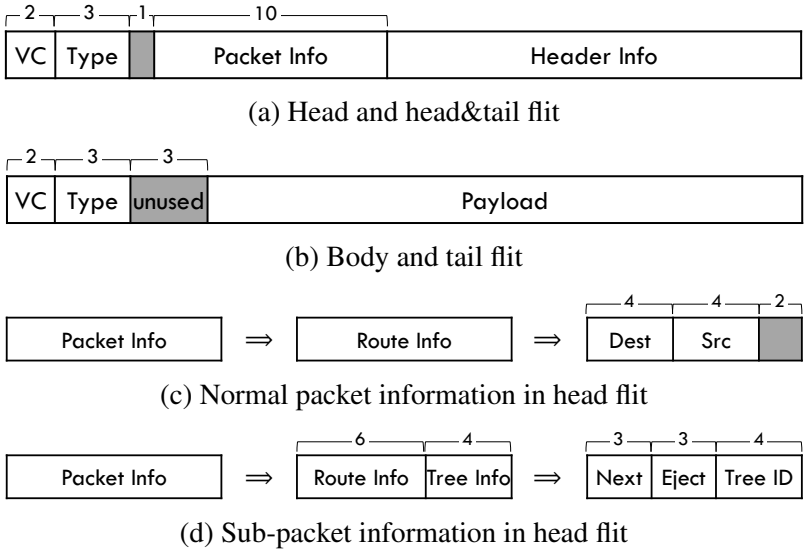


Figure 6.5: Flit formatting in a (4×4) Torus network for head and head&tail flit (a), body and tail flit (b), packet information in head flit for normal packet (c) and sub-packet (d).

6.2.1 Wide Network Interface for MULTITREE

MULTITREE all-reduce leverages the network topology and router radix to construct trees with higher fanout rather than the serialized rings. However, the homogeneity of injection/ejection bandwidth and network channel bandwidth in a general purpose network is insufficient to support the trees' fanout requirements. Even worse, this may lead to performance degradation. Therefore, architectural supports are required to achieve the full potential of the algorithmic benefits. To this end, we specialize the interconnects for MULTITREE All-Reduce algorithm and propose heterogeneous bandwidth provisioning for injection/ejection and network channels.

Figure 6.3a shows a commonly used narrow network interface (NI) configuration that provides the same bandwidth as the network channels. This simple design is general enough to support short and random messages that can efficiently sustain the network throughput. However, it is inefficient to support MULTITREE algorithm. For example, during the reduce-scatter phase in Figure 6.1d, node 0 needs to communicate with node 1 and 2 at the same time. Either blocking or multiplexing due to the narrow NI can delay the completion time of reduce-scatter operations, thereby failing to exploit the topology for performance improvement from the algorithm. To solve this problem, we

propose wide NI according to the high fanout requirement of MULTITREE algorithm. A wide NI is shown in Figure 6.3b. We design the NI connection with higher bandwidth to match the network bandwidth. In order to achieve best algorithm performance, the NI width is decided according to the router radix in the network. With this torus co-designed hardware feature, MULTITREE can effectively leverage the topology knowledge and accelerate all-reduce communication.

Table 6.1: Packet and Flit Types

Type		**Code**
Packet	**Flit**	
Normal Packet	Head	0 0 0
	Body	0 0 1
	Tail	0 1 0
	Head & Tail	0 1 1
Sub-Packet	Head	1 0 0
	Body	1 0 1
	Sub-Tail	1 1 0
	Tail	1 1 1

6.2.2 Message-based Flow Control for Big Gradient Exchanges

Unlike general purpose applications, all-reduce communication in data-parallel DNN training has a relatively fixed traffic pattern. With a particular all-reduce algorithm, the communication pattern is known apriori for a training task. For example, MULTITREE constructs the schedule trees before training starts. This prior knowledge can be leveraged for simpler control and arbitration in hardware, thereby simplifying logic and improving energy efficiency. MULTITREE algorithm aims to coordinate among the trees with a global view, where less dynamism in interconnection networks helps maintain the communication schedules, thereby keeping concurrent communications progressing at a similar rate. In addition, the long traffic (between a communicating pair) for all-reduce of large gradients unnecessarily incurs bandwidth overhead of massive number of packet head flits. To optimize these aspects, we revisit the traditional flow control techniques and

redesign them specifically for all-reduce communication.

Figure 6.4a shows a commonly used packet-based switching mechanism, where large gradients are divided into many messages. Each message is partitioned into multiple packets. Each packet consists of a head flit and body/tail flits. The highlighted head flits consumes bandwidth and incurs extra control such as routing and arbitration, causing extra delay and power consumption. On the other hand, we adapt a message-based approach to reduce such overheads, as shown in Figure 6.4b. Instead of having fixed message size, we take the whole chunk of gradients as a message, which can be further converted to many sub-messages starting with a head sub-message and ending with a tail sub-message. Each sub-message is divided into sub-packets, where the first sub-packet of the head sub-message is a head sub-packet, which behaves as the head of the large gradient message. The last sub-packet of the tail sub-message is the tail sub-packet to end the gradient message. Similarly, the sub-packets are partitioned into flits. Unlike conventional packet-based switching, body and tail sub-packets start with a body flit, while head and body sub-packets end with a sub-tail flit to indicate the completion of a sub-packet. This leads to only one head flit overhead for a large gradient message. This way, we can achieve near perfect bandwidth efficiency and reduce the control and arbitration overhead, thereby improving performance and energy efficiency.

The flit formats are detailed in Figures 6.5a and 6.5b for head/head&tail flit and body/tail flit, respectively. The `VC` field indicates the allocated virtual channel and the `Type` field specifies the packet and flit type, as listed in Table 6.1. The `Packet Info` field are encoded differently for normal packets and all-reduce sub-packets, as shown in Figures 6.5c and 6.5d. For normal packets, the `Packet Info` is simply the `Route Info`, including `Dest` and `Src` that are used by distributed routing algorithms. For all-reduce sub-packets, `Packet Info` includes both `Route Info` and `Tree Info`, where the `Tree Info` is the `Tree ID` that this message belongs to. Since MULTI-TREE only communicates between neighbors, we use source routing to include the next hop output port `Next` and ejection port `Eject` in the head flit. In the network interface, these information are pre-computed and stored in `Route Info`, which can be directly used in the routers. More specifically, in the source router, the `Next` field is used to route to the neighbor, which will interchange

with the `Eject` field after the routing computation stage. The `Next` field is kept towards the destination in order to identify which child the message is from to clear dependencies for scheduling purposes. To keep track of the message status, the network interface is augmented with message records as depicted in Figure 6.6.

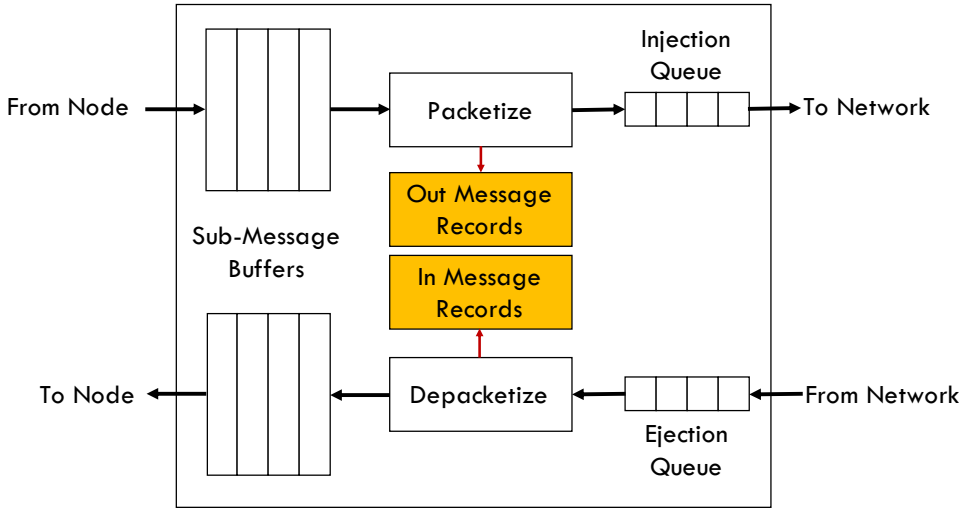


Figure 6.6: Augmented network interface for sub-message and sub-packet.

Since MULTITREE All-Reduce only schedules communications between two neighboring nodes, the flits always take one hop. Therefore, such a design does not increase the possibility or risk of deadlock. Note that it can still work with wormhole switching seamlessly to support other types of traffic, such as control and synchronization traffic. Virtual channels are used to avoid starvation of these short messages.

6.3 Methodology

6.3.1 System Modeling and Configuration

We extended a neural network inference simulator, `SCALE-Sim` [153], to support back-propagation for training. Both the forward and backward passes apply output stationary dataflow. Other types of dataflow and optimizations in computation acceleration are out of scope of this paper. We configured a TPU-like accelerator consisting of 16 processing elements (PEs), where each PE had a

Table 6.2: MULTITREE System Configuration

Parameter		Configuration
PE	MAC array	32×32
	Dataflow	Output Stationary
	Precision	32 bits
Accelerator	Number of PEs	16
	Clock	1 GHz
Network	Number of Accelerators	16
	Topology	4×4 2D Torus
	Routing	Dimension-Order w/ Dateline
	Flow Control	Virtual Cut-Through
	Number of VCs	4
	VC Buffer Depth	318 flits
	Data Packet Payload	256 Bytes for Baselines
	Router Clock	1 GHz
	Link Latency	150 ns
	Link Bandwidth	16 GB/s

(32×32) systolic array. We assumed double buffering and sufficient memory bandwidth (such as high bandwidth memory) to maintain the peak compute throughput as much as possible.

For the interconnect of accelerators, we formed a (4×4) 2D Torus for a 16-accelerator pod that is similar to Google Cloud TPU [154] and Microsoft Catapault [155]. We also evaluated scalability by extending the network size up to 256 accelerators. A recent indirect BiGraph network was also evaluated [126]. We use `BookSim` [156] for interconnect modeling where the network interface module was augmented to support the co-designed MULTITREE All-Reduce algorithm. Network interfaces and routers are integrated on-chip with accelerators for lower cost and better performance. The link latency and bandwidth was chosen to reflect off-chip inter-board communication, while the buffer size was configured to cover the credit round-trip latency. We integrated `SCALE-Sim` and `BookSim` to model both computation and communication of DNN training, but the protocol overhead between host and accelerators was not simulated. The accelerator is also used for aggregation during all-reduce communication. MULTITREE All-Reduce schedule was implemented as a module in the accelerator node to regulate the gradient injection for schedul-

ing purposes. We use DSENT power model [34] for on-chip router and links, and assume 30 pJ/bit for off-chip link with power-gating [157]. Unless otherwise stated, the default configuration parameters are listed in Table 6.2.

6.3.2 Workloads

We evaluate the co-designed algorithm and architecture using state-of-the-art DNN models provided by `SCALE-Sim` [153], including *AlexNet* [10], *AlphaGoZero* [158], *FasterRCNN* [159], *GoogLeNet* [160], NCF recommendation *(NCF)* [161], *ResNet50* [24] and *Transformer* [25, 162]. We ran with a minibatch size of 256 (16 samples/minibatch for each accelerator) and evaluated the training time (forward+backward) and all-reduce communication time for one iteration. We also estimated the optimistic computation/communication overlap with layer-wise all-reduce.

For sensitivity study, we simulate a synthetic DNN model for both strong and weak scalability. We use a fixed per node model size of 375 KB for weak scalability and a fixed total model size of 32 MB for strong scalability. We also run strong scalability for the DNN workloads.

Table 6.3: Baselines and MultiTree Configurations.

Technique	Description
RING	*Topology-aware* Baidu Ring All-Reduce [117]
RING-γ	*Topology-aware* Ring All-Reduce with message-based flow control
2BINARYTREE	Double-binary tree all-reduce [32, 123]
MXNETTREE-α	MXNETTREE without architectural support [33]
MXNETTREE-β	MXNETTREE with wide network interface
HDRM	Halving-doubling with rank mapping [126]
HDRM-γ	HDRM with message-based flow control
MULTITREE-α	MULTITREE without architectural support
MULTITREE-β	MULTITREE with wide network interface
MULTITREE-γ	MULTITREE with message-based flow control
MULTITREE-δ	MULTITREE with wide network interface interface and message-based flow control

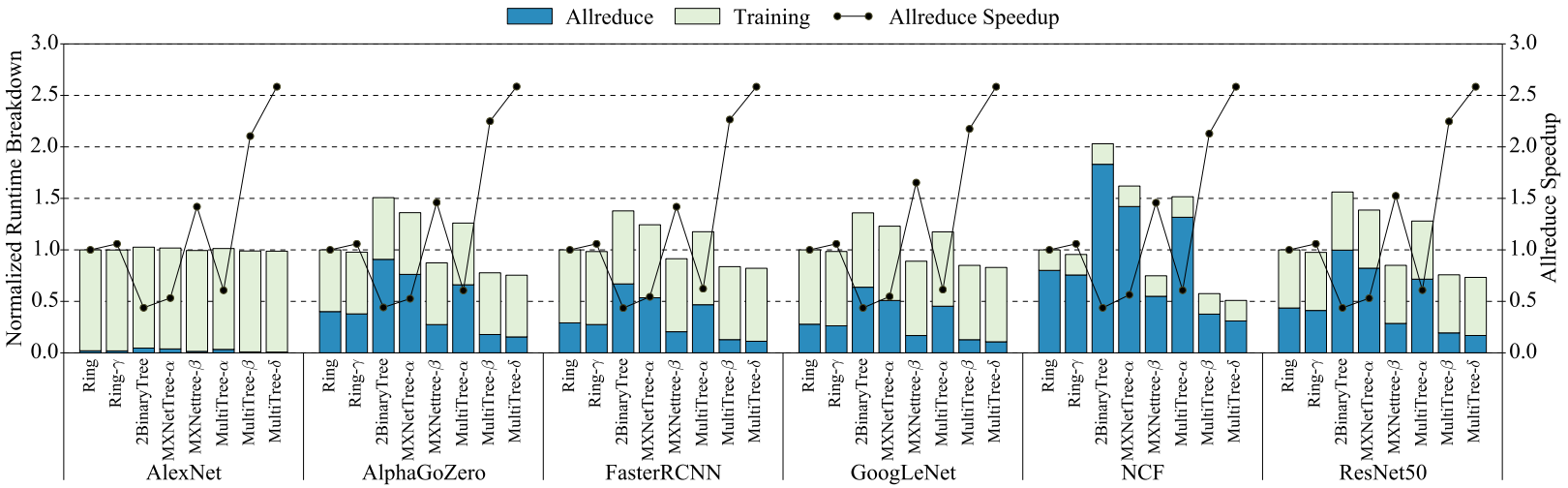


Figure 6.7: Runtime breakdown of training and all-reduce (primary) and all-reduce speedup (secondary) normalized to RING in 4×4 Torus network (α: without architectural support, β: with wide network interface, γ: with message-based flow control, δ: with both wide network interface and message-based flow control).

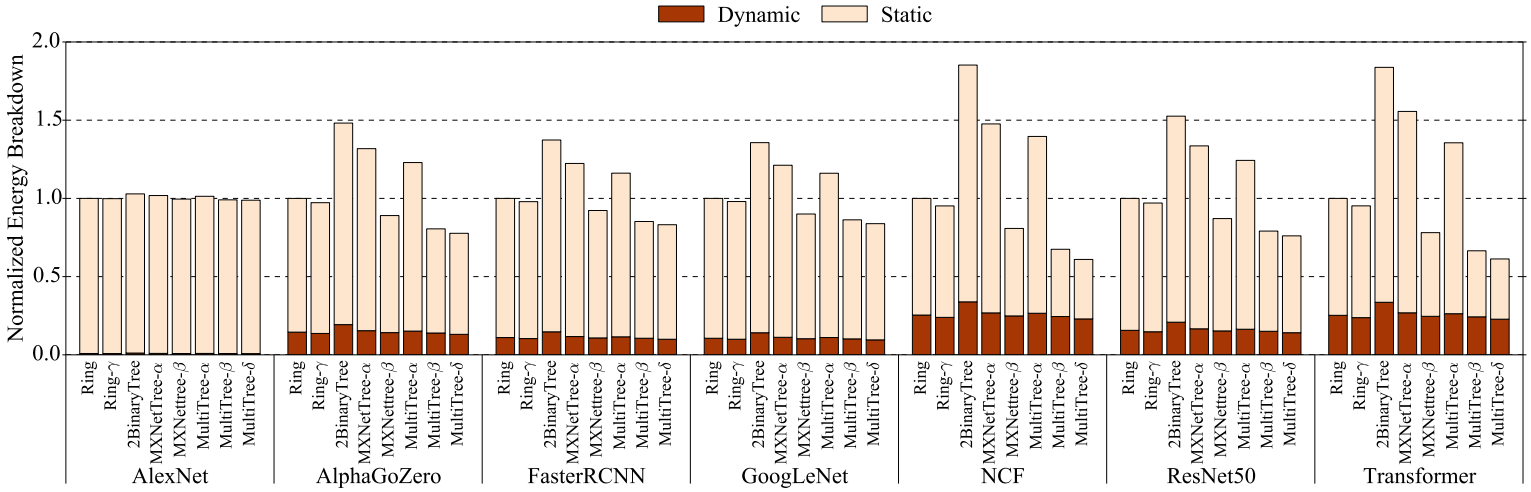


Figure 6.8: Dynamic and static energy consumption normalized to RING in 4×4 Torus network (α: without architectural support, β: with wide network interface, γ: with message-based flow control, δ: with both wide network interface and message-based flow control).

6.4 Evaluation

In this section, we evaluate performance speedup, energy consumption, and network scalability comparisons among RING, double-binary tree (2BINARYTREE) [123, 32], MXNETTREE [33], halving-doubling with rank mapping (HDRM) [126], and the proposed MULTITREE, as described in Table 6.3.

6.4.1 Performance

Figure 6.7 shows the normalized runtime breakdown of training and all-reduce using RING. Except for *AlexNet*, other DNN models have a considerable amount of time spent on all-reduce communication, leaving the compute nodes in idle state. CNNs such as *AlexNet*, *FasterRCNN*, *GoogLeNet*, and *ResNet50* are compute-intensive and need to compute transposed convolution to calculate the gradients in order to propagate back to the previous layer. In contrast, *NCF* and *Transformer* models have more embedding and attention layers, which have less computation requirements, making communication more dominant. In summary, communication time can vary from 30%–88% in the baseline RING.

Figure 6.7 also shows normalized all-reduce speedup over RING. MXNETTREE-α and MULTITREE-α are pure communication algorithms without architecture co-design, which show similar performance degradation for all the benchmarks compared to RING. This is because the fanout of the network interface does not match the degree of the tree, leading to serialization or interleaving of concurrent communications. This type of contention due to limited hardware support leads to worse performance than RING which is a perfect unidirectional pipeline without any congestion. Therefore, it is essential to co-design architecture and algorithm to realize the full potential of algorithmic optimizations.

MXNETTREE-β and MULTITREE-β are co-designed with architecture to have a wide network interface in order to satisfy the fanout requirement of the trees. With this architectural support, MXNETTREE-β and MULTITREE-β improve performance over RING by 40% and 90%, respectively, demonstrating the effectiveness of communication algorithm and architecture co-design.

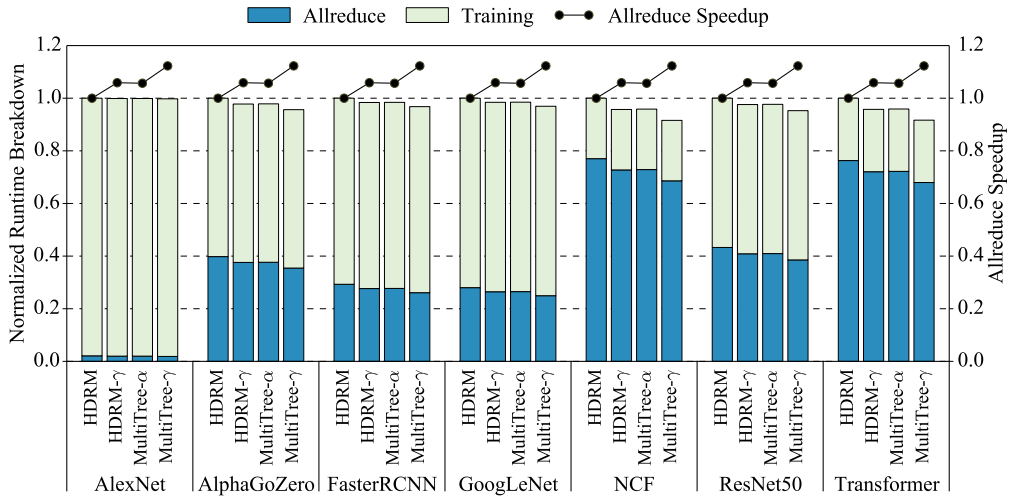


Figure 6.9: Runtime breakdown of training and all-reduce (primary) and all-reduce speedup (secodnary) normalized to HDRM in 32-node 4×8 BiGraph network (α: without architectural support, γ: with message-based flow control).

MULTITREE-δ pushes the performance improvement further by specializing the message-based flow control to reduce unnecessary bandwidth waste during large gradient communications due to packet head flits. With this optimization, MULTITREE-δ improves performance by 2.5 times compared to RING.

While the tree-based algorithms outperform RING by using architecture co-design to benefit from tree structure, MXNETTREE and MULTITREE show performance differences due to inherent algorithmic design. As shown in Figure 6.7, MULTITREE-α/β perform better than MXNETTREE-α/β. It is because MXNETTREE constructs trees ignoring the underlying scheduling and resource availability, leading to contention in links and thereby degrading performance. On the contrary, MULTITREE leverages the concurrent communication among all the trees to combine scheduling with tree construction, which regulates message scheduling with a global coordination, thereby eliminating resources contention during communication. Even though the network may have dynamism, the interference is limited due to the large communication of big gradient messages. It is worth mentioning here that MULTITREE builds the trees using a top-down approach, being able to leverage all the fanout degrees of a router, unlike MXNETTREE which builds trees bottom-up using Kernighan-Lin binary partitioning for binary trees.

We also implemented the double-binary tree (2BINARYTREE) for comparison, which is even worse than RING. Since 2BINARYTREE is topology-oblivious and builds two logical trees, where connected nodes in the trees can cross multiple hops. Furthermore, the contention on links of large messages due to large models even worsen the performance. We also apply message-based flow control to Ring All-Reduce named as RING-γ, which shows about 5.9% improvement over RING owing to 6% bandwidth saving on head flits. If we double the bandwidth for RING for a similar hardware budget (10-port switch in RING while 8-port switch in MULTITREE), ideally it can reduce the time by half, which is similar to MULTITREE-β. After applying big-message flow control, it can further improve by 6%, which is in effect 2.1 times speedup over baseline RING, worse than MULTITREE-δ.

Applying MULTITREE to BiGraph Topology. Figure 6.9 shows the runtime breakdown and all-reduce speedup of MULTITREE-γ compared to HDRM on BiGraph topology [126], demonstrating the applicability of MULTITREE to concentrated and indirect networks. We observe that MULTITREE outperforms HDRM by 6% for all-reduce. This is mainly due to the fact that HDRM has communication only between higher and lower switches, even for nodes that are connected to the same switch, while MULTITREE first communicates with nodes within the same switch and reduces a switch hop. Note that communication latency is mainly determined by serialization and contention latency for large model size. Since both MULTITREEand HDRM are contention-free and communicate the same amount of data, the reduced number of steps in HDRM has minimum impact as its average communicated data per step is higher. Moreover, HDRM is tightly coupled with the fully connected BiGraph topology while MULTITREE is applicable to various network topologies.

Computation/Communication Overlap. Computation/ communication overlap can be achieved by *layer-wise all-reduce*. For an L-layer DNN, the non-overlap computation includes the forward pass and the back-propagation of layer$_{(L-1)}$. The non-overlap communication is the layer$_{(0)}$ all-reduce. Assuming perfect overlapping, the back-propagation (from layer$_{(L-2)}$ to layer$_{(0)}$) can overlap with the all-reduce (from layer$_{(L-1)}$ to layer$_{(1)}$). We found that all-reduce time can be largely

hidden by computation in most DNNs. But for the DNNs whose communication is more dominant, such as *NCF* and *Transformer*, MULTITREE achieves 2.1× speedup compared to RING.

6.4.2 Energy Consumption

Figure 6.8 shows the interconnect energy consumption of different schemes. One observation is that the static energy consumption is high for all of them across all the benchmarks. It is because of the long training computation time of the jobs which can cause standby leakage, contributing to static energy. For CNNs that have more compute portion as shown in Figure 6.7, the static energy portion is higher. During communication, if the network resources are not efficiently utilized, the network also tends to consume more static energy. The dynamic energy consumption mainly comes from data movement of gradients. The variance of energy consumption among different techniques may come from the contention that may lead to more arbitration overhead. Although marginal, it is still observable that algorithm co-designed architecture features save some energy. For MULTITREE-δ, the reduction in packet head flits contributes to more energy saving, where a small portion is from dynamic energy while a large portion is from static energy due to performance speedup. Therefore, communication algorithm/architecture co-design not only improves performance but also gains energy efficiency.

6.4.3 Scalability

6.4.3.1 Algorithmic Scalability

Figure 6.10a shows the algorithmic scalability of MULTITREE, MXNETTREE and RING. As they communicate the same amount of data, we only consider the communication steps derived from the algorithms, and normalize them to RING in a 16-node network. All the three algorithms scale linearly to the number of nodes while sustaining different linear factors, where MULTITREE is the best and RING is the worst. Although not achieving logarithmic scalability, MULTITREE improves the scalability over RING and MXNETTREE by factors of 4 and 2, respectively.

6.4.3.2 *Weak Scalability*

Figure 6.10b shows the weak scalability experiment by fixing the average model size per node to 375 KB, and scaling out the network size from 16 to 256 nodes. All the runtimes are normalized to RING's runtime with 16 nodes. All the three algorithms show linear scaling following algorithmic scaling with only a marginal difference. The runtime difference from algorithmic derivation may come from two sources. First, the runtime dynamics in the network and actual scheduling may not fully isolate the traffic flows serialized in the algorithm. For example, although the end node sends out all the packets of the previous communication, it may not know whether the network interface has finished the transmission, which may have small overlap with the later flow, incurring minor interference. Second, the algorithm has no consideration of data whose serialization latency of its data stream may offset the derived performance.

6.4.3.3 *Strong Scalability*

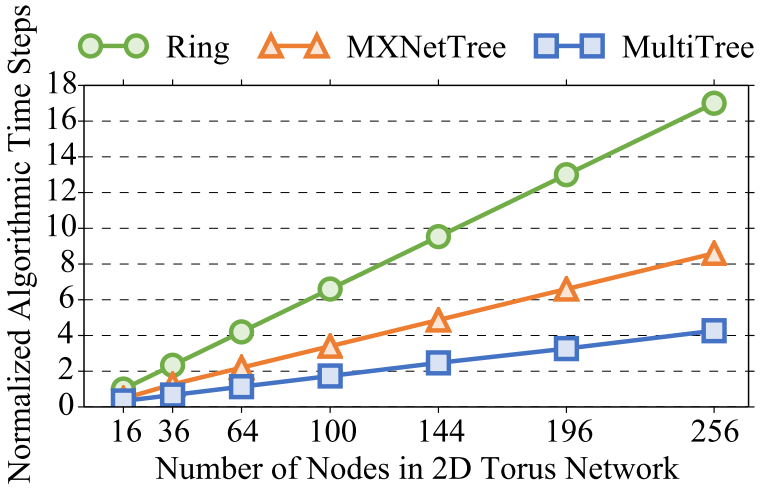
Figure 6.10c shows the strong scalability of the algorithms, where a synthetic and fixed 32 MB model size is used to test network scale from 16 to 256 nodes with performance normalized to RING in a 16 node network. All the three techniques have very stable runtime across different network scales, where MULTITREE performs the best while RING performs the worst. Given a particular model size M and network size of N nodes, the data size D that a node needs to communicate is reciprocal to N. And the communication steps S is linear to the number of nodes N with a ratio η as shown in Figure 6.10a. Therefore, we have:

$$D = \frac{M}{N} \quad \text{and} \quad S = \eta N$$

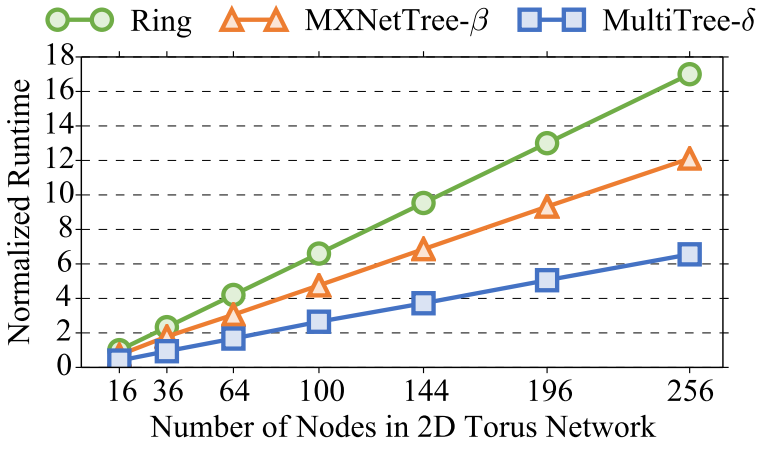
Without considering the overhead of packet head, we can derive an estimated runtime of these algorithms:

$$T_{estimate} \approx D \times S = \eta M$$

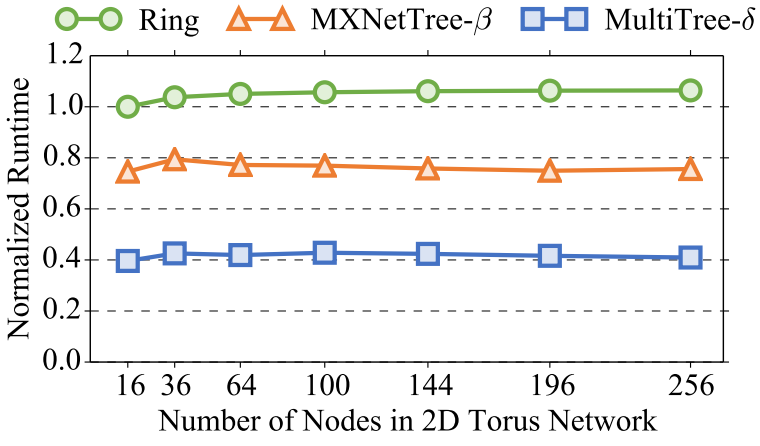
This indicates the communication time is mainly determined by the model size M and η, the linear slope of the algorithm. It implies that given any model, the communication time is similar with



(a) Algorithmic scalability



(b) Weak scalability



(c) Strong scalability

Figure 6.10: Algorithmic scalability (a), weak scalability with fixed per node 375 KB model size (b) and strong scalabililty with fixed problem size of 32 MB model size (c) normalized to 16-node performance of RING (β: with wide network interface, δ: with both wide network interface and message-based flow control).

different network scales. We also run strong scalability for the DNN workloads. Both RING and MULTITREE-δ show all-reduce performance variations within 9% across different network scales while MXNETTREE-β shows performance variations up to 18%.

6.5 Discussions

6.5.1 Broader Applications

Although MULTITREE is co-designed for data parallelism, it can also support hybrid-parallel inference and training. The message-based flow control can be used to improve bandwidth efficiency in both cases. It can also be adapted for indirect networks such as networks involving NVSwitch with minor modification in flit encoding. In addition, MULTITREE can speedup data-parallel components in a hybrid approach. When the model of parallelism and DNN workload are determined, MULTITREE runs for the nodes that involve all-reduce communication. The all-gather trees can also easily support all-to-all collective in recent DNN workloads such as DLRM [163]. For networks with heterogeneous link bandwidths, the network topology graph can be represented as a multigraph where each edge is a unit of bandwidth and MULTITREE applies properly.

6.5.2 Limitations

Although a tree structure has a logarithmic height, the current co-design cannot further improve its scalability as shown in Figures 6.10a and 6.10b. The limitation is inherent in the network topology. As the network scales up, the number of trees also increases linearly. Theoretically, in a balance tree, the amount of leaf nodes participating in communication is half of the tree nodes, which is also linear to the size of the network. In an extreme case, the total communication pairs are exponential to the size of the network. However, with a fixed node degree, the available links only increase linearly, which cannot satisfy the exponential need of the algorithm. In hardware, the number of router ports is increased by $\frac{3}{5}$ to support the degree of tree nodes. The on-chip interconnect area increases from 0.4 mm^2 to 0.66 mm^2 in 45-nm node.

6.5.3 Opportunities

MULTITREE demonstrates the effectiveness of algorithm and architecture co-design for communication acceleration by exploiting network topology and big message size of all-reduce in distributed deep learning. This study also reveals more co-design opportunities with topology, such as topology design for data-parallel training [126] or reconfigurable interconnects for more complex hybrid-parallel deep learning and asynchronous all-reduce. In addition, reducing the number of trees by trading bandwidth and latency as an attempt in recent work [125] can be further explored in the algorithm design. We leave these aspects for future work.

6.6 Summary

In this chapter, we co-design communication algorithm and interconnection architecture to support efficient and scalable all-reduce operation. We observe significant inefficiency in link utilization and long latency in commonly used Ring All-Reduce. The inefficiency and long latency motivates MULTITREE, a scalable all-reduce algorithm that is applicable to various topologies. MULTITREE couples tree construction and message scheduling with topology and global link utilization awareness from roots in a top-down approach. It leverages the insight that tree levels closer to the roots are more sparse and tree levels closer to the leaves are denser. As a result, MULTITREE effectively moves more communication closer to the roots to make communication closer to the leaves sparse so that there is nearly no contention on link resources for concurrent reduction trees. Moreover, we specialize the interconnection network according to the proposed communication algorithm. We also simplify the flow control and arbitration to exploit the characteristics of large gradients in all-reduce operations. The codesign contributes to 2.5× and 1.7× speedup over RING and state-of-the-art approach, respectively.

7. CONCLUSIONS

The primary contribution of this dissertation is the development of communication specialization for the multifaceted requirements in heterogeneous architectures. By specializing communication for diverse application domains and architectures, interconnection networks are able to achieve extremely power-/energy-efficient data movement and scalable performance, keeping in pace with computation specialization for efficient and sustainable performance improvement.

7.1 Dissertation Summary

In this dissertation, we investigate router power-gating mechanisms to design low-power network-on-chip for general-purpose workloads and chip multiprocessors. We develop a power-gating policy (FLOV) consisting of multiple modes trading off between throughput performance and power saving. The FLOV policy takes a voting approach that is adaptable to the local and global traffic status by collecting votes from routers on the same dimensions. The accumulative votes are used to decide the power-gating mode either towards more aggressive power saving or better performance. Such a cooperative approach among routers achieves a balance that considers both local optimum and global desire. In addition, we augment the router microarchitecture with bypassing channels with co-designed routing algorithms to achieve zero overhead for latency and negligible throughput degradation. The bypassing mechanism, called *Fly-Over*, avoids detouring around and waking up power-gated routers on the path by flying over them. This also eliminates the router pipeline and contributes even lower latency. This holistic interconnect solution not only saves power, but also achieves better performance at low traffic loads. Our full system simulation results show that this design saves 31% and 20% more power compared to no power-gating and state-of-the-art, respectively. It also improves the performance by 3.9% at low network loads.

We also propose approximate communication for emerging applications to improve the effective network throughput. APPROX-NOC, an NoC data approximation framework is developed to approximate similar data by exploiting data value locality to increase compression effectiveness,

thereby reducing bandwidth requirement for data transmission. To reduce the overhead of approximation on the critical path, we design an approximation engine that includes a lightweight error computation logic unit, which approximates value using only fast shifting and concatenation operations. The datapath and control can be used for both integer and complex IEEE 754 floating point values. And its conservative error tolerance makes it work in synergy with other approximation layers without exceeding the error budget. Moreover, this module can be used in the fashion of plug and play for any underlying NoC data compression mechanisms. We present two microarchitecture implementations by integrating APPROX-NOC with two compression techniques, one in a tightly-coupled way and the other in a plug-and-play manner. The evaluation results show that APPROX-NOC can improve the network throughput by up to 60% compared to precise compression, providing promising opportunities to reduce data movements in big data applications.

To further reduce data movement for big data workloads, we explore near-data processing paradigm with in-network computing and propose *Active-Routing* that offloads computations to the scalable memory network for in-network dataflow processing. *Active-Routing* targets program kernels that compute pure reduction or aggregate over intermediate results of arithmetic operators on a myriad of data, such as sum reduction or dot product. Computations are offloaded to the memory network nodes to perform locally or closely to take advantage of the massive bandwidth and parallelism in memory. Meanwhile, it dynamically builds topology-oblivious dataflow trees and leverages the network concurrency to optimize reduction of data along the route. Therefore, it reduces the data movement of source operands and eliminates the contention effect or thrashing on cache caused by read and write operations on rarely reused data. *Active-Routing* can be integrated seamlessly with modern processors by instruction set architecture extension with minimum changes of processor architecture. Furthermore, it works in synergy with virtual memory and coherence protocol, making it attractive for near-term adoption in products. The simulated system shows up to 7$\times$ speedup with an average of 60% performance improvement and reduces energy-delay product by 80% on average across various benchmarks compared to the state-of-the-art processing-in-memory architecture.

This dissertation also seeks to accelerate communication for distributed deep learning by co-designing all-reduce algorithm, application and the interconnect architecture. The optimizations are based on the observations of inefficiency in link utilization in the widely used Ring All-Reduce algorithm and the lack of hardware support for tree-based algorithms. Therefore, we design MULTITREE All-Reduce algorithm that couples tree construction and message scheduling with awareness of topology and link utilization. It builds trees in a top-down approach from roots by leveraging the insight that tree levels closer to the roots are more sparse and tree levels closer to the leaves are denser. Consequently, MULTITREE effectively moves more communication closer to the roots to make communication closer to the leaves sparse in order to achieve contention-free tree all-reduce. Moreover, we specialize the interconnection network with heterogeneous bandwidth provisioning of injection/ejection and network channels required by the algorithm. We also simplify the flow control to exploit the characteristics of large gradients in deep neural networks. Our evaluations using state-of-the-art DNN models show that MULTITREE improves scalability by a factor of 4, and achieves 2.5× performance speedup compared to Ring All-Reduce.

7.2 Future Directions

As heterogeneous architecture evolve with more specialization for computation, there are distinct communication characteristics and requirements for different applications. With the development of deep learning, more and more larger deep neural network models are developed and deployed on accelerator pods. These giant models demand much more memory and higher bandwidth. Data movement and communication will soon be the bottleneck to supply data for efficient specialized computation. Therefore, data movement minimization and communication acceleration is imperative for the near future. In this section, we discuss three optimization opportunities of specialized interconnection network design for deep learning towards efficient data movement and communication acceleration.

7.2.1 Approximate Communication

Deep neural network fundamentally is an approximation of the function for particular classification tasks. It trains the model to learn the relation between the statistical distribution and representation of the input data to the desired output space. With this approximation nature in design, data movement of input and weights can also be statistically adjusted as a projection or quantization of the original data. Therefore, approximate data movement can be treated as such a method towards efficient communication. The data can be approximated in a way that is within statistical bounds, for example, dropping values with some threshold, similar to error threshold or budget control. The approximate data can be further co-designed with compression techniques to increase the compression rate so as to achieve bandwidth saving. Such an approximate representation and distribution should be included in the close loop of model training to also learn the statistical representation adjusted by approximation. It leaves as a question how much the approximation can be learned without tempering too much the original data distribution. Including approximation in the training loop will allow us to understand the potential of approximate communication in deep learning and its effect on the model accuracy.

7.2.2 In-Network Computing

As distributed deep learning deployments become popular due to the demanding requirements of computation, memory and bandwidth of large DNN models, data are moved across the network for simple computation. The receivers need to reserve memory space to store these temporal data for the simple computation such as reduction. Especially when multiple data streams go to the same node simultaneously, as is the case in all-reduce, the memory requirement and end node bandwidth is even higher. This poses a question that can these simple operations be computed when flowing data in the network to consume them while moving them? If there is an effective way to do so, this introduces a new endeavor to not just reduce the data movement, but also reduce the bandwidth and memory requirement. The reduction for all-reduce is a typical application when data streams to be reduced come to the same switch, they can be reduced at the meeting point so

that the incoming multiple streams become a single merged stream. Other opportunities in forward and backward passes are awaiting more exploration.

7.2.3 Topology Specialization

Most distributed deep learning algorithms have static communication and computation patterns after scheduled for deployment. In both inference and training, both computation and communication phases last for a long time for both data-parallelism and model-parallelism. For a particular distributed placement of a DNN on accelerator grids, it will be static until it finishes the job. We may be able to exploit this simplicity to extract benefits for communication acceleration. After the placement of a DNN model, the communications between different nodes are determined based on the neural network architecture. With this prior knowledge, it is possible to derive the favorable topology to match the compute graph topology of the deployed model. Also, dedicated resources and connections can be also configured to accelerate the communication between these nodes. It remains as a question how flexible it could be to support diverse placement and deployment strategies. It is worth a study to understand how the reconfiguration of the topology and interaction of DNN placement could help communication acceleration of various DNNs.

REFERENCES

[1] R. Boyapati, J. Huang, N. Wang, K. H. Kim, K. H. Yum, and E. J. Kim, "Fly-Over: A Light-Weight Distributed Power-Gating Mechanism For Energy-Efficient Networks-on-Chip," in *Proceedings of the 2017 International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 708–717, May 2017.

[2] R. Boyapati, J. Huang, P. Majumder, K. H. Kim, and E. J. Kim, "Approx-NoC: A Data Approximation Framework for Network-On-Chip Architectures," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA-44)*, pp. 666–677, July 2017.

[3] J. Huang, R. R. Puli, P. Majumder, S. Kim, R. Boyapati, K. H. Kim, and E. J. Kim, "Active-Routing: Compute on the Way for Near-Data Processing," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA-25)*, pp. 674–686, February 2019.

[4] 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," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[5] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, April 1965.

[6] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] A. Samih, R. Wang, A. Krishna, C. Maciocco, C. Tai, and Y. Solihin, "Energy-Efficient Interconnect via Router Parking," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 508–519, 2013.

[8] L. Chen and T. M. Pinkston, "NoRD: Node-Router Decoupling for Effective Power-gating of On-Chip Routers," in *Proceedings of the 2012 45th Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO-45, pp. 270–281, 2012.

[9] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–55, 2015.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *International Conference on Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

[11] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. R. Iyer, M. S. Yousif, and C. R. Das, "Performance and Power Optimization Through Data Compression in Network-on-Chip Architectures," in *Proceedings of the 14th International Conference on High-Performance Computer Architecture (HPCA-14)*, pp. 215–225, 2008.

[12] Y. Jin, K. H. Yum, and E. J. Kim, "Adaptive Data Compression for High-performance Low-power On-chip Networks," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, pp. 354–363, 2008.

[13] P. Zhou, B. Zhao, Y. Du, Y. Xu, Y. Zhang, J. Yang, and L. Zhao, "Frequent Value Compression in Packet-based NoC Architectures," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*, pp. 13–18, 2009.

[14] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *2017 IEEE International Symposium on High Performance Computer Architecture, (HPCA)*, pp. 481–492, 2017.

[15] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache Automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pp. 259–272, 2017.

[16] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks," in

Proceedings of the 45th Annual International Symposium on Computer Architecture, ISCA '18, pp. 383–396, 2018.

[17] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit Prudent In-Cache Acceleration of Deep Convolutional Neural Networks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 81–93, Feb 2019.

[18] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *International Symposium on Computer Architecture (ISCA)*, pp. 336–348, 2015.

[19] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *International Symposium on Computer Architecture (ISCA)*, pp. 105–117, 2015.

[20] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *International Symposium on Computer Architecture (ISCA)*, pp. 204–216, 2016.

[21] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 457–468, 2017.

[22] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 639–651, 2017.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–444, 5 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, Curran Associates, Inc., 2017.

[26] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A Guide to Deep Learning in Healthcare," *Nature Medicine*, vol. 25, no. 1, p. 24, 2019.

[27] J. Hestness, N. Ardalani, and G. Diamos, "Beyond Human-level Accuracy: Computational Challenges in Deep Learning," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pp. 1–14, 2019.

[28] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch Sgd: Training ImageNet in 1 Hour," *CoRR*, vol. abs/1706.02677, 2017.

[29] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[30] J. Huang, M. Patwary, and G. Diamos, "Coloring Big Graphs with AlphaGoZero," *CoRR*, vol. abs/1902.10162, 2019.

[31] P. Patarasuk and X. Yuan, "Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[32] P. Sanders, J. Speck, and J. L. TrÃff, "Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan," *Parallel Computing*, vol. 35, no. 12, pp. 581–594, 2009. Selected papers from the 14th European PVM/MPI Users Group Meeting.

[33] C. Yang, "Tree-based Allreduce Communication on MXNet," tech. rep., University of California, Davis, 2018. [Online; accessed 26-August-2019].

[34] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT – A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *International Symposium on Networks on Chip (NoCS)*, pp. 201–210, 2012.

[35] A. Guzhva, S. Dolenko, and I. Persiantsev, "Multifold Acceleration of Neural Network Computations Using GPU," in *Proceedings of the 19th International Conference on Artificial Neural Networks: Part I (ICANN 2009)*, pp. 373–380, 2009.

[36] M. Creel and M. Zubair, "High Performance Implementation of an Econometrics and Financial Application on GPUs," in *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SCC 2012)*, pp. 1147–1153, 2012.

[37] O. A. Aguilar and J. C. Huegel, "Inverse Kinematics Solution for Robotic Manipulators Using a CUDA-Based Parallel Genetic Algorithm," in *Proceedings of the 10th Mexican International Conference on Advances in Artificial Intelligence - Volume Part I (MICAI 2011)*, pp. 490–503, 2011.

[38] K. Anjan and T. M. Pinkston, "An Efficient, Fully Adaptive Deadlock Recovery Scheme: DISHA," in *proceedings of the 22nd annual international symposium on computer Architecture*, pp. 201–210, 1995.

[39] J. Duato, "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1320–1331, 1993.

[40] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[41] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks (1976)*, vol. 3, no. 4, pp. 267–286, 1979.

[42] W. J. Dally and C. L. Seitz, "The torus routing chip," *Distributed computing*, vol. 1, no. 4, pp. 187–196, 1986.

[43] L.-S. Peh and W. J. Dally, "A Delay Model and Speculative Architecture for Pipelined Routers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 255–266, IEEE, 2001.

[44] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[45] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. Van Der Wijngaart, "A 48-Core IA-32 Processor in 45nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, 2011.

[46] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, 2007.

[47] L. Chen and T. M. Pinkston, "NoRD: Node-Router Decoupling for Effective Power-Gating of On-Chip Routers," in *International Symposium on Microarchitecture (MICRO)*, pp. 270–281, IEEE Computer Society, 2012.

[48] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT – A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *International Symposium on Networks on Chip (NoCS)*, pp. 201–210, IEEE, 2012.

[49] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram, "A Case for Guarded Power Gating for Multi-Core Processors," in *International Symposium on High Performance Com-*

puter Architecture (HPCA), pp. 291–300, IEEE, 2011.

[50] R. Kumar, A. Martínez, and A. González, "Dynamic Selective Devectorization for Efficient Power Gating of SIMD Units in a HW/SW Co-Designed Environment," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 81–88, IEEE, 2013.

[51] V. Soteriou and L.-S. Peh, "Design-Space Exploration of Power-Aware On/Off Interconnection Networks," in *International Conference on Computer Design (ICCD)*, pp. 510–517, IEEE, 2004.

[52] H. Matsutani, M. Koibuchi, D. Ikebuchi, K. Usami, H. Nakamura, and H. Amano, "Ultra Fine-Grained Run-Time Power Gating of On-Chip Routers for CMPs," in *International Symposium on Networks-on-Chip (NOCS)*, pp. 61–68, IEEE, 2010.

[53] G. Kim, J. Kim, and S. Yoo, "Flexibuffer: Reducing Leakage Power in On-Chip Network Routers," in *Design Automation Conference (DAC)*, pp. 936–941, IEEE, 2011.

[54] R. Parikh, R. Das, and V. Bertacco, "Power-Aware NoCs through Routing and Topology Reconfiguration," in *Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2014.

[55] E. J. Kim, K. H. Yum, G. M. Link, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, M. Yousif, and C. R. Das, "Energy Optimization Techniques in Cluster Interconnects," in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 459–464, ACM, 2003.

[56] H. Matsutani, M. Koibuchi, D. Wang, and H. Amano, "Run-Time Power Gating of On-Chip Routers Using Look-Ahead Routing," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 55–60, IEEE Computer Society Press, 2008.

[57] L. Chen, D. Zhu, M. Pedram, and T. M. Pinkston, "Power Punch: Towards Non-Blocking Power-Gating of NoC Routers," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–12, IEEE, 2015.

[58] J. Zhan, Y. Xie, and G. Sun, "NoC-Sprinting: Interconnect for Fine-Grained Sprinting in the Dark Silicon Era," in *Proceedings of the 51st Annual Design Automation Conference (DAC)*, pp. 1–6, ACM, 2014.

[59] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy Proportional Multiple Network-on-Chip," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 320–331, 2013.

[60] A. Samih, R. Wang, A. Krishna, C. Maciocco, C. Tai, and Y. Solihin, "Energy-Efficient Interconnect via Router Parking," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 508–519, IEEE, 2013.

[61] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express Virtual Channels: Towards the Ideal Interconnection Fabric," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 150–161, 2007.

[62] A. Kodi, A. Louri, and J. Wang, "Design of Energy-Efficient Channel Buffers with Router Bypassing for Network-on-Chips (NoCs)," in *International Symposium on Quality Electronic Design (ISQED)*, pp. 826–832, IEEE, 2009.

[63] U. Y. Ogras and R. Marculescu, "Application-Specific Network-on-Chip Architecture Customization via Long-Range Link Insertion," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 246–253, IEEE, 2005.

[64] U. Y. Ogras and R. Marculescu, ""It's a Small World After All": NoC Performance Optimization via Long-Range Link Insertion," *IEEE Transaction on Very Large Scale Integration Systems*, vol. 14, no. 7, pp. 693–706, 2006.

[65] S. J. Hollis, C. Jackson, P. Bogdan, and R. Marculescu, "Exploiting Emergence in On-Chip Interconnects," *IEEE Transactions on Computers*, vol. 63, no. 3, pp. 570–582, 2014.

[66] H. Zheng and A. Louri, "EZ-Pass: An Energy & Performance-Efficient Power-Gating Router Architecture for Scalable Nocs," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 88–91, 2017.

[67] H. Farrokhbakht, H. M. Kamali, and N. E. Jerger, "Muffin: Minimally-Buffered Zero-Delay Power-Gating Technique in On-Chip Routers," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[68] H. Farrokhbakht, M. Taram, B. Khaleghi, and S. Hessabi, "Toot: an efficient and scalable power-gating method for noc routers," in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pp. 1–8, IEEE, 2016.

[69] H. Farrokhbakht, H. M. Kamali, N. E. Jerger, and S. Hessabi, "Sponge: A scalable pivot-based on/off gating engine for reducing static power in noc routers," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[70] Y. Xue and P. Bogdan, "Improving noc performance under spatio-temporal variability by runtime reconfiguration: a general mathematical framework," in *2016 tenth IEEE/ACM international symposium on networks-on-chip (NOCS)*, pp. 1–8, IEEE, 2016.

[71] Z. Qian, P. Bogdan, G. Wei, C.-Y. Tsui, and R. Marculescu, "A traffic-aware adaptive routing algorithm on a highly reconfigurable network-on-chip architecture," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 161–170, 2012.

[72] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch, "Methods for fault tolerance in networks-on-chip," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–38, 2013.

[73] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pp. 164–174, 2011.

[74] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate Storage in Solid-State Memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, pp. 25–36, 2013.

[75] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A Programmer-Guided Compiler Framework for Practical Approximate Computing," *University of Washington Technical Report UW-CSE-15-01*, vol. 1, 2015.

[76] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture Support for Disciplined Approximate Programming," *SIGPLAN Not.*, vol. 47, no. 4, pp. 301–312, 2012.

[77] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, pp. 449–460, 2012.

[78] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *Proceeedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA-21)*, pp. 603–614, 2015.

[79] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose Code Acceleration with Limited-precision Analog Computation," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA-41)*, pp. 505–516, 2014.

[80] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural Acceleration for GPU Throughput Processors," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 482–493, 2015.

[81] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*, pp. 213–224, 2011.

[82] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelganger: A Cache for Approximate Computing," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 50–61, 2015.

[83] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. R. Iyer, M. S. Yousif, and C. R. Das, "Performance and Power Optimization Through Data Compression in Network-on-Chip Architectures," in *Proceedings of the 14th International Conference on High-Performance Computer Architecture (HPCA-14)*, pp. 215–225, 2008.

[84] P. Zhou, B. Zhao, Y. Du, Y. Xu, Y. Zhang, J. Yang, and L. Zhao, "Frequent Value Compression in Packet-based NoC Architectures," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*, pp. 13–18, 2009.

[85] J. Zhan, M. Poremba, Y. Xu, and Y. Xie, "Leveraging Delta Compression for End-to-End Memory Access in NoC Based Multicores," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 586–591, Jan 2014.

[86] Y. Jin, K. H. Yum, and E. J. Kim, "Adaptive Data Compression for High-performance Low-power On-chip Networks," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, pp. 354–363, 2008.

[87] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *International Symposium on Computer Architecture (ISCA)*, pp. 453–464, 2008.

[88] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips 23 Symposium (HCS)*, pp. 1–24, IEEE, 2011.

[89] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 432–433, IEEE, 2014.

[90] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *International Symposium on Computer Architecture (ISCA)*, pp. 336–348, IEEE, 2015.

[91] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 457–468, 2017.

[92] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating Linked-List Traversal through Near-Data Processing," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 113–124, IEEE, 2016.

[93] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *International Symposium on Computer Architecture (ISCA)*, pp. 105–117, IEEE, 2015.

[94] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 639–651, 2017.

[95] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 145–156, IEEE Press, 2013.

[96] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, pp. 481–492, 2017.

[97] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair, "Data Access Optimization in a Processing-in-Memory System," in *International Conference on Computing Frontiers (CF)*, p. 6, ACM, 2015.

[98] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory + Logic Devices on MapReduce Workloads," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 190–200, 2014.

[99] A. F. Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 283–295, 2015.

[100] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[101] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *International Symposium on Computer Architecture (ISCA)*, pp. 204–216, IEEE Press, 2016.

[102] J. Ahn, S. Yoo, and K. Choi, "AIM: Energy-Efficient Aggregation inside the Memory Hierarchy," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 34, 2016.

[103] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, p. 17, 2015.

[104] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1–14, ACM, 2018.

[105] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, *et al.*, "Google workloads for consumer devices: mitigating

data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 316–331, ACM, 2018.

[106] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *International Symposium on Computer Architecture (ISCA)*, pp. 256–266, IEEE, 1992.

[107] G. F. Pfister and V. A. Norton, ""Hot Spot" Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. c-34, no. 10, pp. 943–948, 1985.

[108] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 795–809, ACM, 2017.

[109] S. Ma, N. E. Jerger, and Z. Wang, "Supporting Efficient Collective Communication in NoCs," in *High Performance Computer Architecture (HPCA)*, pp. 1–12, IEEE, 2012.

[110] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, vol. c-32, no. 2, pp. 175–189, 1983.

[111] D. K. Panda, "Global Reduction in Wormhole k-ary n-cube Networks with Multidestination Exchange Worms," in *International Parallel Processing Symposium (IPPS)*, pp. 652–659, 1995.

[112] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network and Message Unit," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–10, IEEE, 2011.

[113] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 461–475, ACM, 2018.

[114] R. Mayer and H.-A. Jacobsen, "Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques and Tools," *CoRR*, vol. abs/1903.11314, 2019.

[115] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis," *ACM Comput. Surv.*, vol. 52, pp. 65:1–65:43, Aug. 2019.

[116] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, 2014.

[117] A. Gibiansky and J. Hestness, "baidu-research/tensorflow-allreduce." https://github.com/baidu-research/tensorflow-allreduce, 2017. [Online; accessed 4-November-2019].

[118] A. Gibiansky, "Bringing HPC Techniques to Deep Learning." http://andrew.gibiansky.com, 2017. [Online; accessed 24-November-2019].

[119] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter, "BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy," in *SysML 2019*, 2019.

[120] L. Wang, M. Li, E. Liberty, and A. J. Smola, "Optimal Message Scheduling for Aggregation," in *SysML 2018*, 2018.

[121] NVIDIA, "NVIDIA Collective Communication Library (NCCL)." https://developer.nvidia.com/nccl, 2017. [Online; accessed 4-November-2019].

[122] A. Sergeev and M. D. Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *CoRR*, vol. abs/1802.05799, 2018.

[123] S. Jeaugey, "Massively Scale Your Deep Learning Training with NCCL 2.4." https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/, Feburary 2019. [Online; accessed 6-February-2020].

[124] L. Luo, P. West, A. Krishnamurthy, L. Ceze, and J. Nelson, "PLink: Discovering and Exploiting Datacenter Network Locaility for Efficient Cloud-based Distributed Training," in *MLSys 2020*, 2020.

[125] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and Generic Collectives for Distributed ML," in *MLSys 2020*, 2020.

[126] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng, Y. Guo, X. Jiang, L. Tang, Y. Du, Y. Zhang, P. Pan, and Y. Xie, "EFLOPS: Algorithm and System Co-design for a High Performance Distributed Training Platform," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture6(HPCA-25)*, pp. 610–622, February 2020.

[127] R. Boyapati, J. Huang, N. Wang, K. H. Kim, K. H. Yum, and E. J. Kim, "Fly-over: A light-weight distributed power-gating mechanism for energy-efficient networks-on-chip," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 708–717, IEEE, 2017.

[128] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing," in *International Symposium on Computer Architecture (ISCA)*, p. 278287, 1992.

[129] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J.-H. Kim, and W. J. Dally, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 86–96, IEEE, 2013.

[130] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and

D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, 2011.

[131] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[132] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelganger: A Cache for Approximate Computing," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 50–61, 2015.

[133] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An Online Quality Management System for Approximate Computing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA-42)*, pp. 554–566, 2015.

[134] A. R. Alameldeen and D. A. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[135] B. Agrawal and T. Sherwood, "Ternary CAM Power and Delay Model: Extensions and Uses," *IEEE Trans. Very Large Scale Integr. Syst.*, pp. 554–564, 2008.

[136] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pp. 190–200, 2005.

[137] B. Lucia, "A Coherent Multiprocessor Cache Simulator Based on the SuperESCalar Cache Model." https://github.com/blucia0a/MultiCacheSim, 2013.

[138] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[139] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, pp. 127–139, 2014.

[140] D. A. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *Proceedings of the 12th International Conference on High Performance Computing (HiPC 2005)*, pp. 465–476, 2005.

[141] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection." Available from http://snap.stanford.edu/data, June 2014.

[142] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs with Loop Perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*, pp. 124–134, 2011.

[143] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *International Symposium on Microarchitecture (MICRO)*, pp. 3–14, 2007.

[144] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A Manycore Simulator with Application-Level+ Simulation and Detailed Microarchitecture Modeling," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 74–85, April 2013.

[145] D. I. Jeon and K. S. Chung, "CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube," *IEEE Computer Architecture Letters*, vol. 16, pp. 10–13, Jan 2017.

[146] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 190–200, ACM, 2005.

[147] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G. H. Loh, "There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes," in *International Symposium on Computer Architecture (ISCA)*, pp. 678–690, ACM, 2017.

[148] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. Patwary, M. Ali, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[149] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11, IEEE Computer Society, 2010.

[150] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–55, IEEE Computer Society, 2015.

[151] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," tech. rep., University of Illinois at Urbana-Champaign, March 2012.

[152] R. Rabenseifner, "Optimization of Collective Reduction Operations," in *International Conference on Computational Science*, pp. 1–9, Springer, 2004.

[153] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN Accelerator Simulator," *CoRR*, vol. abs/1811.02883, 2018.

[154] Google Cloud, "Cloud TPU." https://cloud.google.com/tpu. [Online; accessed 1-June-2019].

[155] Microsoft, "Project Catapult." https://www.tacc.utexas.edu/systems/catapulta. [Online; accessed 4-November-2019].

[156] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J.-H. Kim, and W. J. Dally, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 86–96, 2013.

[157] K. Bergman and S. Rumley, "Optical Interconnects for Energy Efficient HPC," *Supercomputing Conference 2016 Working Group Presentations*.

[158] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the Game of Go without Human Knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[159] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-Cnn: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in Neural Information Processing Systems*, pp. 91–99, Curran Associates, Inc., 2015.

[160] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[161] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural Collaborative Filtering," in *Proceedings of the 26th International Conference on World Wide Web*, pp. 173–182, 2017.

[162] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and Ł. Kaiser, "Universal Transformers," *CoRR*, vol. abs/1807.03819, 2018.

[163] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *CoRR*, vol. abs/1906.00091, 2019.