# A CROSS-STACK, NETWORK-CENTRIC ARCHITECTURAL DESIGN FOR NEXT-GENERATION DATACENTERS

BY

MOHAMMAD ALIAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

# ABSTRACT

This thesis proposes a full-stack, cross-layer datacenter architecture based on in-network computing and near-memory processing paradigms. The proposed datacenter architecture is built atop two principles: (1) utilizing commodity, off-the-shelf hardware (*i.e.,* processor, DRAM, and network devices) with minimal changes to their architecture, and (2) providing a standard interface to the programmers for using the novel hardware. More specifically, the proposed datacenter architecture enables a smart network adapter to collectively compress/decompress data exchange between distributed DNN training nodes and assist the operating system in performing aggressive processor power management. It also deploys specialized memory modules in the servers, capable of performing general-purpose computation and network connectivity.

This thesis unlocks the potentials of hardware and operating system co-design in architecting application-transparent, near-data processing hardware for improving datacenter's performance, energy efficiency, and scalability. We evaluate the proposed datacenter architecture using a combination of full-system simulation, FPGA prototyping, and real-system experiments.

*To my parents, for their indefinite love and support.*

# ACKNOWLEDGMENTS

I cannot believe how fast these almost seven years of graduate school passed. So far, it has been the most challenging and most fruitful time in my life. This journey was impossible without Nam Sung Kim's guidance, my teachers' instructions, my collaborators' help, my friends' company, and my family's support. I want to take some time to thank every single one of them.

First and foremost, I would like to thank Nam Sung for his unequivocal support and guidance throughout my graduate school career. Since the beginning of the graduate school, I have been in Nam Sung's group. Nam Sung showed me the path instead of the solution and let me walk through it myself. I have to acknowledge that it was indeed hard, but rewarding. He provided me advice that helped me become a better scholar and writer. Thanks to Nam, I never had to worry about funding during my graduate school, and all my focus was on the academic affairs.

I would like to thank Wen-mei Hwu, Josep Torrelas, Rakesh Kumar, and Marc Snir for serving in my Ph.D. committee and their tremendous help in my academic matters. Hadi Esmaeilzadeh offered me indispensable advice and support in the last few months of my Ph.D.! Thank you, Hadi.

I want to acknowledge several of my colleagues and collaborators that surely, I could not finish this thesis without their help. They played a vital role in all the projects that I worked together with them. Thank you Daehoon Kim, Ren Wang, Myoungsoo Jung, Hadi Asghari-Moghaddam, Ahmed Abulila, Seung Won Min, Ashutosh Dhar, Lokesh Jindal, Umur Darbaz, Youjie Li, Yifan Yuan, Jie Zhang, and Krishna Parasuram Srinivasan.

I couldn't pass this milestone in my life if a lot of wonderful friends did not surround me. My friends in Madison and Chambana helped me to forget that I am thousands of miles away from home. I will miss every second I spent with them having little parties, watching random videos, doing barbeques, and more importantly, playing volleyball!

Lastly, I would like to thank my family. My dad was always my advocate for taking risks and having new adventures in life. I always said my dad could be another Elon Musk if he was in the right time and right place. Unfortunately, he did not become an Elon Musk in reality, but he is always a hero for me. My mom is the definition of love. I never can return an epsilon of the love she gave me. I am incredibly grateful for all the scarifies that my parents did for me; the least was not seeing their boy for almost seven years (thanks to the travel ban)! I am grateful for having a smart, supportive, and reliable brother like Alireza. Even if Alireza is thousands of miles away from me, his presence helps me to feel I have someone out there that is always looking after me. Just like my brother, my sisters are always available whenever I need them. Thank you, Vida, Nafiseh, and Maryam. You are amazing.

# TABLE OF CONTENTS

# LIST OF ABBREVIATIONS

| | |
|---|---|
| CPU | Central Processing Unit |
| DIMM | Dual Inline Memory Module |
| DNN | Deep Neural Network |
| DRAM | Dynamic Random Access Memory |
| FPGA | Field Programmable Gate Array |
| GPU | Graphics Processing Unit |
| INCEPTIONN | In-Network Computing to Exchange and Process Training Information of Neural Networks |
| LSB | Least Significant Bit |
| MCN | Memory Channel Network |
| MSB | Most Significant Bit |
| NetDIMM | Network-Attached DIMM |
| NCAP | Network-driven, packet Context-Aware Power management for Client-server architecture |
| NIC | Network Interface Card |
| OLDI | On-Line, Data-Intensive |
| OS | Operating System |
| SLO | Service Level Objective |
| TPU | Tensor Processing Unit |

# CHAPTER 1

# INTRODUCTION

## 1.1 Introduction

The Internet is about to celebrate its fortieth birthday.[1] In the past four decades, the Internet has transformed many businesses and created many more. Online services are part of our daily lives, and the Internet has become as essential as other household utilities like water, electricity, and telephone services. While writing this thesis, the world is wrestling with the COVID-19 pandemic, which sadly has taken many lives, and more is counting. From the technology point of view, the COVID-19 pandemic has accelerated the departure from conducting in-person businesses by relying on the Internet. Video conference meetings and virtual classrooms are the new norms in universities, and enterprise is demanding immediate support for conducting their business online. Even construction companies are looking for ways to reduce the on-site visits of their engineers by making 3D models of the construction sites and feeding them to their engineers. The question of interest here is, how are all these services delivered to the users? All we know, the Internet services, from large-scale web services to small enterprise applications, are powered on by datacenters.

Although it is possible to run a wide range of Internet services on user devices (*e.g.,* desktops, smartphones, or laptop computers), server-side computing (*i.e.,* running applications on a datacenter) offers several advantages that makes it attractive, even for conventional applications. Sharing hardware resources in the datacenters reduces per user computation cost. Moreover, server-side computing hugely simplifies software deployment and updates. Instead of reaching out to millions of user devices, a datacenter

[1]In 1983, ARPAnet started using TCP/IP, which is the backbone of the modern Internet to this date

deployment of an application can be updated once and without any interruption in the user experience. Moreover, the computation model of many applications, such as social networks, demands a centralized database to reduce the amount of data exchange between user devices and the database. Similar to the software advantages, since a datacenter is often managed by one company, hardware upgrades and accelerators' deployment are cheaper. They can be done without upgrading the hardware of millions of user devices.

Unquestionably, datacenters are playing a vital role not only in our digital world but also in our safety, productivity, and, more importantly, helping to perform environmentally friendly computation. Although datacenters consume hundreds of megawatts of power, the watts per operation inside datacenters are hugely lower than that of a desktop server. However, the end of Dennard scaling, the slowdown of Moore's law, and explosion in data volumes make it extremely challenging to design datacenters that can supply the computation demand for future Internet-scale and cloud workloads. The current datacenters resemble a collection of individual servers connected over a scale-out network fabric. We should embrace the fact that sooner or later, the performance of individual servers stops scaling. Then the only path forward would be to scale out by increase the number of servers. However, not only the networking cost of connecting more and more servers increases exponentially, the communication overhead of distributing applications on more servers can diminish the effectiveness of having more servers. The main message of this thesis is to promote a datacenter architecture that resembles a large-scale computer system (*i.e.,* a warehouse-scale computer [1]) instead of a collection of individual servers. In this thesis, we propose to *blur the boundaries between processors, memory, and networking by performing computation not only on the CPU but also inside the network and memory of datacenter servers.* The proposed architectures consider *full-stack, hardware, and operating system co-optimization.*

The applications that run on an enterprise datacenter can be broadly classified into two categories: (i) online, latency-critical applications (*i.e.,* "online applications" such as web serving, web search, social media, and online gaming), and (ii) data-intensive, throughput applications (*i.e.,* "throughput applications" such as data analytics, classification, and graph processing applications). The performance metric for online applications is to perform a task within a deadline, and as long as the task (*e.g.,* a web request) is processed within that deadline, we have satisfactory performance. On the other hand, the amount of work performed in unit time is all that matters for throughput applications. Next, we discuss the implications of running online and throughput applications in datacenters.

The deadline for processing an online request is perceived as a metric for measuring user experience satisfaction. It is often defined as the high percentile ranks (*e.g.,* 95th or 99th percentile) of the end-to-end response time.[2] Online applications often operate at a massive scale where a single request can activate thousands of servers. Operating at this scale as well as having tight SLO requirements make online services extremely sensitive to hardware and software performance variations. Our studies show that processor power management and network stack processing are the two important sources of performance variations in online servers. In fact, because of the performance variations and the reactive nature of software-managed power management policies, online service providers avoid employing aggressive processor power management policies on online servers [2, 3]. That is, online servers are always kept on at their highest performance level regardless of their current load level. The lack of processor power management leads to significant energy wastes when the online servers are underutilized.

The Internet network service model is based on a best-effort model with soft states. This service model implies that when a packet leaves a server's boundaries, there is no guarantee when the packet arrives at the destination server. The datacenter network architecture evolved around the same principles. For example, Ethernet, which is the predominant network technology inside datacenters, does not even guarantee the delivery of a packet. Even though lossless network technologies such as Infiniband [4] or lossless Ethernet [5] exist, the packet transmission and reception are still software managed and vulnerable to software performance variability. Until a few years ago, this variability in the network latency was not a concern due to the huge performance discrepancy between the network and CPU. However, in the past decade, while the performance of the CPU (as well as

[2]End-to-end response time is defined as the difference between the time that a request leaves a client device and the time that a response is received at the client. In the datacenter context, we consider front end servers as clients

memory) has stagnated, the network bandwidth has been doubling every 12 to 15 months, even beating the Moore's law for processors performance [6]. However, the network hardware organization of servers has been intact since the construction of the first modern datacenters [7]. That is, sending and receiving network packets involves several costly PCIe transactions and memory copies. For a small size network packet, PCIe transactions and memory copies can take up to 90% of the end-to-end network packet latency [8, 9].

In general, the main performance bottleneck in today's datacenter is data movement. Datacenters deploy high-end general-purpose processors, GPUs, and application-specific accelerators that are starving for data to process. For better or worse, all the data in datacenters is networking data. Network Interface Cards (NICs) are gateways for bringing data in the server and sending it out. Such a strategic location of the NIC makes it attractive for pre- or partial processing of the ingress and egress data to accelerate applications or make proactive power management decisions. Besides networking overhead, over 40∼60% of processor cycles are wasted on waiting for DRAM accesses across various datacenter applications [10, 11]. A naive solution for reducing data movement within a system is to bring intelligence inside the memory and perform computation inside or near memory. Such computation paradigm, known as near-memory processing, has been studied for several decades [12, 13, 14, 15, 16]. Nevertheless, none of these previous proposals got commercialized due to two main reasons. First, the previous near-DRAM processing proposals require significant changes in the processor and DRAM architecture, neither is embraced by industry [17]. Second, there was no volume market for near-DRAM processing as the processor, or memory manufacturers could rely on Moore's law and wait for several months to build a higher-performance chip at the same cost. However, now, in the post Moore's law era, even a few percentage performance or power improvement in a datacenter server can save millions of dollars in the long term [18].

In this thesis, we aim to architect a datacenter that minimizes inter- and intra-server data movement and proactively adjusts the processor speed based on the load on the server to maximize energy efficiency. We propose architectures that leverage in-network computing and near-memory processing technologies. More specifically, this thesis is organized into four parts. Part I discusses two in-network computing architectures for proactive processor management (Chapter 2) and distributed deep neural network (DNN) acceleration (Chapter 3). Part II discusses two near-memory processing architectures for accelerating distributed data-intensive applications (Chapter 4) and an ultra-low latency network interface architecture (Chapter 4). Part III discusses the thesis' contributions to the open-source community, where we have developed a parallel, distributed simulation framework based on a state-of-the-art architectural simulator (Chapter 6). Part IV concludes this thesis by discussing a future research direction.

## 1.2 Thesis Contributions

This thesis's key contribution is redefining the division of tasks between processor, memory, and NIC in a server under the umbrella of OS and architecture co-design. Conventionally, memory was used *only* for storing data, network was used *only* for inter-server communication, and CPU was the only unit for performing the computation. What we propose is to break the boundaries between processor, memory, and network to first, reduce the data movement within and across the servers, and second, and more importantly, build a specialized warehouse-scale computer with a tailored hardware and OS architectures for large-scale computing.

More specifically, here is the list of this thesis' contributions:

- Implementing a full-stack, in-network computing framework for proactive processor power management on a full-system simulator (Chapter 2).

- Building an FPGA prototype of an in-network compression/decompression framework for minimizing communication overhead of distributed training (Chapter 3).

- Implementing a full-stack, application transparent near-memory processing framework for accelerating data-intensive applications on a full-system simulator as well as on an experimental FPGA memory module.

- Implementing a full-stack, ultra low latency near-memory network interface architecture on a full-system simulator.
- Developing and open-sourcing a full-system simulator for modeling computer clusters at the instruction level.

## 1.3 Bibliographic Notes

This thesis is built atop a chain of co-related projects conducted under the supervision of Professor Nam Sung Kim, and got published in top computer architecture venues. Chapter 2 is derived from a best paper nominee article published in the IEEE International Symposium on High-Performance Computer Architecture (HPCA 2017) [19]. Chapter 3 is the product of collaboration with Professor Hadi Esmaeilzadeh,[3] Professor Alexander Schwing,[4] and Dr. Ren Wang[5] and is derived from a conference paper published in the IEEE International Symposium on Microarchitecture (MICRO 2018) [20]. Chapter 4 is the product of collaboration with Professor Wen-mei Hwu,[6] Professor Deming Chen,[7] Professor Daehoon Kim,[8] and several researchers at IBM research and is based on a best paper nominee article published at MICRO 2018 [21]. Chapter 5 is based on a conference paper published in MICRO 2019 [22]. Finally, Chapter 6 is based on a best paper nominee article published in the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2017) [23].

[3]UCSD
[4]UIUC
[5]Intel Research
[6]UIUC
[7]UIUC
[8]Daegu Gyeongbuk Institute of Science and Technology (DGIST)

# Part I

# In-NIC Computing for Higher Energy Efficiency and Lower Communication Cost

The long latency and limited bandwidth of transferring data within a computer and across computers have become a critical bottleneck to improving performance and energy efficiency. In-network computing is an attractive solution for reducing data movement by performing computation near the data. Offloading computation to the network devices is essential to overcome the computation bottleneck in terabits per second network devices: A modern server-class CPU requires ∼15ns [24] to access L3 cache. At the same time, a 200Gbps NIC can deliver cacheline size packets[9] every 2.4ns! Currently, in-network computing is used in production datacenters in the form of offloading network functions to SmartNICs. Microsoft deploys SmartNICs in its Azure fleet [25], and Amazon uses them in its AWS Nitro systems for accelerating network function virtualization [26].

For better or worst, all the data inside a datacenter is network data. This means that the data that is to be processed in a server crosses a server's NIC at least once. This property makes the network devices an attractive place for performing *structural computing*. Unlike *temporal computing*[10] that involves fetching data from memory before performing the computation, in *structural computing*, logic (*i.e.,* structure) is fixed on an ASIC chip, or a reconfigurable fabric such as FPGA, and data is streamed through the structure. Therefore, data movement is no longer a bottleneck in *structural computing*. A range of network applications can benefit from the structural computing opportunity at the NIC. For example, Microsoft uses a bump-in-the-wire FPGA device for accelerating virtual filtering [25] or AWS's Nitro accelerate encryption and decryption of network data inside the SmartNIC. Besides facilitating structural computing, since NIC is the point of entry/exit to/from a server, in-NIC computing can enable early prediction of the processor's future processing demand.

Leveraging the unique characteristics of in-network computing, in this part, we discuss two proposals for not only accelerating the performance of networking applications but also improving the energy efficiency of network-connected computers. In Chapter 2 we propose to use in-NIC computing for efficient power management of servers in datacenters. In Chapter 3, we take an initial step toward reducing inter-node communication overhead in distributed deep neural network (DNN) training by proposing an in-

[9]Assuming 64 Bytes cachelines

[10]*i.e.,* von Neumann style computing

network accelerator for compression and decompression of inter-node gradient updates.

# CHAPTER 2

# IN-NETWORK COMPUTING FOR POWER MANAGEMENT

In datacenters, servers are demanded to be not only energy-efficient but also capable of processing every request from clients in a certain amount of time (or satisfy a Service Level Objective (SLO)). To improve energy efficiency, servers may deploy an aggressive power management policy which frequently transitions system hardware components such as processors and memory to low-performance/sleep states when request rates are low. However, such servers may not respond to all the requests from clients without violating a given SLO especially when request rates suddenly surge. This is because transitioning a hardware component from a low-performance/sleep state to a high-performance state incurs a significant performance penalty; if we account for the overhead of system software associated with these transitions, the performance penalty is even higher [27]. Such a performance penalty can increase high-percentile response time. This in turn discourages servers from deploying an aggressive power management policy and thus wastes energy at low utilization.

Tackling this challenge, starting with an intuitive observation that the rate of network packets from clients can significantly affect the utilization and thus performance/sleep states of processor cores in servers, we propose `NCAP`, **N**etwork-driven, packet **C**ontext-**A**ware **P**ower management for **C**lient-server architecture. `NCAP` enhances a network interface card (NIC) and its driver such that it can examine received and transmitted network packets, determine the rate of network packets containing "latency-critical" requests, and proactively transition a processor to an appropriate performance or sleep state. To demonstrate the efficacy, we evaluate on-line data-intensive (OLDI) applications and show that a server deploying `NCAP` consumes 37$\sim$61% lower processor energy than a baseline server while satisfying a given SLO at various load levels.

## 2.1 Introduction

In a client-server architecture, when servers receive requests sent from clients, the servers process the requests and send back responses to the clients. In particular, for OLDI applications such as web search, servers need to reduce high-percentile response time and satisfy a given SLO [28]. At the same time, these servers must improve their energy efficiency.

The processor is the most power-consuming component even in servers with many DRAM modules (DIMMs). For example, the processors in a Google server consume two-third of the total server power at peak utilization and ∼40% when the server is idle [1]. To maximize energy efficiency, therefore, it is important for a power management policy [29, 30] to fully exploit various sleep and performance states supported by modern processors. Depending on the current performance demand, cores in a processor can operate at various performance states by increasing or decreasing their voltage/frequency (and thus power consumption). Moreover, idle cores in a processor can transition to various sleep states by turning off their clock, decreasing their voltage to a level that barely maintains their architectural states after turning off their clock, or turning off both their clock and power supply.

Transitioning a processor core from a sleep or low-performance state to a high-performance state, however, incurs a significant performance penalty. If we account for the overhead of system software layers associated with these transitions, the performance penalty is even higher [27]. Such a notable performance penalty can substantially increase high-percentile response time and discourages server operators from deploying an aggressive power management policy that frequently transitions processor cores to a low-performance or sleep state [2, 31, 32].

It is intuitive that the rate of network packets from clients can significantly affect the utilization and thus performance/sleep states of processor cores in servers. For example, as a server suddenly receives many network packets containing "latency-critical" requests from clients, its processor cores need to operate at a high-performance state so that it can process the requests and send responses back in time. However, if necessary processor cores have been in a sleep or low-performance state, the server needs to transition these processor cores to a high-performance state. If a server occasionally receives only a few network packets enclosing latency-critical requests from clients, it should transition unnecessary processor cores to a low-performance or sleep state.

In this chapter, we propose `NCAP`, **N**etwork-driven, packet **C**ontext-**A**ware **P**ower management for client-server architecture. Specifically, we first show a strong correlation between the rate of received/transmitted network packets and the utilization and performance/sleep state of processors in servers after analyzing the complex interplay between them.

Second, we propose to enhance a NIC and its driver such that `NCAP` can (1) examine received/transmitted network packets; (2) detect latency-critical requests in the network packets; (3) speculate the completion of requested services; (4) predict an appropriate processor performance or sleep state; and (5) proactively transition a processor to an appropriate performance or sleep state. Especially, `NCAP` overlaps a large fraction of a notable performance penalty of transitioning processor cores to a high-performance state with a long latency of transferring received network packets from a NIC to the main memory. Consequently, `NCAP` allows server operators to deploy an aggressive power management policy without notably increasing high-percentile response time. Note that `NCAP` does not simply respond to a high rate of any network packets (*e.g.,* network packets associated with VM/container migrations and storage server operations), as it selectively considers latency-critical network packets.

Lastly, we demonstrate the effectiveness of `NCAP` for two representative OLDI applications with notably dissimilar characteristics: `Apache` and `Memcached` at various load levels using `dist-gem5`. To establish the SLO, we take a baseline server that always operates its processor cores at the highest performance state and measure its 95th percentile response time at a high-load level [33]. At medium- to high-load levels, a server deploying `NCAP` consumes 37∼61% lower processor energy than the baseline server, while satisfying the SLO. At low- to medium-load levels, it consumes 21∼49% lower processor energy than a server employing the most energy-efficient, SLO-satisfying power management policy among the state-of-the-art power management policies supported by Linux.

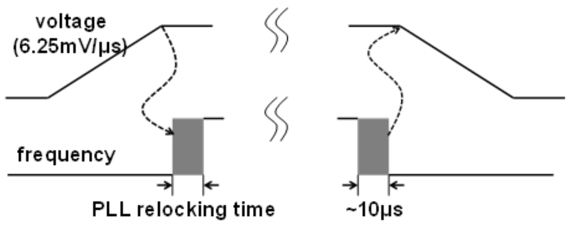


**Figure 2.1:** An example of voltage/frequency changes and performance overhead due to PLL relocking time; the shaded region depicts the duration that a processor core must halt.

## 2.2 Background

### 2.2.1 Processor Power Management

For power management, processors support performance (P) and sleep (C) states that are interfaced with the OS by advanced configuration and power interface (ACPI) [34].

**P state.** The deeper the P state is, the lower the power consumption is at the expense of lower performance. A core in P0 state operates at a voltage/frequency point that offers the maximum sustainable performance under thermal and power constraints.

The current Linux kernel offers three static P-state management policies (*i.e.,* `performance`, `powersave`, and `userspace` governors), and one dynamic P-state management policy (*i.e.,* `ondemand` governor) [29]. Amongst these governors, the `performance` governor always operates the cores at P0, whereas the `powersave` governor always operates the cores at the deepest P state. Lastly, the `userspace` governor enables a user to set the P state of processor cores. In contrast, the `ondemand` governor periodically adjusts the P state based on the utilization of cores.

Figure 2.1 illustrates a typical sequence of changing P state of a core. To increase voltage/frequency, voltage is ramped up to a target level at the rate of 6.25mV/$\mu$s (for Intel processors) before the frequency is raised. To decrease voltage/frequency, frequency is reduced before voltage is decreased. In Intel i7-3770 processors, for example, a transition from the lowest to highest voltage/frequency ($\sim$50$\mu$s) takes much longer time than a transition from the highest to lowest voltage/frequency ($\sim$5$\mu$s) [27], because of the

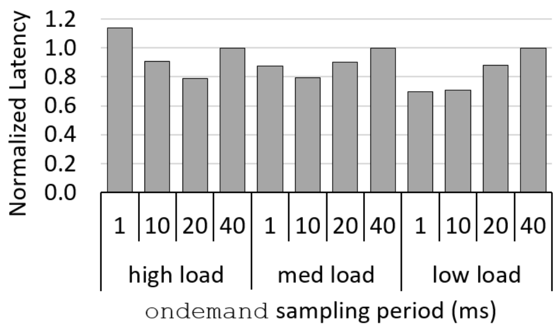


**Figure 2.2:** 95th percentile latency of `Apache` for various invocation periods of the `ondemand` governor.

latency of ramping up voltage before raising frequency. In both cases, the core must halt for $5\mu$s (*i.e.,* PLL re-locking time) while changing frequency. Note that frequency becomes unpredictable and unstable when the PLL attempts to relock the feedback loop of its oscillator to another frequency level.

Figure 2.2 shows the 95th percentile latency of `Apache` for three load levels and various periods of invoking the `ondemand` governor. See Sec. 2.5 for our detailed evaluation methodology. As the minimum invocation period for the `ondemand` governor is hard coded to 10ms in the Linux kernel, we recompiled the Linux kernel after changing the minimum period to 1ms. As shown, the best invocation period varies under different load levels and reducing the invocation period does not always improve the response time due to the performance penalty of frequently invoking the `ondemand` governor and changing voltage/frequency. This is the key reason that the minimum invocation period is hard coded to 10ms [35].

**C state.** `C0`, `C1`, `C3`, and `C6` states denote idle, halt, sleep, and off states. The deeper the C state is, the lower the power consumption is at the expense of higher performance penalty due to longer wake-up latency. The current Linux kernel provides two C-state management policies (*i.e.,* `ladder` and `menu` governors [29]). The ladder governor first transitions a processor core to `C1` state and then a deeper C state if the sleep time was long enough. The `menu` governor records how long a processor core has been in a C state in the past and predicts how long it will stay in the C state in the future. Then, it chooses the most energy-efficient C state based on the prediction and wake-up penalty. Currently, the `menu` governor is used by default.

The current Linux kernel invokes `cpuidle loop` when the scheduler run queue does not have any schedulable job. This function consists of an infinite while loop that repeatedly checks whether or not the run queue has any schedulable job. If there is any newly arrived job, the scheduler is invoked and the jobs in the run queue are executed after being prioritized by the scheduler's policy. Otherwise, the control is delegated to the C state governor (*i.e.,* `menu` governor) to reduce the power consumption of idle cores. In `C0` state the core waits for a job dispatched to the run queue while executing NOP in a kernel while loop. The C state governor is to apply a chosen C state to a core based on its policy. To transition the core from a C state, `MWAIT` and `MONITOR` are used in x86 architecture, as cores in `C1` – `C6` states cannot check whether or not there is a job to do. `MONITOR` arms the address monitoring hardware using an address region specified in `EAX` register. When a store occurs in the specified region, it transitions the core to a P state. One of the C states is denoted by a specific number as a parameter to `MWAIT`. As `MWAIT` and `MONITOR` are privilege (level 0) instructions, they can be executed only in the kernel space and incur a performance penalty of 6–60$\mu$s in Intel i7-3770 processors [36].

### 2.2.2 Network Stack

TCP/IP is the most widely used communications protocol for high-performance computing (HPC) despite its well-known overheads. The Ethernet, as the backbone of a data-center network, is tightly coupled with the TCP/IP layers. However, in addition to bandwidth, low-latency communication is desired to satisfy a given SLO for OLDI applications. The major contributor to the end-to-end TCP/IP packet latency is the network software layers and multiple long latency PCIe transactions to deliver a received packet from a NIC to the main memory and processor (Fig. 2.3).

More specifically, the sequence of receiving a packet from a NIC is as follows. (1) Before receiving any packet, the NIC driver creates a descriptor (or ring buffer) in the main memory (`rx-desc-ring` in Fig. 2.3), which contains the metadata of received packets, and initializes the descriptors to point to a receive kernel buffer (`SKB` in Fig. 2.3). Subsequently, the NIC driver informs the NIC DMA engine of the start address of `rx-desc-ring`. (2) When a packet is received, based on the descriptor information, the

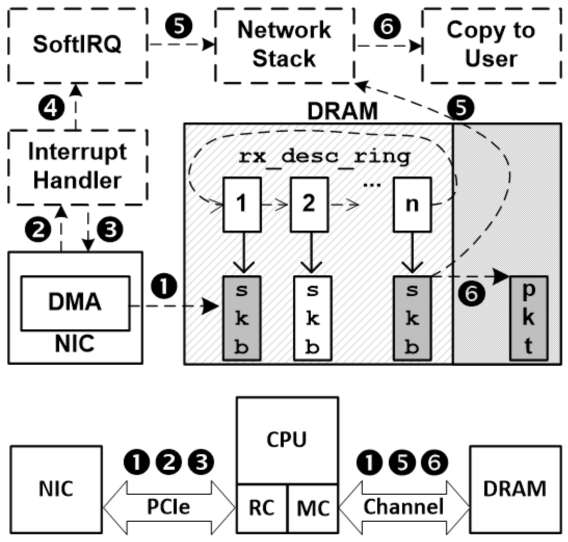


**Figure 2.3:** The sequence of receiving a packet from NIC.

NIC performs a DMA transfer to copy the packet to the associated SKB (step ❶ in Fig. 2.3). (3) The NIC will generate a hardware interrupt to the processor. The interrupt handler of the NIC examines the NIC to determine the interrupt cause, which is done by reading a NIC register called interrupt cause read register (ICR) through a PCIe bus (❷ and ❸ in Fig. 2.3). Some NICs use interrupt moderation technique to reduce the number of interrupts posted to a processor by coalescing several hardware interrupts to one interrupt. Although this technique reduces the load on the processor, it increases the end-to-end latency of delivering each packet [37]. (4) After identifying the cause of the interrupt, the interrupt handler enqueues the request for processing the received packet and schedules a SoftIRQ (❹ in Fig. 2.3). (5) The SoftIRQ handler passes the received packet's SKB to higher layers in network stack and reallocates another SKB for the used descriptor (❺ in Fig. 2.3). (6) The packet will be copied into a user space buffer after it is processed by the software layers in the network stack (❻ in Fig. 2.3).

For NCAP, we leverage the latency of steps ❶, ❷, and ❸ to hide the performance penalty of transitioning cores from a sleep or low-performance state to a high-performance state. Our experiment running Apache shows that these steps consume 86$\mu$s on average.

## 2.3 Correlation between Network Activity and Processor Power Management

In many cases, network packets received by a server contain requests to be processed by processor cores. Thus, as a server receives more network packets, the processor utilization will increase. For example, suppose a client sends a request to an OLDI server. As HTTP requests are encapsulated in TCP packets, the request should go through the server network layers before a processor core in the server can start to process the request. Subsequently, the application will decode the request, bring the requested values from the main memory and send them to the client through one or more TCP packets. The key OLDI processing code along with the network software layers (for both receiving requests and transmitting responses) can overwhelm the processor, especially when the server receives a burst of requests. Thus, we hypothesize that the rate of received and transmitted network packets substantially affects the power management of the processor, as the utilization of the processor typically determines the P and C states of the processor.

To demonstrate the correlation amongst the rate of network packets, processor core utilization, and dynamic power management policies, we use `dist-gem5` (Chapter 6) to run `Apache`. See Sec. 2.5 for our detailed evaluation methodology.

In Fig. 2.4 we simulate one server and fifteen clients, run `Apache`, and measure the server's (1) network transmit and receive bandwidth utilization (denoted by `BW(tx)` and `BW(rx)`); (2) processor core utilization (`U(core)`); (3) processor core frequency (`F(core)`); and (4) processor core time spent in `C1`, `C3`, and `C6` states (`TC1(core)`, `TC3(core)` and `TC6(core)`). `BW(rx)` and `BW(tx)` at each point are normalized to the maximum `BW(rx)` and `BW(tx)` during the entire application runtime, respectively. The `ondemand` governor dynamically changes P state (`F(core)`) every 10ms (*i.e.,* the minimum period supported by the `ondemand` governor).

Figure 2.4 (top) shows that a burst of HTTP requests from clients causes a surge in `BW(rx)`, leading to an increase in `U(core)` and eventually a surge in `BW(tx)` for sending the requested responses. The increase in `U(core)` is due to the processing of (1) received and transmitted packets in the network software layers and/or (2) requests by the clients; one core processes

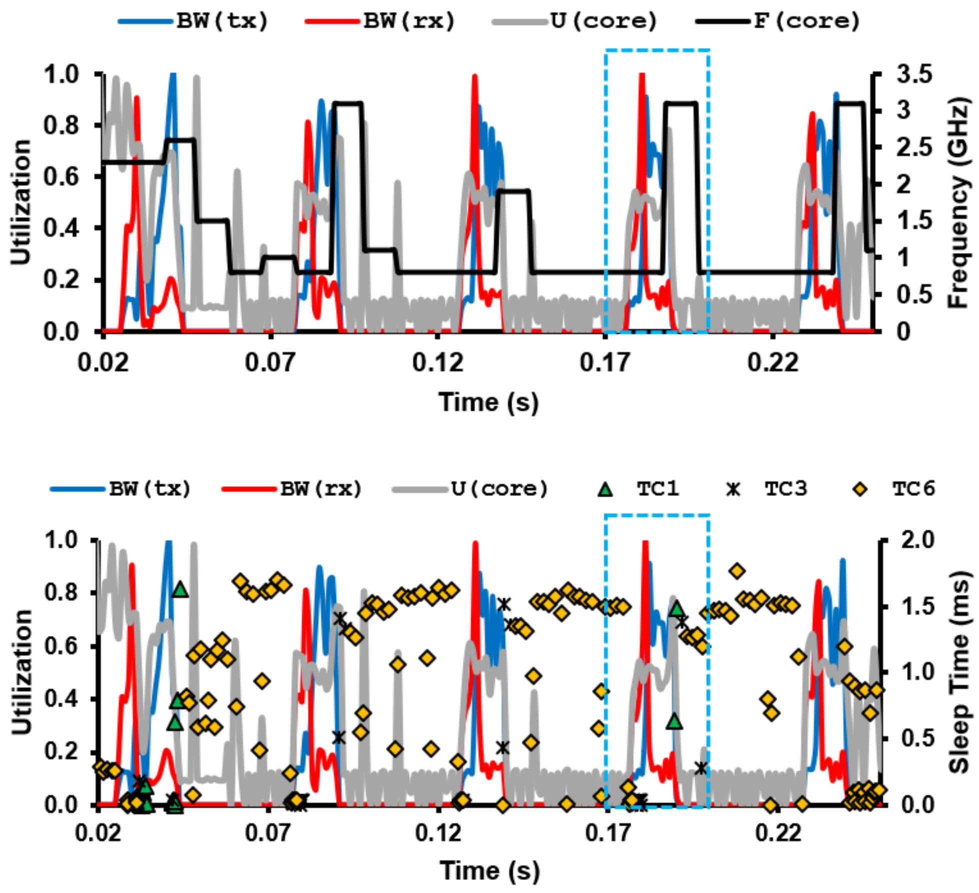


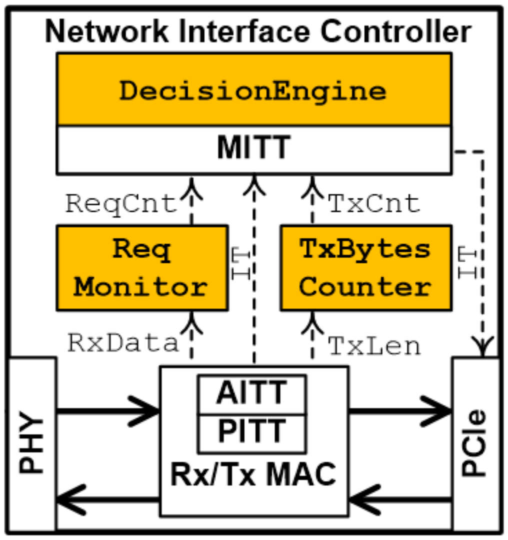
**Figure 2.4: Top**: Network bandwidth (`BW(tx)` and `BW(rx)`), core utilization (`U(core)`), and resulting core frequency (`F(core)`). **Bottom**: core time spent in `C1`, `C2`, and `C3` states (TCx).

received network packets while another core can process requests. In this experiment, we observe that the `ondemand` governor does not immediately react to a sudden increase in `U(core)`, because it can increase `F(core)` only at the end of every 10ms period to amortize the performance penalty of invoking the `ondemand` governor and changing `F(core)` (Sec. 2.2.1). Furthermore, the `ondemand` governor increases `F(core)` only after detecting high `U(core)` in the previous period. Consequently, if the previous period exhibits low `U(core)`, there is a significant delay in increasing `F(core)` (*i.e.,* the maximum delay of up to the period of invoking by the `ondemand` governor), significantly increasing the response time.

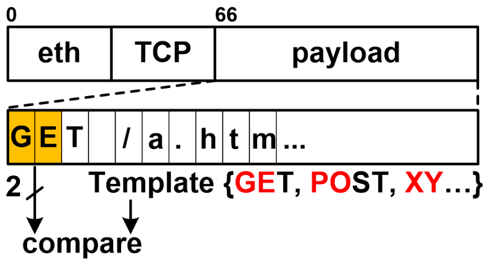
Analyzing the `BW(rx)` burst marked with a dotted box in Fig. 2.4 (top), we observe the surge of `BW(rx)` for 6ms at time 177ms (and that of `U(core)` shortly after that of `BW(rx)`). Subsequently, we observe the surge of `BW(tx)` for 9ms at time 181ms. The average `BW(rx)`, `BW(tx)`, and `U(core)` during this surging period is 42%, 47%, and 48%, respectively. A perfect `ondemand` governor would have boosted `F(core)` at time 177ms, keep `F(core)` high for 14ms, and reduce `F(core)` at time 191ms. However, the `ondemand` governor increases `F(core)` from 0.8 to 3.1 GHz at 188ms (*i.e.,* 11ms late) and reduce `F(core)` at 198ms (*i.e.,* 7ms late).

Figure 2.4 (bottom) shows that entering C states is also highly correlated with the bursts of HTTP requests and their duration. During the idle period between two request bursts, the processor core often transitions to `C6` state to reduce power consumption. This shows that the `menu` governor is effective in transitioning a processor core to a C state when the processor core has been idle for a certain period. However, as depicted in Fig. 2.4 (bottom), the processor core frequently transitions to `C6` state before a surge of `BW(rx)`. That is, the processor core for processing network packets (and possibly requests) is in a C state. Thus, the `menu` governor needs to transition the processor core from the C state to a P state before any code execution. Furthermore, some transitions to C states are very short during the surges of `BW(rx)`. These short transitions to C states can hurt the power efficiency of the processor [38]. Analyzing the `BW(rx)` surge marked with a dotted box in Fig. 2.4 (bottom), we see that core 0, 1, 2, and 3 are in `C3`, `C6`, `C6`, and `C6` for 2$\mu$s, 337$\mu$s, 111$\mu$s, and 2.12ms, respectively. At the very beginning of the surge, the cores transition to `C3` and `C6` states 10 and 5 times and stay in these C states for 30$\mu$s and 31$\mu$s, respectively, on average. After 2ms from the beginning of the surge period, the menu governor does not transition the processor cores to C states anymore until the `BW(rx)` and subsequent `BW(tx)` surging periods end.

The rate of network packets is inherently unpredictable at the low- to medium-levels of request rates (or simply load levels). That is, the network packet rate suddenly increases and decreases after it stays low for a long period. On the other hand, as discussed in Sec. 2.2.1, a server experiences long latency to transition processor cores from a deep C or P state to `P0` state before it can process received requests. Such long latency increases high-percentile response time for next bursts of requests, and thus may entail SLO violation. Consequently, server operators may simply deploy the



(a) Enhanced NIC



(b) Request Monitor

**Figure 2.5:** NCAP comprised of (a) enhanced NIC; (b) REQMonitor in NIC.

performance governor that always operates processor cores at P0 state and thus waste energy at low- to medium-load levels.

## 2.4 NCAP Power Management

In this section, we explain NCAP that is developed based on our observations from Sec. 2.3. More specifically, NCAP, which aims to assist the ondemand and menu governors, leverages a low-level network packet context to proactively transition cores to an appropriate C or P state. This can significantly improve both response time and energy efficiency compared with the default ondemand and menu governors.

Figure 2.5 depicts the key aspects of NCAP. In the enhanced NIC (Fig. 2.5a), REQMonitor and txBytesCounter observe received and transmitted network packets. If REQMonitor and txBytesCounter detect a significant increase in the rate of received network packets (encapsulating latency-critical requests) and decrease in the rate of transmitted network packets, DecisionEngine triggers a special interrupt sent to the processor. Subsequently, the enhanced interrupt handler of the NIC driver running on a processor core proactively transitions necessary processor cores to P0 state (*i.e.*, the highest-performance state) if the processor cores have been in deep P (low-

**Algorithm 1:** `Decision engine` algorithm in NIC.

```
@(MITT expiration)
 if ReqRate > RHT & level != FCONS then
     ICR |= ITRX | ITHIGH
     post interrupt
     level = FCONS
 else if [reqRate < RLT & txRate < TLT] for 1 ms & level > 0 then
     ICR |= ITLOW
     post interrupt
     level --
@(reqCnt changes)
 if (currentTime − LastInterrupt) > CIT & reqCnt > 0 then
     ICR |= ITRX
     post interrupt
```

**Algorithm 2:** Enhanced NIC interrupt handler driver.

```
if ICR & ITHIGH then
    set_Fcore(max)
    disable ondemand for one sampling period
    disable menu governor
else if ICR & ITLOW then
    if Fcore > min then
        set_Fcore(Fcore − (max − min) / FCONS)
        disable ondemand for one period
        enable menu governor
```

performance) states (Alg. 2). Furthermore, if a request is received and the `DecisionEngine` observes a long interval between the past interrupts and the current time, it speculates that the processor cores are in C states and immediately generates an interrupt to proactively transition these processor cores to active state. Such immediate C and P state changes for such events allow a server deploying `NCAP` to service a large number of requests abruptly sent from clients in a timely manner and consume lower energy than a server adopting the `ondemand` and `menu` governors.

### 2.4.1 Context-Aware Packet Rate Detection

As observed in Sec. 2.3, it is likely that OLDI applications need high-performance right after a surge in the rate of network packets enclosing latency-critical requests received by servers. A naive approach to respond to such an event is to transition processor cores from their C states to `P0` state as soon as the rate of "any" received network packets exceeds a certain threshold value. Such a naive approach, however, has some limitations.

First, certain types of network packets received by servers are not latency-critical. One example is a packet containing a request to update the content of a web page (*i.e.,* a `PUT` request in Hypertext Transfer Protocol (HTTP)). Another example is network packets of "off-line" data analytic applications that consume high off-chip network bandwidth but do not have SLO to satisfy. Second, the network packet length of most latency-critical requests is often very short [39], and thus the aggregate size of a burst of network packets containing latency-critical requests may not surpass the threshold value set to operate processor cores at `P0` state. On the contrary, the naive approach may unnecessarily transition processor cores to `P0` state, as simply observing the received network packet rate lacks a context (*i.e.,* whether or not the received network packets are latency-critical).

The requests that are generated by OLDI applications typically have a predefined format, following a standardized universal protocol. For instance, HTTP is a unified application protocol that is widely used for OLDI applications. An HTTP request starts with a request type (*e.g.,* `GET`, `HEAD`, `POST`, `PUT`, etc.) which is followed by a requested URL, and other request header fields. To proactively transition processor cores from a deep C or P state to `P0` state, instead of simply using the received packet rate as a hint, we exploit the fact that latency-critical requests of OLDI applications often have a predefined format.

To detect latency-critical requests, we propose `REQMonitor` (Fig. 2.5b) in an enhanced NIC. Most online requests are encapsulated in a TCP packet. The payload field, which includes a request, starts from the 66th byte of a received TCP packet. `REQMonitor` compares the first two bytes of the payload with a set of templates that are stored in NIC internal registers. These template registers, which are programmable through the operating system's `sysfs` interface [40], can be programmed to store latency-critical request types such as `GET`, when running the initialization subroutine of the NIC driver. Consequently, `REQMonitor` can determine whether or not a received network packet is a latency-critical one. If so, `REQMonitor` increments `reqCnt`.

Furthermore, we observe that the significant decrease in the rate of transmitted network packets subsequently entails to low `U(core)` and the `ondemand` and `menu` governors eventually transition the processor cores to deep P or C states in Fig. 2.4, as the processor has completed its service of the requests. Thus, we also propose `txBytesCounter`, which counts the

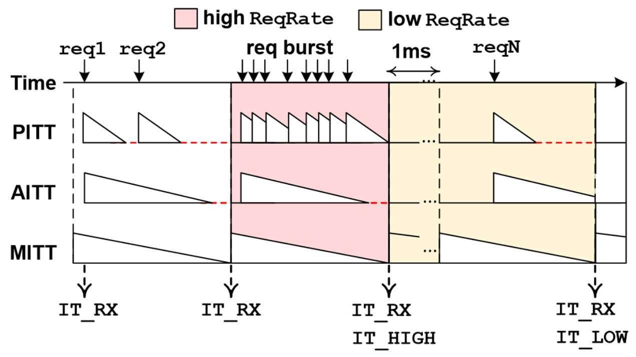


**Figure 2.6:** Illustration of `NCAP` and its interaction with interrupt throttling timers in a NIC.

number of transmitted bytes to determine `txCnt`. The rationale behind using `txCnt` without any context is that most responses are larger than the Ethernet Maximum Transmission Unit (MTU) and thus several TCP packets constituting a single response are transmitted. Detecting such a long chain of latency-critical response packets requires a complex hardware. Moreover, even if the transmitted packets are not latency-critical, operating the processor at `P0` state to complete the packet transmission faster allows the processor cores to transition to a C state sooner. `reqCnt` and `txCnt` will be used for the enhanced `MITT` to determine `reqRate` and `txRate`, respectively, the use of which will be discussed in Sec. 2.4.3.

### 2.4.2 Generation of `NCAP` Interrupt

In order to prevent a NIC from posting too many interrupts to the processor whenever the NIC receives a packet, NICs employ a set of Interrupt Throttling Timers (ITTs) to moderate the number of interrupts that a NIC generates. This is depicted in Fig. 2.6. More specifically, all the Gigabit Ethernet (GbE) controllers contain five timers to moderate the interrupt rate: two Absolute Interrupt Throttling Timers (AITTs); two Packet Interrupt Throttling Timers (PITTs); and one MITT. The AITT and PITT are triggered by a network event (*i.e.,* whenever a packet is received or transmitted) to limit the maximum number of interrupts posted upon receiving or transmitting packets. In contrast, the MITT operates independently from any interrupt source or network event, and constrains the total interrupt rate of a NIC. That is, an interrupt is posted to the processor when the MITT expires. Before posting an interrupt to the processor, the NIC sets an ICR (Sec. 2.2.2) with the type of interrupt that it intends to send to the processor from a set of interrupt types predefined by the device driver (*e.g.,* IT_RX when a received packet is ready to be passed to the network software layers).

For NCAP to trigger a transition of a processor core from a deep C or P state to P0 state at appropriate moments, we propose two more interrupt types, using the unused bits of ICR: IT_HIGH and IT_LOW, respectively. When to trigger IT_HIGH or IT_LOW will be discussed in Sec. 2.4.3.

### 2.4.3 Decision of P and C State Changes

For NCAP to set an ICR and subsequently post an interrupt to the processor at the right moment, we propose DecisionEngine depicted in Alg. 1. Two events trigger DecisionEngine: (1) MITT expiration and (2) ReqCnt changes. When MITT expires (at every 40 to 100$\mu$s), a new ReqRate is determined by ReqCnt. If ReqRate is greater than a request rate high threshold (RHT) and frequency is not already set to the maximum (P0 state), then DecisionEngine posts an interrupt to the processor after setting IT_HIGH and IT_RX bits of ICR. On the contrary, if ReqRate and TxRate are smaller than a request rate low threshold (RLT) and a transmission rate low threshold (TLT) for 1ms, respectively, DecisionEngine posts an interrupt to the processor after setting IT_LOW bit of ICR. When an interrupt with IT_HIGH and IT_RX is posted, NCAP performs a sequence of actions as follows: (1) increasing frequency to the maximum frequency; (2) disabling the menu governor; and (3) disabling ondemand governor for one invocation period. We disable menu governor to prevent short transitions to C states during a surge period of BW(Rx) (Fig. 2.4). We also disable ondemand governor for one invocation period to prevent any conflict between NCAP and ondemand governor decisions. While NCAP sets frequency to the maximum frequency upon an assertion of IT_HIGH, it can be more conservative in decreasing frequency (*i.e.,* reducing frequency to the minimum over several steps). FCONS is a parameter to determine the number of steps to reach the minimum F. That is, the number of required back-to-back interrupts with

IT_LOW to reduce frequency to the minimum. NCAP enables menu governor when the first IT_LOW interrupt is posted.

A change in ReqCnt infers that new requests have been received by NIC. If the time interval between the current request and the last interrupt posted to the processor (CurrentTime – LastInterruptTime) is larger than the processor idle time threshold (CIT), which is typically set by a user or *menu* governor, DecisionEngine immediately posts an interrupt with IT_RX to the processor. When the processor has not been interrupted for a long time, NCAP speculates that processor cores have been in an idle state for a while, and thus transitioned to a C state (Sec. 2.3). In such an event, NCAP immediately sends an interrupt to the processor so that the target processor core to process the request(s) can transition from a C state to an active state and gets ready to service the requests.

Algorithm 2 shows the enhancements in the NIC hardware interrupt handler. When an interrupt is received from the NIC, if the IT_HIGH bit of ICR is set, the NIC hardware interrupt handler calls some APIs of the cpufreq driver, which is responsible for changing frequency in the Linux kernel, to change frequency to the maximum. Otherwise, if the IT_LOW bit is set, then the NIC hardware interrupt handler determines the next frequency based on FCONS.

Figure 2.6 overviews NCAP under a certain packet arrival scenario. Assume that req1 is received after the NIC has been in a long idle period (longer than CIT). Subsequently, DecisionEngine immediately sends an interrupt with IT_RX to transition a processor core to a P state regardless of the MITT expiration time. Later, when a burst of requests is received, ReqRate is re-calculated upon the expiration of MITT. This triggers DecisionEngine to send an interrupt with IT_HIGH to change frequency to the maximum and disable menu governor. After detecting a low-activity period of 1ms, DecisionEngine sends one or several interrupts with IT_LOW to decrease frequency, and enable menu governor again, depending on whether a given policy is aggressive or conservative.

## 2.5 Methodology

For our study, we use `dist-gem5`, which is an enhanced version of `gem5` that supports the simulation of multiple nodes using multiple simulation hosts and synchronization among these simulated nodes [23]. Chapter 6 explains `dist-gem5` framework in detail. To enable `ondemand` governor in `gem5`, we leverage prior work [41]. More specifically, we enable the P state controller, a memory-mapped device that provides registers for the ACPI and changes F of simulated cores. The implemented P state controller with a modified `cpufreq` driver in Linux allows `ondemand` governor to control the current P state of simulated cores. For our experiment, we run `ondemand` governor with an invocation period of 10ms (*i.e.,* the minimum period supported by the default `ondemand` governor [42]).

Furthermore, we implement a `cpuidle` driver for `gem5` to enable the `menu` governor [30]; a `cpuidle` driver conveys the information on available C states to the `menu` governor. Two key parameters that affect the `menu` governor's decision are the exit latency (*i.e.,* latency associated with transitioning from a C state to a P state and the residency (*i.e.,* the minimum amount of time the processor core should spend in a given C state to make the transition worth the energy penalty). In our experiments, we study the `cpuidle` behavior with three C states, `C1`, `C2`, and `C3` with exit latency of 2$\mu$s, 10$\mu$s, and 22$\mu$s and residency of 10$\mu$s, 40$\mu$s, and 150$\mu$s, respectively, which are taken from [27]. We model these C states by halting the execution of instructions. Once the `cpuidle` driver commands a core to transition to a C state, the instruction fetch is stalled and it idles as soon as the processor core pipeline is drained. On the other hand, when it is decided to transition an idle processor core to a P state, the processor core resumes fetching and executing instructions after applying an appropriate delay to model the exit latency.

To evaluate `NCAP`, we model a four-node cluster. We configure each simulated node using the parameters tabulated in Table 2.1 (similar to Intel i7-3770 processor). Then we run `Memcached` [43] and `Apache` [44] to get the round trip latency of each request by annotating the source code of `Memcached` and `Apache` clients with `gem5` pseudo instructions. This is not to perturb the system under evaluation by injecting performance-monitoring functions. We modified the source code to implement open-loop `Memcached` and `Apache` clients and run them on multiple simulated nodes.

**Table 2.1:** Processor Configuration.

| **Parameters** | **Values** |
|---|---|
| Number of cores | 4 |
| Number of P and C states | 15 and 3 |
| Voltage/frequency at P states | 0.65V/0/.8GHz to 1.2V/3.1GHz |
| Processor max power at P states | 12-80W |
| `C1`, `C3`, and `C6` transition latency | 2, 10, and 22$\mu$s |
| Core static power at `C1` | 1.92-7.11W |
| Core static power at `C3` | 1.64W |
| System bus frequency | 1.2 GHz |
| Superscalar | 5 ways |
| Integer/FP ALUs | 3/2 |
| ROB/IQ/LSQ entries | 128/36/72/42 |
| Branch predictor | Bi-Mode |
| L1I/L1D/L2/L3 size (KB) | 64/64/256/4096 |
| L1I/L1D/L2/L3 associativity | 2/4/8/8 |
| DRAM | 8GB DDR3_1600 |
| Network interface driver | Intel 82574GI Gigabit Ethernet |
| Network link | 10Gbps with 1$\mu$s latency |
| Operating System | Linux Ubuntu 11.04 |

This is necessary to prevent (1) client-side queuing bias and (2) dependency between request bursts, which are the two common pitfalls of the OLDI benchmarking tools [45]. We set up three clients, each of which sends requests to one server for `Memcached` and `Apache`. With three client nodes, we are able to achieve the maximum load level that an `Apache` or `Memcached` server can sustain without introducing the client-side queuing delay, which was shown to often incur misleading long response time for some requests [45]. Scaling up the number of nodes will distribute requests from more clients to more servers, entailing similar load levels as our setup. To model the bursty nature of the datacenter traffic [46], we set up each client such that it periodically sends a burst of requests (*e.g.,* 200 requests per burst) to the `Memcached`/`Apache` server. Depending on the target load level, we change the period between 1.3 and 20 ms.

We use McPAT [47] to calculate power and energy consumption of processor cores. For calculating the energy consumption of processor cores in C states, we make assumptions as follows: In `C1`, `C3`, and `C6` states, processor cores consume no dynamic power, while processor cores consume static power at the voltage used right before transitioning to `C1` state, static power at 0.6V, and no static power, respectively.

To demonstrate the advantages of `NCAP` over a software approach to detect latency-critical requests (*e.g.,* [48]), we also implement the algorithms of `REQMonitor` and `DecisionEngine` hardware in the NIC device driver. The RX `SoftIRQ` interrupt handler calls a `REQMonitor` function (Fig. 2.5b) for each received packet before sending it to upper network layers and increments `reqCnt` if the packet contains a latency-critical request. We also count `txBytes` in the TX `SoftIRQ` interrupt handler and utilize a high-resolution kernel timer, which expires every 1ms to determine `reqRate` and `txRate`. We transition cores from a C state to `P0` state as soon as we detect a burst of latency-critical requests. The P state change policy is the same as hardware implementation (Alg. 1).

## 2.6 Evaluation

In this section, we evaluate the effectiveness of `NCAP` in reducing both the response time and energy consumption. We consider four power management policies of Linux. (1) `perf` disables C states and uses only the `performance` governor. (2) `ond` disables C states and uses only the `ondemand` governor. (3) `perf.idle` uses both the `performance` and `menu` governors. (4) `ond.idle` uses the `ondemand` and `menu` governors. Then we compare these four policies with three policies of `NCAP` running atop `ond.idle`. (1) `ncap.sw` is a pure software-based implementation of `NCAP`, implementing `REQMonitor`, `txCounter`, and `DecisionEngine` in the NIC kernel driver. (2) `ncap.cons` is `NCAP` with `FCONS` set to 5 to reduce frequency conservatively over five steps. (3) `ncap.aggr` is `NCAP` with `FCONS` set to 1 to reduce frequency aggressively. We set the threshold values of `DecisionEngine` as follows: `RHT` = 35K request per second (RPS), `RLT` = 5K RPS, `TLT` = 5M bit per second (BPS), and `CIT` = 500$\mu$s, all of which are determined by our experimental analysis of `Memcached` and `Apache`.

Servers are often overprovisioned to meet a given SLO under a certain high load. Therefore, the SLO is typically set right at the knee of the latency-load curve [33]. Figure 2.7 plots the 95th percentile latency versus load. The measured latency of the knee is 3ms and 41ms for `Memcached` and `Apache`, respectively. We measure the response times of requests and

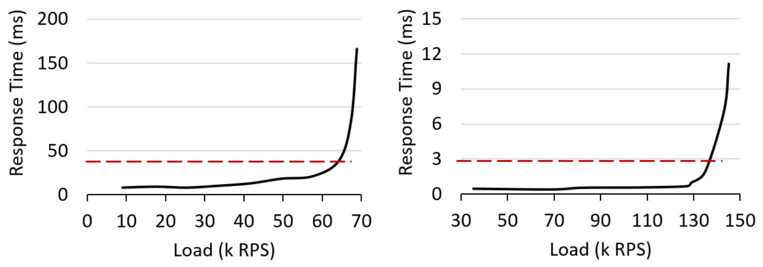


**Figure 2.7:** Latency versus load for `Apache` (left) and `Memcached` (right).

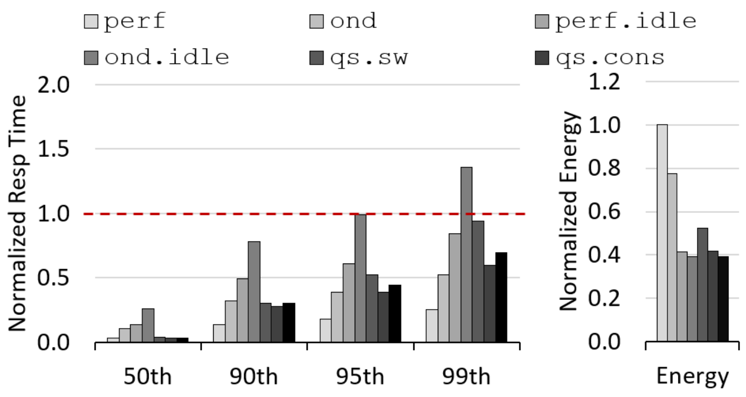


**Figure 2.8:** `Apache` response time distribution (left) and energy consumption (right) when running `Apache` at low-load level.

energy consumption after running `Apache` and `Memcached` at low-, medium- and high-load levels (which corresponds to 24K, 45K, and 66K RPS for `Apache` and 35K, 127K and 138K RPS for `Memcached`). We normalize the measured response times to SLO [31]. The energy consumption of each policy is normalized to `perf`. We also plot a 200ms snapshot of server's `BW(rx)` and `F(core)` for `ond.idle` and `ncap.cons`. `INT(wake)` marks the time that `NCAP` sends interrupts to the processor cores in order to proactively transition processor cores from a deep C or P state to `P0` state.

**Apache.** Figure 2.8 and Fig. 2.9 demonstrate that `NCAP` improves the energy efficiency of the `Apache` server while satisfying the SLO. Analyzing the energy consumption at the low-load level (Fig. 2.8), we observe that `ond` offers 22% lower energy consumption than `perf`. On the other hand, `perf.idle` provides 58% lower energy consumption than `perf`. This emphasizes the importance of transitioning cores into a C state when a server is underutilized. Note that `ond.idle` give marginally ( 5%) lower

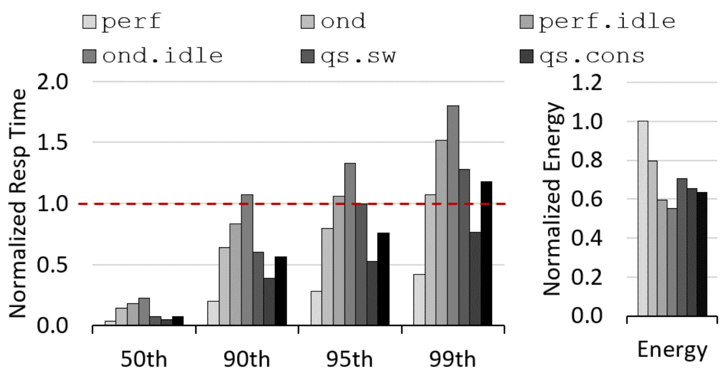


**Figure 2.9:** `Apache` response time distribution (left) and energy consumption (right) when running `Apache` at medium-load level.

energy consumption than `perf.idle`. This is because `perf.idle` makes cores process incoming requests as fast as possible at P0 state and then transitions the cores to a deep C state. This is often more energy-efficient than a policy that makes cores process the requests at a deep P state, which consumes lower power but takes a longer time.

While all the seven policies satisfy the SLO at the low-load level, as shown in Fig. 2.9, `perf.idle` and `ond.idle` fail to satisfy the SLO at the medium-load level. Therefore, `perf.idle` and `ond.idle` are not viable policies for our server configuration. In contrast, `NCAP` (`ncap.aggr` and `ncap.cons`) meets the SLO at both the low- and medium-load levels. Among conventional policies (*i.e.*, `perf`, `ond`, `perf.idle`, and `ond.idle`), `ond` is the most energy-efficient one that can satisfy the SLO. Nonetheless, `ncap.aggr` offers 49% and 21% lower energy consumption than `ond` at the low- and medium-load levels, respectively. Compared with `ond`, the relative energy reduction by `NCAP` diminishes as the load level increases, as the opportunity for cores to run at a deep P state or transition to a deep C state decreases.

A conservative `NCAP` (*i.e.*, `ncap.cons`) can provide lower response time than an aggressive `NCAP` (*i.e.*, `ncap.aggr`). This is because `ncap.cons` can prevent cores from hastily transitioning to a deep P state when the inter-arrival time between `BW(rx)` bursts is short. Consequently, the response time disparity between `ncap.cons` and `ncap.aggr` is more significant at the medium-load level than the low-load level.

Compared with `ncap.aggr`, `ncap.cons` offers 12% and 31% lower 95th percentile response time than `ncap.aggr`, but gives 6% and 3% higher

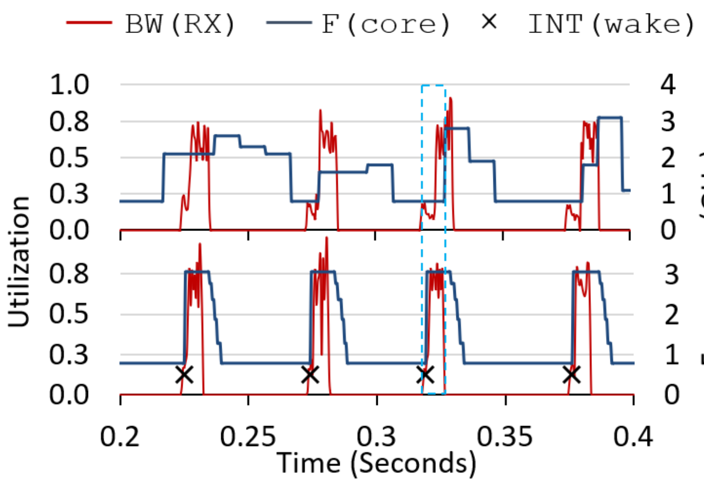


**Figure 2.10:** `BW(rx)` versus `F(core)` snapshot of `ond.idle` (top), and `ncap.cons` (bottom) when running `Apache` at low-load level.

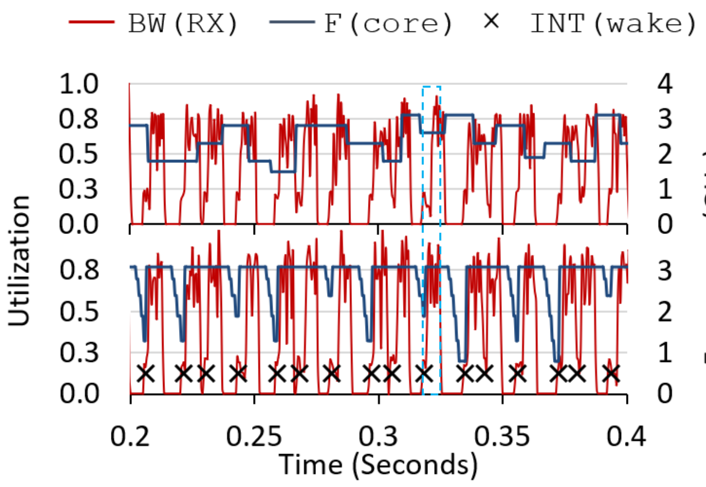


**Figure 2.11:** `BW(rx)` versus `F(core)` snapshot of `ond.idle` (top), and `ncap.cons` (bottom) when running `Apache` at medium-load level.

energy consumption at the low- and medium-load levels, respectively. Lastly, the `BW(rx)` versus `F(core)` snapshot in Fig. 2.10 and Fig. 2.11 clearly demonstrates the shortcoming of `ond` and `ond.idle` as described in Sec. 2.3, and how `NCAP` assists `ond.idle` to quickly meet a sudden processing demand increase for the Apache server.

We observe that ncap.sw can neither fulfil SLO nor provide significant energy reduction, as the load level increases. ncap.sw gives only 11% lower energy consumption but 25% higher 95$^{th}$ percentile response time than ond at the medium-load level. Although this increase in response time does not lead to SLO violations at the given load levels shown in Fig. 2.9, we observe that ncap.sw fails to satisfy SLO at higher-load levels. This is because at high-load levels, the overhead of REQMonitor implemented in software and periodically invoking DecisionEngine can overwhelm the processor cores and keep them from spending cycles for processing packets and requests. Even at low-load level, ncap.aggr and ncap.cons have lower response times than ncap.sw. This demonstrates the effectiveness of proactive power management using an enhanced NIC.

As the NIC and the processor are always highly utilized at high-load levels, idle rarely transitions the processor cores to a C state. Moreover, ond does not change the P state of these cores once ond transitions them to P0 state. This leaves little opportunity for NCAP to exploit. At such high-load levels, the hardware implementation of NCAP just generates ITHIGH interrupt at most every 5ms to set F(core) to the maximum. Note that REQMonitor and DecisionEngine are not in the critical path and the small interrupt generation rate of NCAP does not incur any notable overhead over the default path for packet processing. Therefore, at a high-load level, the energy consumption and response time of NCAP is identical to perf. Note that the power management policies for servers are aimed to reduce energy consumption at low- and medium-load levels without violating a given SLO for occasional surges in the request rate.

Lastly, depending on a given load level, NCAP implements a "race to halt" policy as it transitions all the cores to P0 states and race to complete the task as quickly as possible, then transition the cores to a C state [49]. The dotted boxes in Fig. 2.10 and Fig. 2.11 clearly show the effectiveness of NCAP in accomplishing this.

**Memcached.** Figure 2.12 and Fig. 2.13 show that NCAP improves the energy efficiency of the Memcached server while satisfying the SLO. We observer that the response time of the Memcached server is more sensitive to F(core) than the Apache server. In contrast, the response time of the Apache server was more sensitive to whether or not the menu governor is enabled (perf versus perf.idle) than F(core) (perf versus ond). perf.idle gives

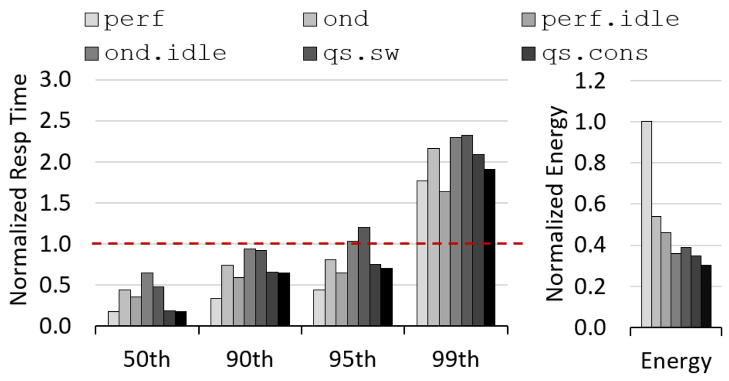


**Figure 2.12:** `Memcached` response time distribution (left) and energy consumption (right) when running `Memcached` at low-load level.

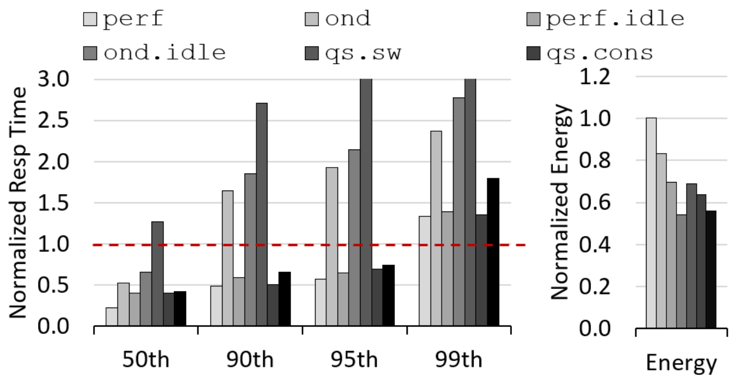


**Figure 2.13:** `Memcached` response time distribution (left) and energy consumption (right) when running `Memcached` at medium-load level.

47% and 12% longer 95$^{th}$ percentile response time than `perf` at the low- and medium-load levels, respectively. Moreover, and `ond` gives 83% and 340% longer 95$^{th}$ percentile response time than `perf` at the low- and medium-load levels, respectively. Note that `Apache` is an I/O-intensive database application that frequently retrieves a “large” amount of data from a storage device of a server. In contrast, `Memcached` is a key-value store application that retrieves mostly small values from the main memory of the server [39]. Consequently, `Memcached` is more sensitive to `F(core)` than `Apache`. This is also confirmed by much longer mean response time of Apache (1.7ms) than `Memcached` (0.6ms).

Although `ond` cannot timely react to the `BW(rx)` surges even at the low-load level (Fig. 2.14, it can at least identify the high processing demand period

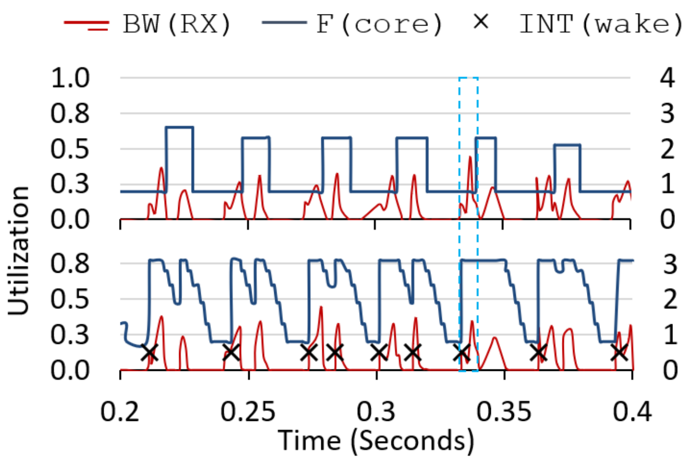


**Figure 2.14:** `BW(rx)` versus `F(core)` snapshot of `ond.idle` (top), and `ncap.cons` (bottom) when running `Memcached` at low-load level.

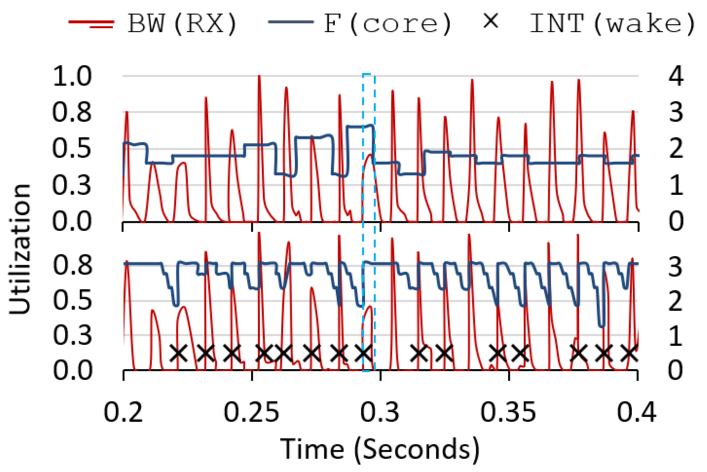


**Figure 2.15:** `BW(rx)` versus `F(core)` snapshot of `ond.idle` (top), and `ncap.cons` (bottom) when running `Memcached` at medium-load level.

incurred by the `BW(rx)` surge and set `F(core)` to high frequency (between 2.3 to 2.6GHz). However, at the medium-load level, `ond` fails to detect the high processing demand period and sets `F(core)` randomly between 1.3 GHz and 2.6GHz. This observation agrees to a previous study demonstrating that considering only CPU utilization for DVFS is often inefficient and leads to SLO violations for servers [33].

Among the conventional power management policies, `perf.idle` is the most energy-efficient one that can satisfy the SLO for `Memcached` at the all load levels. As shown in Fig. 2.12 and Fig. 2.13, `ncap.cons` offers 24% and 9% lower energy consumption, but 15% and 7% longer 95th percentile response time than `perf.idle` at the low- and medium-load levels, respectively. On the other hand, `ncap.aggr` offers 34% and 20% lower energy consumption, but 8% and 14% longer 95th percentile response time than `perf.idle` at the low- and medium-load levels, respectively.

Increasing the load level reduces the opportunity for `NCAP` (and other conventional adaptive power management policies such as `ond`) to transition to a deep P or C state as processor cores are constantly busy. Therefore, the energy consumption of a server deploying `NCAP` eventually converges to that of `perf` as the load level increases. Besides, even `perf.idle` can find little opportunity to transition the processor cores to a C state at high-load levels, leading to no energy reduction compared with `perf`.

`ncap.sw` fails to satisfy the SLO for `Memcached`, as the absolute maximum sustained load level of the `Memcached` server is 2.1X higher than that of the `Apache` server (68 RPS versus 143 RPS). This underscores the overhead of `ncap.sw` for `Memcached` at high-load levels.

As expected, Fig. 2.12 shows that the 50th, 90th, and 95th percentile response times of `perf` is smaller than `perf.idle`. However, the 99th percentile response time of `perf.idle` is lower than `perf`. This is because the response time of OLDI applications is a complex function of the interplay among several hardware/software components, network traffic patterns and associated queuing delays. We see that enabling/disabling idle often reshapes network traffic patterns. This in turn incurs some unexpected severe network resource contentions between received and transmitted network packets at the beginning of a `BW(rx)` surge period, leading to a notable response time increase for a few requests.

## 2.7 Discussion

In Sec. 2.6, we illustrated the effectiveness of `NCAP` in improving the energy efficiency of servers running two OLDI applications with inherently different characteristics and QoS requirements at different load levels. A hardware implementation of NCAP significantly reduces the energy consumption of the processor without any SLO violations. Compared with perf, NCAP significantly reduces energy consumption with comparable overall response time. As shown, at both low- and medium-load levels, NCAP exhibit some slack between the achieved 95th percentile latency and the SLO. This slack can be exploited for further reduction of energy consumption using other techniques [33, 50].

For NCAP, we are simulating a cluster with just one OLDI server. However, a production datacenter consists of hundreds or thousands of servers running OLDI applications. One of key characteristics of large-scale datacenters is the load imbalance amongst server nodes. Therefore, there is a significant fraction of underutilized servers even at a high overall load level [33] and NCAP can achieve energy reduction for such underutilized servers.

The gem5 NIC model that we used for NCAP experiments is a single queue model without any TCP offload engines (TOE). NCAP can also be applicable to a server with high-end multi-queue NICs with TOEs. In a multi-queue NIC, as the target core for packet/request processing is known, NCAP changes the P and C states of the target core independent from other cores (per-core versus chip-wide change of P and C states). This can further improve the effectiveness of NCAP. Because TOEs reduce the load on the processors processing packets, a server employing TOE-capable NICs can sustain a higher rate of network packets, compared with a server with a conventional NIC at the same performance state. NCAP can adapt to such scenarios by increasing the threshold values (RHT and RLT in Alg. 1). Lastly, as a TOE-enabled NIC holds packets longer time within the NIC than a conventional NIC, NCAP has more slack to hide the latency of processor cores transitioning from a sleep or low-performance state to a high-performance state, which in turn allows NCAP to deploy a more aggressive policy.

## 2.8 Related Works

**Selective fast performance boost for tail latency.** To improve tail latency of OLDI applications, Hsu et al. propose Adrenaline, a query-level performance boosting scheme [48]. Adrenaline identifies latency-critical requests in a network-stack "software layer" and rapidly increase V/F using "special on-chip voltage regulator (VR) and clock delivery circuits" for such requests to reduce tail latency. In contrast, `NCAP` does not rely on special on-chip VR and clock delivery circuits. Furthermore, `NCAP` detects latency-critical requests at the lowest network layer (*i.e.,* NIC), which can make much faster detection and decisions than an upper network layer. Finally, `NCAP` can offer higher energy efficiency as it not only quickly increases performance for latency-critical requests but also proactively decreases performance by observing the rate of transmitted packets and detecting the end of a burst of responses.

**Selective performance reduction for energy efficiency.** Pegasus [33] and TimeTrader [50] exploit latency slacks of queries that arrive before deadline to reduce energy consumption without increasing SLO violations for OLDI applications. They slow down the system and reshape distribution of latency with the slack. Rubik [51] proposes a DVFS scheme that finds the lowest performance state of processors that does not violate SLOs using statistical models.

**Energy-efficiency improvement exploiting sleep states.** Meisner et al. propose PowerNap [2] that makes server components quickly transition between a high-performance state and a sleep state to minimize the idle power consumption of servers. They demonstrate that PowerNap effectively reduce idle power consumption for servers with low utilization, but they also argue that PowerNap can experience frequent transitions between the two states and thus increase high-percentile response time for OLDI applications [31]. Thus, they recommend leveraging low-performance states instead of low-power sleep to improve energy efficiency for OLDI services. Later, Rossi et al. [52] show that although making processors operate at low-performance states can decrease energy consumption by up to 15%, it increases high-percentile response time of OLDI applications by 70%. Liu et al. propose SleepScale that effectively combine various performance and sleep state [32].

**Queuing-theoretic analyses of performance and energy**. All the aforementioned proposals in this section formulate and tackle server energy-efficiency challenges using queuing-theoretic analysis approaches in which it is challenging to capture the interplay between low-level system hardware and software layers of the computing stack and evaluate schemes modifying such system hardware and software layers. In contrast, `NCAP` is devised and evaluated based on our full-system simulator that can simulate many nodes connected by a network and run the full-system software stack on the simulated system hardware.

## 2.9 Conclusion

In this chapter, first we made three observations as follows. (1) A sudden increase or decrease in network packet rate is highly correlated with the utilization, and thus performance and sleep states of processor cores. (2) The latency-critical requests of OLDI applications are often encapsulated in network packets with a predefined format. (3) The latency to deliver received network packets from a NIC to the processor is notable in network hardware and software layers. Subsequently, based on these three observations, we proposed `NCAP`, network-driven, packet context-aware power management that enhances a NIC and its driver to assist existing power management policies to improve the energy efficiency of OLDI applications without violating the SLO. More specifically, the enhanced NIC and its driver can detect network packets encapsulating latency-critical requests, speculates the start and completion of a request burst, and predicts the optimal performance and power states of processor cores to proactively transition processor cores from a sleep or low-performance state to a high-performance state. We demonstrate the effectiveness of `NCAP` for two OLDI applications: `Apache` and `Memcached` at various load levels using an enhanced full-system simulator. At low- to medium-load levels, a server deploying `NCAP` consumes 61-37% lower processor energy than the baseline while satisfying a given SLO. Further-more, `NCAP` can provide notably lower energy consumption with faster 95th percentile response time than an approach that detects and reacts to latency-critical requests in a network software layer.

# CHAPTER 3

# IN-NETWORK COMPUTING FOR COMMUNICATION ACCELERATION

Single node Deep Neural Network (DNNs) training can take weeks or months. And even distributed training can take a long time with a large fraction of the training time wasted in communicating weights and gradients over the network. State-of-the-art distributed training algorithms use a hierarchy of worker and parameter server nodes. The aggregators repeatedly receive gradient updates from their allocated group of the workers, and send back the updated weights. In this chapter, we propose an in-network computing architecture for accelerating communication of gradient updates in distributed DNN training. To maximize the benefits of in-network acceleration, the proposed solution, named INCEPTIONN (**In-N**etwork **C**omputing to **E**xchange and **P**rocess **T**raining **I**nformation **O**f **N**eural **N**etworks), uniquely combines hardware and algorithmic innovations by exploiting the following three observations. (1) Gradient updates are tolerant to precision loss therefore they lend themselves better to aggressive lossy compression algorithms. (2) Compressing and decompressing gradients inside software is costly and can increase the overall training time compared to a setup without compression. (3) The centralized parameter server can become a bottleneck with compression as it needs to compress/decompress multiple streams from their allocated worker group.

To reduce the communication overhead in distributed DNN training, we propose a lightweight, hardware-friendly lossy-compression algorithm for floating-point gradients, which exploits the unique characteristics in their values. This compression technique significantly reduces the gradient communication without accuracy loss and with a low-complexity, bump in the wire hardware implementation in the NIC. To maximize the compression ratio and avoid the bottleneck at the parameter servers, we also propose a ring based, decentralized training algorithm where each worker collectively aggregates the gradient values in a distributed manner. The proposed

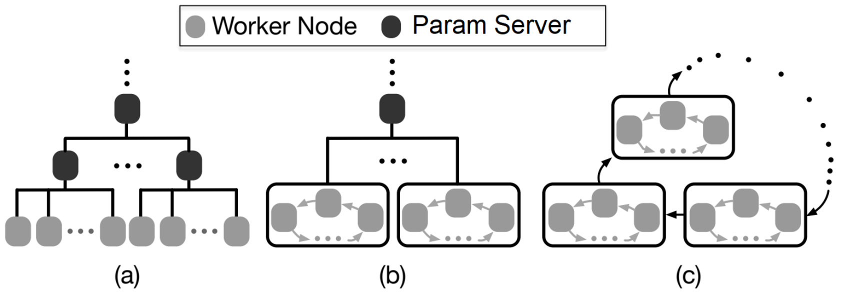


**Figure 3.1:** (a) Hierarchical structure of the stat-of-the-art distributed training. (b) `INCEPTIONN` in the conventional hierarchy. (c) Hierarchical use of `INCEPTIONN`.

decentralized training algorithm leverages the associative property of the aggregation operator and enables our in-network accelerators to (1) apply compression for all network communication between worker nodes, and (2) avoid all-to-one communication pattern and prevent one node from being a bottleneck.

## 3.1 Introduction

Distributed training [53, 54, 55, 56, 57, 58, 59, 60, 61] is the only path forward for supplying the ever-increasing computation demand of DNN applications. However, distributing training suffers from costly inter-node communication that is proportional to the DNN size (*e.g.,* the wight size of AlexNet and ResNet-50 are 232 MB and 98 MB, respectively). Moving forward, employing application-specific accelerators such as TPU or FPGA based accelerators further cuts the computation time and signifies the communication cost [62, 63]. As shown in Fig. 3.1(a), a state-of-the-art distributed training framework [58, 64, 65] uses a hierarchical structure of worker and parameter server nodes. In each training iteration, the parameter servers gather the gradient updates from their sub-nodes, send the cumulative gradients to the higher level parameter server, and send back the aggregated weights downwards. This communication at each iteration involves sending and receiving hundreds of megabytes in real-world DNN models (*e.g.,* 525 MB for VGG-16 [66]), imposing significant pressure on the network. In this chapter, we aim to reduce this communication cost by compressing the network data inside each server's NIC.

Using a general-purpose compression algorithm and offloading it to the NIC hardware provides little gains due to a small compression ratio. Furthermore, the hardware complexity can affect the NIC's line rate. In this chapter, we instead propose a hardware-algorithm co-designed solution, dubbed `INCEPTIONN`[1], that offers a lossy compression technique for gradients, an accelerator architecture for in-network compression/decompression, and a decentralized distributed training algorithm to maximize the benefits the in-network accelerator. `INCEPTIONN` design exploits the following observations: (1) Gradients are more tolerant of precision loss compared to weights. Therefore, gradients lend themselves better to lossy compression with large compression ratios without requiring techniques to alleviate their loss. (2) Compressing and decompressing gradients in software is costly and can increase the overall training time compared to a setup without compression. (3) The centralized parameter server can become a bottleneck with compression as it needs to compress/decompress multiple streams from their allocated worker group.

Leveraging these observations, `INCEPTIONN` comes with a hardware-friendly and simple lossy-compression algorithm for single precision floating-point gradient values. This specialized compression algorithm exploits a unique characteristic of gradient values: the values usually fall in the range of -1∼1 and the distribution peaks near zero. Given this observation, our lossy compression algorithm is tailored for the common case where floating-point values are in the -1∼1 range. The compression algorithm is optimized for simple hardware implementation in the NIC. We also developed a set of standard API's inside Open-MPI framework for seamless integration of the in-NIC accelerators with the network software stack.

In a centralized distributed training framework, workers send gradient updates to the parameter server, and the parameter server sends wights back to the workers. Because weights are not amenable to lossy compression, we can only apply compression for half of the network communications. Moreover, the all-to-one communication pattern from workers to the parameter server requires compressing/decompressing multiple streams at the parameter server, which complicates the accelerator design. To tackle

[1]`INCEPTIONN`: **I**n-**N**etwork **C**omputing to **E**xchange and **P**rocess **T**raining **I**nformation **O**f **N**eural **N**etworks

these challenges, `INCEPTIONN` implements a decentralized, aggregator-free training algorithm, which only communicates gradients between worker nodes (see Fig. 3.1(b and c)). The training algorithm is based on this insight that the aggregation operator (which is typically sum) is associative. Therefore, the gradients can be aggregated gradually by a group of workers. The intuition is to let each worker node to receive partial gradient aggregates from another node in a circular manner and add its contribution to the partial sum. This way, we do not need to have a centralized parameter server for aggregating all gradient values. This algorithm enables the distributed nodes only to communicate gradients and equally share the load of aggregation, which provides more opportunities for compressing gradients and improved load balancing among the nodes.[2] Figure 3.1(b) and (c) depict the view of our algorithm when it is deployed at different hierarchies.

`INCEPTIONN` combines lossy compression algorithm for gradients, in-NIC compression accelerator, and decentralized distributed training algorithm and constructs a cross-stack solution for alleviating the communication bottleneck in distributed DNN training without affecting the training accuracy. We implement an FPGA prototype of `INCEPTIONN` and train state-of-the-art DNN models such as AlexNet [67], VGG-16 [66], ResNet-50 [68]. `INCEPTIONN` reduces the communication time by 70.9$\sim$80.7% while achieving the same level of accuracy.

## 3.2 Communication in Distributed Training

State-of-the-art distributed training algorithms are based on a hierarchical worker-aggregator (*i.e.,* parameter server) approach as illustrated in Fig. 3.2 [58, 64, 65, 63]. In these algorithms, worker and parameter servers construct a hierarchical structure where the leaves are the workers that compute the gradients, and the parameter servers collect the calculated gradients to update the weights in the model and send back the updated weights to the worker nodes.

The hierarchical parameter server approach has two advantages: (1)

[2] Disclaimer: developing the decentralized training algorithm is not among the contributions of this thesis. We discuss it here because of its synergistic ties with the in-NIC compression/decompression acceleration. Please refer to [20] for more information about our decentralized training algorithm

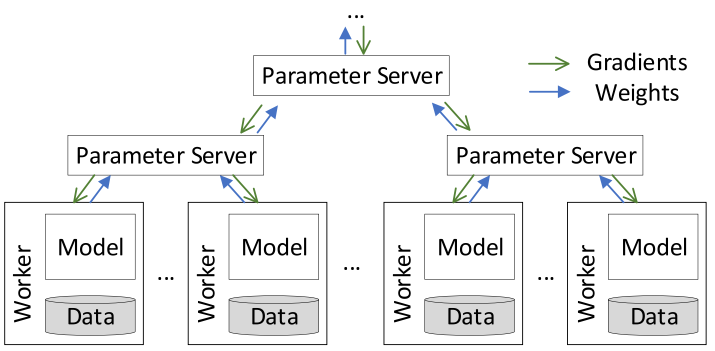


**Figure 3.2:** Parameter server approach for distributed training.

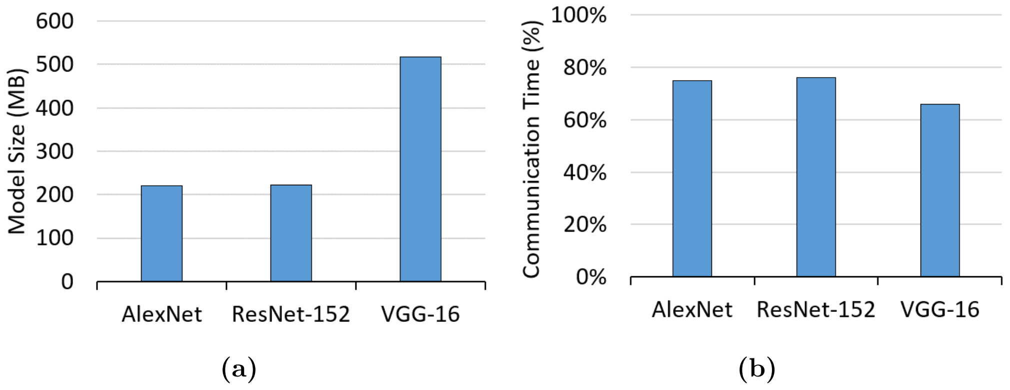


**Figure 3.3:** (a) The size of weights (or gradients). (b) The percentage of the time spent to exchange $g$ and $w$ in total training time with a conventional worker-aggregator approach.

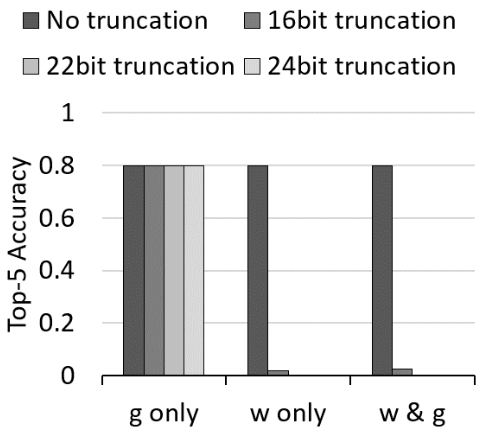
the hierarchical reduction tree organization effectively distributes the aggregation workload to multiple nodes, and (2) reduces the size of system-wide communication by doing the intermediate aggregations. However, the all-to-one traffic pattern between parameter servers and worker nodes can create communication and computation bottlenecks. Figure 3.3 reports the exchanged weight/gradient size and the fraction of communication time when training state-of-the-art DNN models on a five-node cluster with 10Gb Ethernet connections. For instance, per each iteration, AlexNet requires 233 MB of data exchange for each of gradients and weights. Due to the large size of data exchange, 75% of training time for AlexNet goes to the communication. Some recent DNNs (*e.g.,* ResNet-50: 98 MB) that have smaller sizes than AlexNet are also included in our evaluations (Sec. 3.7). Nonetheless, as the complexity of tasks moves past simple object recognition, the DNNs are expected to grow in size and complexity [69]. The communication/computation ratio becomes even larger as the specialized accelerators deliver higher performance and reduce the computation time and/or more nodes are used for training.

## 3.3 Gradients for Compression

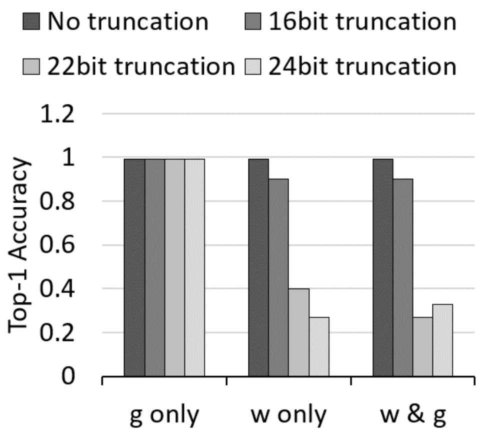
To reduce the communication overhead, `INCEPTIONN` aims to develop a compression accelerator in NICs. Utilizing conventional compression algorithms for acceleration is suboptimal since the algorithms' complexity will impose high hardware cost and latency overhead. Thus, in designing the compression algorithm, we leverage the following algorithmic properties: (1) the gradients have significantly larger amenity to aggressive compression compared to weights, and (2) the gradients mostly fall in the range between -1.0 and 1.0 and the distribution peaks tightly around zero with low variance. These characteristics motivate the design of our lossy compression for gradients.

### 3.3.1 Robustness of Training to Loss in Gradients

Both weights ($w$) and gradients ($g$) in distributed training are normally 32-bit floating-point values, whereas they are 16- or 32-bit fixed-point values in the inference phase [70, 71]. It is widely known that floating-point values are not very much compressible with *lossless* compression algorithms [72]. For instance, using Google's state-of-the-art lossless compression algorithm, Snappy, not only offers a poor compression ratio of $\sim$1.5, but also increases the overall time spent for the training phase by a factor of 2 due to the computing overhead of compression. Thus, we employ a more aggressive *lossy* compression, exploiting tolerance of DNN training to imprecise values at the algorithm level. While lossy compression provides higher compression ratios and thus larger performance benefits than lossless compression, it will affect the prediction (or inference) accuracy of trained DNNs. To further investigate this, we perform an experiment using a simple lossy compression technique: truncating some Least Significant Bits (LSBs) of the $g$ and $w$ values. Figure 3.4 shows the effect of the lossy compression on the prediction



**(a)** AlexNet



**(b)**

**Figure 3.4:** Impact of floating-point truncation of weight $w$ only, gradient $g$ only, and both $w$ and $g$ on training accuracy of AlexNet and Handwritten Digit Classification (HDC). Floating-point truncation drops the LSB mantissa or even exponent bits of the 32-bit IEEE FP format. $x$b-T represents truncation of $x$ LSBs.

accuracy of both trained AlexNet and a handwritten digit classification (HDC) net. This result shows that the truncation of $g$ affects the predictor accuracy significantly less than that of $w$, and the aggressive truncation of $w$ detrimentally affects the accuracy for complex DNNs such as AlexNet. This phenomenon seems intuitive since the precision loss of $w$ is accumulated over iterations while that of $g$ is not.

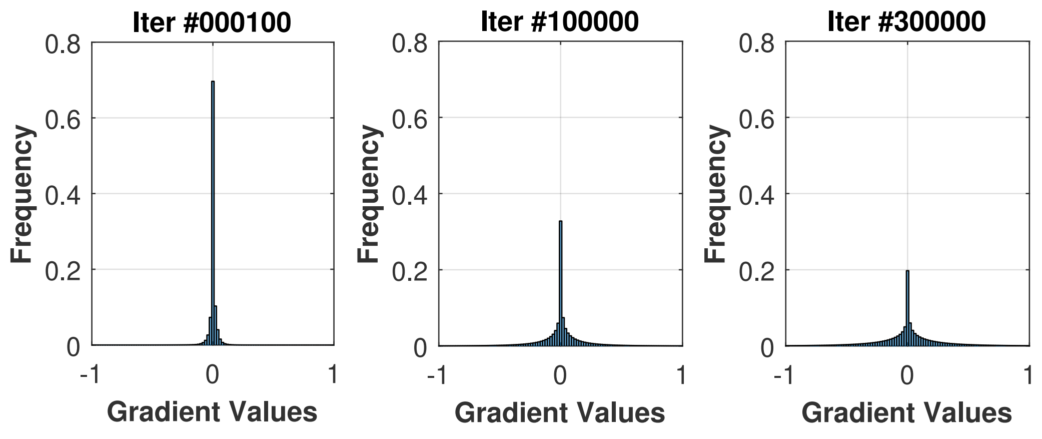


**Figure 3.5:** Distribution of AlexNet gradient values at early, middle, and final training stages.

### 3.3.2 Tightness of Dynamic Range in Gradients

In designing the lossy compression algorithm, we leverage the inherent numerical characteristics of gradient values, *i.e.,* the values mostly fall in the range between -1.0 and 1.0 and the distribution peaks tightly around zero with low variance. We demonstrate the properties, analyzing the distribution of gradients at three different phases during the training of AlexNet. As plotted in Fig. 3.5, all the gradient values are between -1 and 1 throughout the three training phases and most values are close to 0. We also find a similar distribution for other DNN models. Given this observation, we focus on the compression of floating-point values in the range between -1.0 and 1.0 such that the algorithm minimizes the precision loss.

Our lossy compression algorithm (Sec. 3.4) is built upon these two properties of gradients, and exclusively aims to deal with gradients. However, the gradients are only communicated in one direction in the conventional distributed training while the updated weights are passed around in the other direction. Therefore, before delving into the details of our compression technique and its hardware, we first discuss our training algorithm that communicates gradients in all the directions. Hence, this algorithm can maximize the benefits of INCEPTIONN's in-network acceleration of gradients.

## 3.4 Compressing Gradients

**Compression.** Algorithm 3 elaborates the procedure of compressing a 32-bit floating-point gradient value ($f$) into a compressed bit vector ($v$) and a 2-bit tag indicating the used compression mechanism ($t$). Note that this algorithm is described based on the standard IEEE 754 floating-point representation which splits a 32-bit value into 1 sign bit ($s$), 8 exponent bits ($e$), and 23 mantissa bits ($m$). Depending on the range where $f$ falls in, the algorithm chooses one of the four different compression mechanisms. If $f$ is larger than 1.0 (*i.e.,* $e \geq 127$), we do not compress it and keep the original 32 bits (NO_COMPRESS). If $f$ is smaller than an error bound, we do not keep any bits from $f$ (0BIT_COMPRESS). When the gradient values are in the range (error_bound $< f <$ 1.0), we should take a less aggressive approach since we need to preserve the precision. The simplest approach would be to truncate some LSB bits from the mantissa. However, this approach not only limits the

**Algorithm 3:** Lossy compression algorithm for single-precision floating-point gradients.

```
Input  : f: 32-bit single-precision FP value
Output : v: Compressed bit vector (32, 16, 8, or 0 bits)
         t: 2-bit tag indicating the compression mechanism
s ← f[31]        // sign
e ← f[30 : 23]   // exponent
m ← f[22 : 0]    // mantissa
if ($e \geq 127$) then
    v ← f[31 : 0]
    t ← NO_COMPRESS    // 2'b11
else if ($e < error\_bound$) then
    v ← {}
    t ← 0BIT_COMPRESS    // 2'b00
else if ($error\_bound \leq e < 127$) then
    $n\_shift \leftarrow 127 - e$
    $shifted\_m \leftarrow concat(1\text{'}b1,\ m) >> n\_shift$
    if ($e \geq error\_bound + \lceil (127 - error\_bound)/2 \rceil$) then
        $v \leftarrow concat(s,\ shifted\_m[22:8])$
        t ← 16BIT_COMPRESS    // 2'b10
    else
        $v \leftarrow concat(s,\ shifted\_m[22:16])$
        t ← 8BIT_COMPRESS    // 2'b01
    end
end
```

maximum obtainable compression ratio since we need to keep at least 9 MSB bits for sign and exponent bits, but also affects the precision significantly as the number of truncated mantissa bits increases. Instead, our approach is to always set $e$ to 127 and to not include the exponent bits in the compressed bit vector. Normalizing $e$ to 127 is essentially multiplying $2^{(127-e)}$ to the input value; therefore, we need to remember the multiplicand so that it can be decompressed. To encode this information, we concatenate a 1-bit "1" at the MSB of $m$ and shift it to the right by $127 - e$ bits. Then we truncate some LSB bits from the shifted bit vector and keep either 8 or 16 MSB bits depending on the range of value. Consequently, the compression algorithm produces a compressed bit vector with the size of either 32, 16, 8, or 0 and 2-bit tag indicating the used compression mechanism.

**Decompression.** Algorithm 4 describes the decompression algorithm that takes a compressed bit vector $v$ and a 2-bit tag $t$. When $t$ is NO_COMPRESS or 0BIT_COMPRESS, the decompressed output is simply 32-bit $v$ or zero, respectively. If $t$ is 8BIT_COMPRESS or 16BIT_COMPRESS, we should reconstruct the 32-bit IEEE 754 floating-point value from $v$. First, we obtain

**Algorithm 4:** Decompression algorithm.

```
Input  : $v$: Compressed bit vector (32, 16, 8, or 0 bits)
         $t$: 2-bit tag indicating the compression mechanism
Output : $f$: 32-bit single-precision FP value
1  if ($t = NO\_COMPRESS$) then
2      $f \leftarrow v[31:0]$
3  else if ($t = 0BIT\_COMPRESS$) then
4      $f \leftarrow$ 32'$b$0
5  else
6      if ($t = 8BIT\_COMPRESS$) then
7          $s \leftarrow v[7]$
8          $n\_shift \leftarrow first1\_loc\_from\_MSB\ (v[6:0])$
             $m \leftarrow concat(v[6:0] << n\_shift,$ 16'$b$0$)$
9      else if ($t = 16BIT\_COMPRESS$) then
10         $s \leftarrow v[15]$
11         $n\_shift \leftarrow first1\_loc\_from\_MSB\ (v[14:0])$
             $m \leftarrow concat(v[14:0] << n\_shift,$ 8'$b$0$)$
12     end
13     $e \leftarrow 127 - n\_shift$
14     $f \leftarrow concat(s,\ e,\ m)$
15 end
```

the sign bit $s$ by taking the first bit of $v$. Then we find the distance from MSB to the first "1" in $v$, which is the multiplicand used for setting the exponent to 127 during compression. Once we get the distance, $e$ can be calculated by subtracting the distance from 127. The next step is to obtain $m$ by shifting $v$ to left by the distance and padding LSBs with zeros to fill the truncated bits during compression. Since we now have $s$, $e$, and $m$, we can concatenate them together as a 32-bit IEEE 754 floating-point value and return it as the decompression output.

## 3.5 In-Network Acceleration of Gradient Compression

After applying the compression algorithm in Sec. 3.4, we may significantly reduce the amount of data exchanged among nodes in `INCEPTIONN`, but our final goal is to reduce the total training time. In fact, although researchers in the machine learning community have proposed other compression algorithms [73, 74, 75, 76, 77], most of them did not report the total training wall-clock time after evaluating only the compression ratio and the impact of compression on training accuracy. Directly running these compression algorithms in software, though reducing the communication time, can place heavy burden on the computation resources and thus seriously increase computation time. Specifically, such compression algorithms need to run on the CPUs as GPUs cannot offer efficient bit manipulation (*e.g.,* packing some bits from floating-point numbers) compared to CPUs. Prior work [78] shows GPUs offer only $\sim$50% higher throughput at lower compression ratios than Snappy [79].

Figure 3.6 shows that the training time increases by a factor of 2$\sim$4$\times$ even when using the fastest lossless (Snappy) and lossy (SZ [80]) compression algorithms. Even a simple lossy truncation operation significantly increases the computation time, because simply packing/unpacking a large number of $g$ values also significantly burdens the CPUs. This in turn considerably negates the benefit of reduced communication time as shown in Fig. 3.6, only slightly decreasing the total training time. Therefore, to reduce both communication and computation times, we need hardware-based compression for `INCEPTIONN`.

### 3.5.1 Accelerator Architecture and Integration with NIC

**NIC architecture.** To evaluate our system in a real-world setting, we implement our accelerators on a Xilinx VC709 evaluation board that offers 10Gbps network connectivity along with programmable logic. We insert the

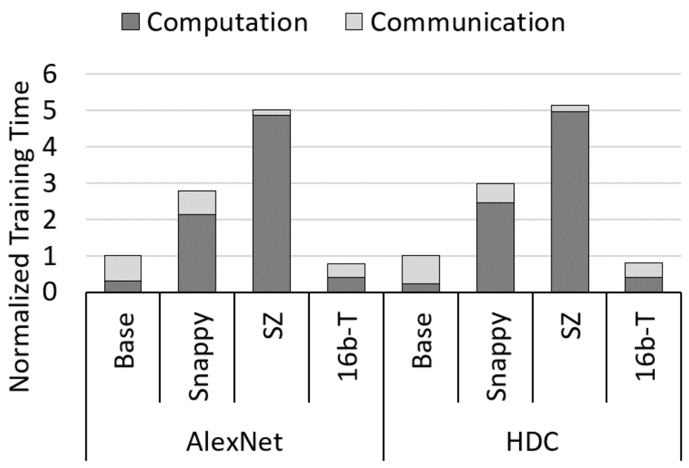


**Figure 3.6:** Impact of software-based lossless (Snappy) and lossy (SZ) compression algorithms on the total training time of AlexNet and HDC. “Base” denotes baseline without compression. $x$b-T represents truncation of $x$ LSBs.

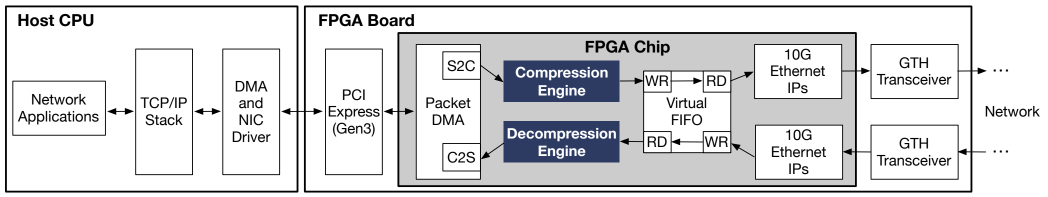


**Figure 3.7:** Overview of NIC architecture integrated with compressor and decompressor.

accelerators within the NIC reference design that comes with the board. Figure 3.7 illustrates this integration of the compression and decompression engines. For output traffic, as in the reference design, the packet DMA collects the network data from the host system through the PCIe link. These packets then go through the Compression Engine that stores the resulting compressed data in the virtual FIFOs that are used by the 10G Ethernet MACs. These MACs drive the Ethernet PHYs on the board and send or receive the data over the network. For input traffic, the Ethernet MACs store the received data from the PHYs in the virtual FIFOs. Once a complete packet is stored in the FIFOs, the Decompression Engine starts processing and passing it to the packet DMA for transfer to the CPU. Both engines use the standard 256-bit AXI-stream bus to interact with other modules.

Although hardware acceleration of the compression and decompression algorithms is straightforward, their integration within the NIC poses several challenges. These algorithms are devised to process streams of floating-point numbers, while the NIC deals with TCP/IP packets. Hence, the accelerators need to be customized to transparently process TCP/IP packets. Furthermore, the compression is lossy, the NIC needs to provide the abstraction that enables the software to activate/deactivate the lossy compression per packet basis. The following discusses the hardware integration and Sec. 3.5.2 elaborates on the software abstraction.

**Compression Engine.** Not to interfere with the regular packets that should not be compressed, the Compression Engine first needs to identify which packets are intended for lossy compression. Then, it needs to extract their payload, compress it, and then reattach it to the packet. The Compression Engine processes packets in bursts of 256 bits, which is the number of bits an AXI interface can deliver in one cycle. Our engines process the packet in this burst granularity to avoid curtailing the processing bandwidth of the

NIC. Our software API marks a packet compressible by setting the Type of Service (ToS) field [81] in the header to a special value. Since the ToS field is always loaded in the first burst, the Compression Engine performs the sequence matching at the first burst and identifies the compressible packets. If the ToS value does not match, compression is bypassed. The Compression Engine also does not compress the header and the compression starts as soon as the first burst of the payload arrives.

Figure 3.8 depicts the architecture of the compression hardware. The payload burst feeds into the Compression Unit equipped with eight Compression Blocks (CBs), each of which performs the compression described in Alg. 3. Each CB produces a variable-size output in the size of either 32, 16, 8, or 0 bits, which need to be aligned as a single bit vector. We use a simple binary shifter tree that produces the aligned bit vector of which possible size is from 0 to 256. The 2-bit tags of the eight CBs are simply concatenated as a 16-bit vector. Finally, the aligned bit vector and tag bit vector are concatenated as the final output of the Compression Unit, of which size is at least 16 bits and can go up to 272 bits. For each burst, the Compression Unit produces

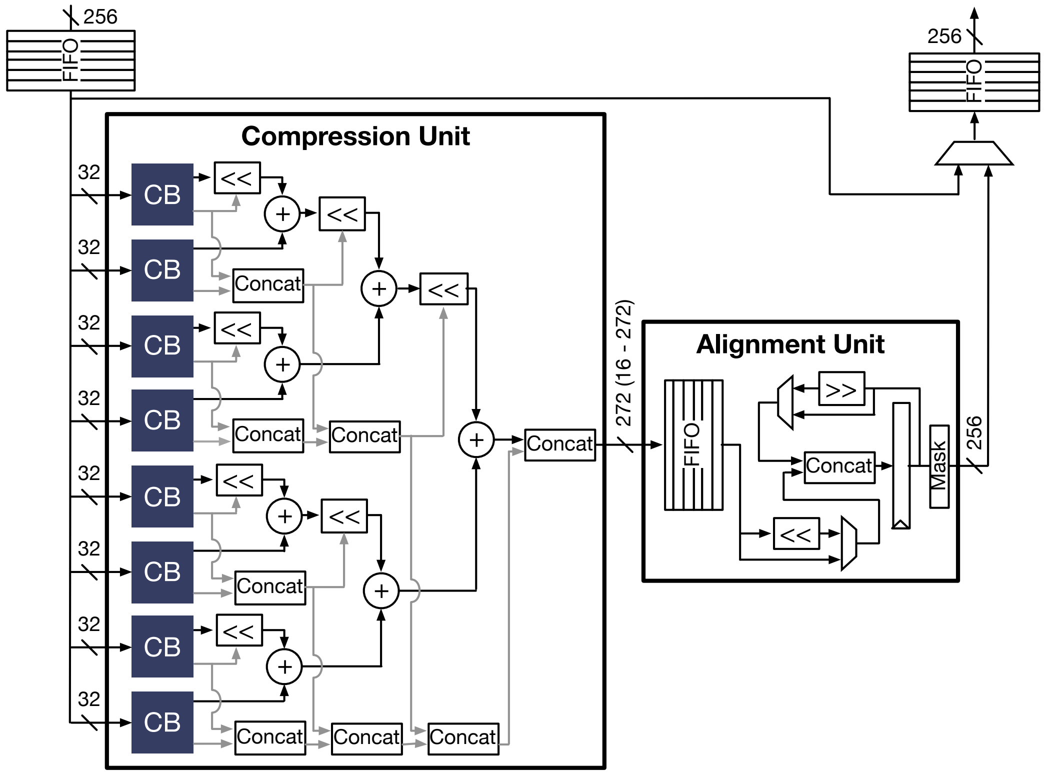


**Figure 3.8:** 256-bit burst compressor architecture.

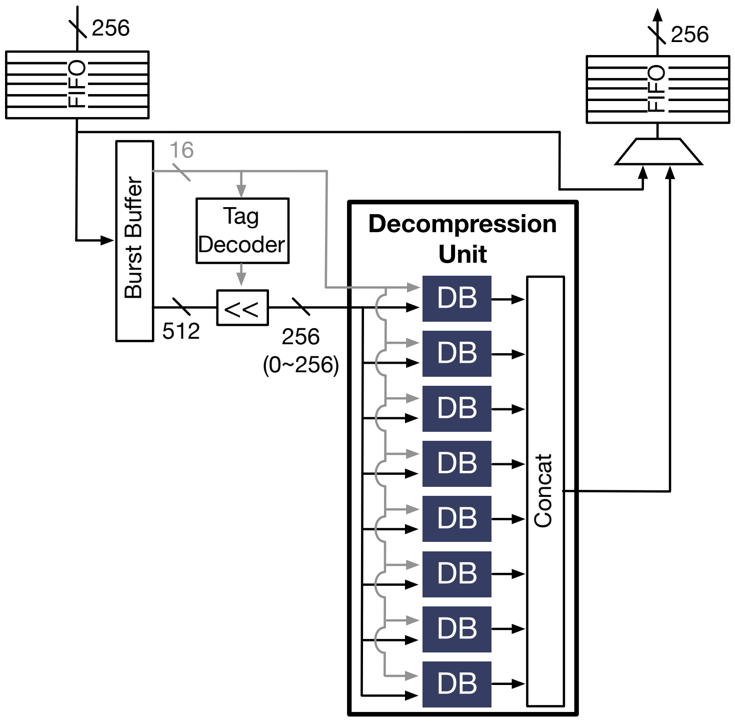


**Figure 3.9:** 256-bit burst decompressor architecture.

a variable-size (16 – 272) bit vector; therefore, we need to align these bit vectors so that we can transfer the 256-bit burst via the AXI interface. The Alignment Unit accumulates a series of compressed bit vectors and outputs a burst when 256 bits are collected.

**Decompression Engine.** Similar to the Compression Engine, the Decompression Engine processes packets in the burst granularity and identifies whether or not the received packet is compressed through the sequence matching of the ToS field at the first burst. If the packet is identified as incompressible or the burst is header, decompression is bypassed. The payload bursts of compressible packets is fed into the decompression hardware, of which its architecture is delineated in Fig. 3.9. Since the compressed burst that contains 8 FP numbers can overlap two consecutive bursts at the Decompression Engine, reading only a single burst could be insufficient to proceed to the decompression. Therefore, the Decompression Engine has a Burst Buffer that maintains up to two bursts (*i.e.,* 512 bits). When the Burst Buffer obtains two bursts, it feeds the 16-bit tag to the Tag Decoder to calculate the size of the eight compressed bit vectors. Given the sizes, the eight compressed bit vectors are obtained from the buffered 512 bits.

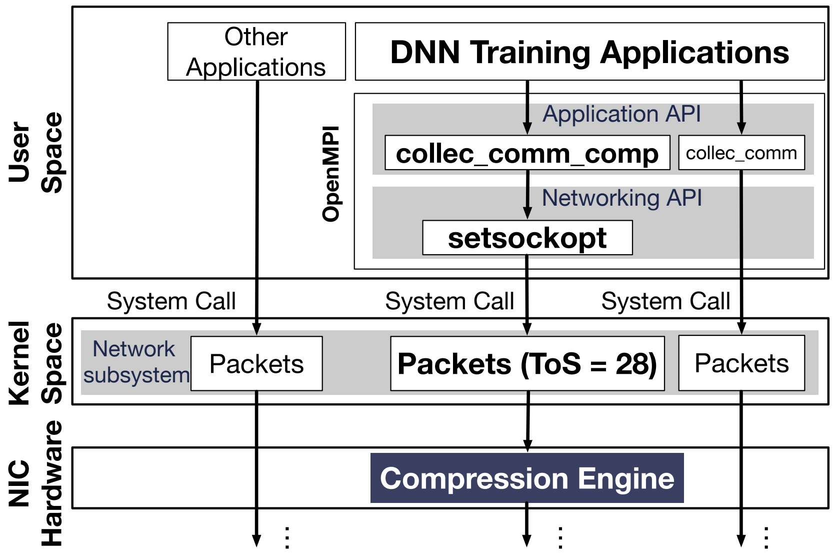


**Figure 3.10:** Dataflow across the software stack and NIC hardware.

Since each compressed bit vector has a variable size of either 32, 16, 8 or 0 bits, the possible size of the eight compressed bit vectors is from 0 and 256. These eight compressed bit vectors (0 – 256) and the tag bit vector (16) are fed into the eight Decompression Blocks (DBs) in the Decompression Unit, which executes the decompression algorithm described in Alg. 4. Then, the Decompression Unit simply concatenates the outputs from the eight DBs and transfers it via the AXI interface. For the next cycle, Burst Buffer shifts away the consumed bits and reads the next burst if a burst (*i.e.*, 256 bits) has been consumed and the left bits are fewer than a burst.

### 3.5.2 APIs for Lossy Compression of Gradients

As mentioned previously, we identify the context of a TCP/IP packet [19] by utilizing the ToS field in the IP header. ToS is an 8-bit field in the header of a TCP/IP packet and is used to prioritize different TCP/IP streams. We tag packets that need to be compressed/decompressed with a reserved ToS value of 0x28. For each socket connection, we can call the setsockopt function to set the ToS field or update it on the fly.

Figure 3.10 demonstrates how we tag TCP/IP packets that need to be compressed or decompressed in the OpenMPI framework. It shows a scenario where we co-run DNN training application and some other networking applications on a server. To properly tag TCP/IP packets that require compression/decompression, we introduce MPI_collective_-communication_comp, which is a specialized MPI_collective_communication API set. We implement our INCEPTIONN algorithm described in Sec. 3.4 without compression with the default MPI_collective_communication APIs. MPI_ collective_communication_comp propagates a variable down to the OpenMPI networking APIs and sets the ToS option of the corresponding TCP sockets used for communication. We do not modify the Linux kernel network stack and the packets with ToS set to 0x28 reach to the NIC like regular TCP packets. Inside the NIC, a simple comparator checks the ToS field of each incoming packet; if the ToS field is set to 0x28, then the packet is sent to the compression engine, otherwise we do not perform compression for the outgoing packet. On a receiver node NIC, we have the same comparator for incoming packets. If the ToS field is set to 0x28, then we perform decompression on the packet. Otherwise, the received packet is a regular Ethernet packet and is directly sent to the processor for reception.

## 3.6 Methodology

### 3.6.1 DNN Models

Table 3.1 enumerates the list of evaluated DNN models with the used hyper-parameters for training.

**AlexNet**. AlexNet [67] is a CNN model for image classification, which consists of five convolutional layers and three fully connected layers with rectified linear unit (ReLU) as the activation function. Before the first and the second fully connected layers, the dropout layers are applied. The model size of AlexNet is 233 MB. For our experiments, we use 1,281,167 training and 50,000 test examples from the ImageNet dataset [82].

**Handwritten Digit Classification (HDC)**. HDC [83, 84, 85, 86, 87] is a DNN model composed of five fully-connected layers, which performs Handwritten Digits Recognition. The dimension of each hidden layer is 500 and the model size is 2.5 MB. The used dataset is MNIST [88], which contains 60,000 training and 10,000 test images of digits.

**Table 3.1:** Hyperparameters of different benchmarks.

| **Hyperparameter** | **AlexNet** | **HDC** | **ResNet-50** | **VGG-16** |
|---|---|---|---|---|
| Per-node batch size | 64 | 25 | 16 | 64 |
| Learning rate (LR) | -0.01 | -0.1 | 0.1 | -0.01 |
| LR reduction | 10 | 5 | 10 | 10 |
| Num of LR reduction iterations | 100000 | 2000 | 200000 | 100000 |
| Momentum | 0.9 | 0.9 | 0.9 | 0.9 |
| Weight decay | 0.00005 | 0.00005 | 0.0001 | 0.00005 |
| Number of training iterations | 320000 | 10000 | 600000 | 370000 |

**ResNet-50**. ResNet [68] is a state-of-the-art DNN model for the image classification task, which offers several variants that have different number of layers. Our experiments use the most popular variant, ResNet-50, which contains 49 convolution layers and 1 fully connected layer at the end of the network. ResNet-50 has a model size of 98 MB and uses the ImageNet dataset.

**VGG-16**. VGG-16 is another CNN model for image classification, which consists of 13 convolutional layers and 3 fully connected layers. VGG-16 also uses ImageNet dataset and its model size is 525 MB.

### 3.6.2 Distributed DNN Training Framework[3]

We develop a custom distributed training framework in C++ using NVIDIA CUDA 8.0 [89], Intel Math Kernel Library (MKL) 2018 [90], and OpenMPI 2.0 [91]. Note that `INCEPTIONN` can be implemented in publicly released DNN training frameworks such as TensorFlow [92]. However, our custom distributed execution framework is more amenable for integration with software and hardware implementation of our lossy compression algorithm. In our custom training framework, all the computation steps of DNN training such as forward and backward propagations are performed on the GPU (also CPU compatible), while communication is handled via OpenMPI APIs. Besides, our framework implements diverse distributed training architectures and communication algorithms using various types of OpenMPI APIs to exchange gradients and weights.

[3]Disclaimer: this is not among the contributions of this thesis. Please refer to [20] for more information about the decentralized training algorithms that authors have developed for this work

### 3.6.3 Measurement Hardware Setup

We use a cluster of four nodes, each of which is equipped with a NVIDIA Titan XP GPU, an Intel Xeon CPU E5-2640 @2.6$GHz$, 32GB DDR4-2400T [93], and a Xilinx VC709 board that implements a 10Gb Ethernet reference design along with our compression/decompression accelerators. We employ an additional node as an aggregator to support the conventional worker-aggregator based approach. We also extend our cluster up to eight nodes to evaluate the INCEPTIONN's scalability, while the rest of experiments are performed on the four-node cluster due to limited resources. All nodes are connected to a NETGEAR ProSafe 10Gb Ethernet switch [94]. We designed the compression/decompression accelerators such that they do not affect the operating frequency (100 MHz) and bandwidth while successfully demonstrating the full functionality with the modified NIC driver and OpenMPI APIs. Our distributed training framework runs concurrently on every node in our cluster and all performance evaluations are based on the real wall clock time. As we discover that the 10Gb Ethernet reference design implemented in a Xilinx VC709 board can achieve only ~2.1 Gb due to inefficiency in its driver and design, we use Intel X540T1 10Gb Ethernet NICs [95] to measure the total training and communication times when we do not deploy hardware compression. That is, we use the Intel X540T1 NIC for all the baseline measurements. To measure the communication time after deploying hardware compression, we first measure the breakdown of communication time (*e.g.,* driver time, NIC hardware time, and TX/RX time through links) from both NICs based on Xilinx VC709 board and Intel X540T1 10Gb Ethernet NICs. Then, we scale the TX/RX time through the link of the Intel NIC based on a compression ratio corresponding to a given iteration to calculate the total communication time while accounting for the compression/decompression time.

## 3.7 Evaluation

### 3.7.1 Performance Improvement with INCEPTIONN

We implement the conventional worker-aggregator training algorithm in a cluster of four workers and one aggregator, as the reference design. Table 3.2 shows a detailed breakdown of the training time of AlexNet, HDC, ResNet-50 and VGG-16, on the cluster. We report both the absolute and normalized time for 100 iterations of training. Irrespective of which DNN model we consider, Table 3.2 shows that (1) less than 30% of the wall clock time is spent for local computations including the forward/backward propagations and update steps, and (2) more than 70% of the time is used for communication, which clearly indicates that the communication is the bottleneck.

Figure 3.11 first compares the training time of the reference design (**WA**) with that of the `INCEPTIONN` (**INC**), when both run for the same number of iterations/epochs without applying compression. We also provide the training time breakdown between computation and communication. This result shows that even in a small cluster without compression, the `INCEPTIONN`'s training algorithm offers 52%, 38%, 49%, and 31% shorter total training time than the worker-aggregator based algorithm for AlexNet, HDC, ResNet-50 and VGG-16, respectively. This is due to 55%, 39%, 58%, and 36% reduction in communication time in comparison with the reference design.

Intuitively, `INCEPTIONN` is much more communication-efficient, because it not only removes the bottleneck link, but also enables concurrent utilization

**Table 3.2:** Detailed time breakdown of training different benchmarks using the worker-aggregator based five-node cluster. Measurements are based on 100-iteration training time in seconds.

| Steps | AlexNet | | HDC | | ResNet-50 | | VGG-16 | |
|---|---|---|---|---|---|---|---|---|
| | Abs. | Norm. | Abs. | Norm. | Abs. | Norm. | Abs. | Norm. |
| Forward pass | 3.13 | 1.6% | 0.08 | 4.9% | 2.63 | 3.5% | 32.25 | 4.3% |
| Backward pass | 16.22 | 8.3% | 0.07 | 4.3% | 4.87 | 6.5% | 142.34 | 17.3% |
| GPU copy | 5.68 | 2.9% | - | - | 2.24 | 3.0% | 12.09 | 1.5% |
| Gradient sum | 8.94 | 4.6% | 0.09 | 5.2% | 3.68 | 4.9% | 19.89 | 2.4% |
| Communication | 148.71 | 75.7% | 1.36 | 80.2% | 60.58 | 80.2% | 583.58 | 70.9% |
| Update | 13.67 | 7.0% | 0.09 | 5.3% | 1.55 | 2.1% | 30.50 | 3.7% |
| Total training time | 196.35 | 100.0% | 1.7 | 100.0% | 75.55 | 100.0% | 823.65 | 100.0% |

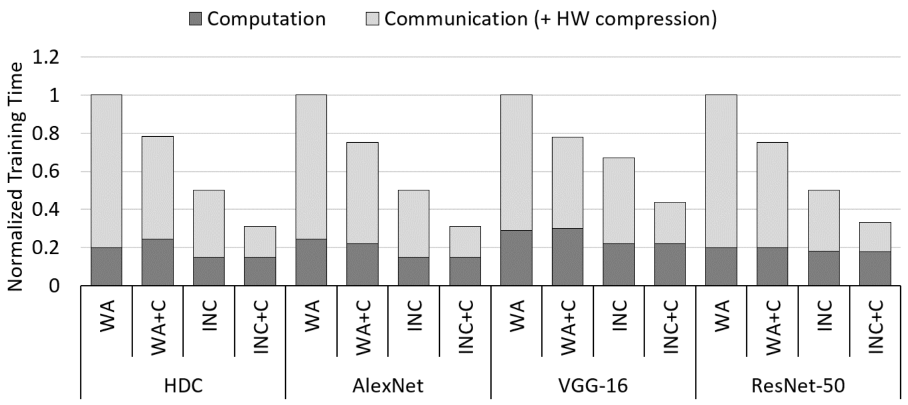


**Figure 3.11:** Comparison of training time between the worker-aggregator based approach (WAx) and the `INCEPTIONN` (INCx) with and without hardware-based compression in NICs. WA denotes the worker-aggregator baseline without compression, and WA+C denotes WA integrated with our compression only on gradient communication with an error bound of $2^{-10}$. INC denotes `INCEPTIONN` baseline without compression, and INC+C denotes with our compression given an error bound of $2^{-10}$. Training time is measured in a cluster of four workers for INCx and one more aggregator for WAx. Note that these measurements are based on the same number of training iterations.

of all the links among nodes. Besides, this balanced gradient exchange also contributes to the reduction of computation time as the gradient summation is done by all the nodes in a distributed manner, whereas the worker-aggregator based algorithm burdens the designated aggregator nodes to perform the aggregation of the gradients collected from a group of subnodes.

Furthermore, Fig. 3.11 compares the training time of the reference design and `INCEPTIONN` system, when both are equipped with our gradient compression (**WA+C, INC+C**). From the result, we see that the conventional worker-aggregator based approach can still benefit from our compression with a $\sim$ 30.8% reduction in communication time compared to its baseline (**WA**), although only one direction of communication is applicable for compression. On the other hand, our gradient-centric `INCEPTIONN` algorithm offers maximized compression opportunities such that `INCEPTIONN` with hardware compression (**INC+C**) gives $\sim$ 80.7% and $\sim$ 53.9% lower communication time than the conventional worker-aggregator baseline (**WA**) and `INCEPTIONN` baseline (**INC**), respectively. Therefore, the full `INCEPTIONN` system (**INC+C**) demonstrates a training time speedup of 2.2 $\sim$ 3.1$\times$ over the conventional approach (**WA**) for the four models trained over the same number of epochs.

### 3.7.2 Effect of INCEPTIONN Compression on Final Accuracy

The accuracy loss in gradients due to lossy compression may affect the final accuracy and/or prolong the training because of the necessity to run more epochs to converge to the lossless baseline accuracy. To understand the effect of our lossy compression on accuracy (and on prolonged training), we take the conventional worker-aggregator system (**WA**) and the `INCEPTIONN` system (**INC+C**), and train the models until both systems converge to the same level of accuracy. Table 3.3 presents the total number of epochs and the final speedup of `INCEPTIONN` system over the conventional training system to achieve the same level of accuracy. From this, we observe that only a modest number of more epochs (1 or 2) are required to reach the final accuracy and thus `INCEPTIONN` system still offers a speedup of 2.2× (VGG-16) to 3.1× (AlexNet) over the convention approach, which matches the performance in Sec. 3.7.1. Furthermore, we find that the extra number of training epochs is small but essential, without which an accuracy drop of $\sim 1.5\%$ might incur.

### 3.7.3 Evaluation of INCEPTIONN Compression Algorithm

Figure 3.12 compares the compression ratios among various lossy compression schemes, and the impact of these compressions on relative prediction accuracy of DNNs which are trained through our training algorithm for the same number of epochs. Specifically, we evaluate truncation of 16, 22, and 24 LSBs of gradients and `INCEPTIONN` compression with the absolute error bound of $2^{-10}$, $2^{-8}$ and $2^{-6}$. We observe that the naïve truncation of floating-point values only provides low constant compression ratios (*i.e.,* 4× at most)

**Table 3.3:** Speedup of `INCEPTIONN` over the conventional approach when both achieve the same level of accuracy. We use the same notations with Fig. 3.11.

| | AlexNet | | HDC | | ResNet-50 | | VGG-16 | |
|---|---|---|---|---|---|---|---|---|
| | WA | INC+C | WA | INC+C | WA | INC+C | WA | INC+C |
| Training Time | 175h | 56h | 170s | 64s | 378h | 127h | 847h | 384h |
| # of Epochs | 64 | 65 | 17 | 18 | 90 | 92 | 74 | 75 |
| Speedup | 1 | 3.125 | 1 | 2.66 | 1 | 2.98 | 1 | 2.2 |
| Final Accuracy | 57.2% | 57.2% | 98.5% | 98.5% | 75.3% | 75.3% | 71.5% | 71.5% |

while suffering from substantial accuracy loss (*i.e.,* up to 62.4% degradation in prediction accuracy of trained ResNet-50). This is due to the fact that the compression errors introduced by naïve truncation are uncontrolled and open ended. Moreover, the potential of truncation is limited by the length of the mantissa. Dropping more bits will perturb the exponent (*e.g.,* "24b-T" in Fig. 3.12), which results in a significant loss of accuracy of trained DNNs. In general, the truncation methods are only suitable for simpler DNNs such as HDC and are not suitable for complex DNNs such as AlexNet, VGG-16, or ResNet-50.

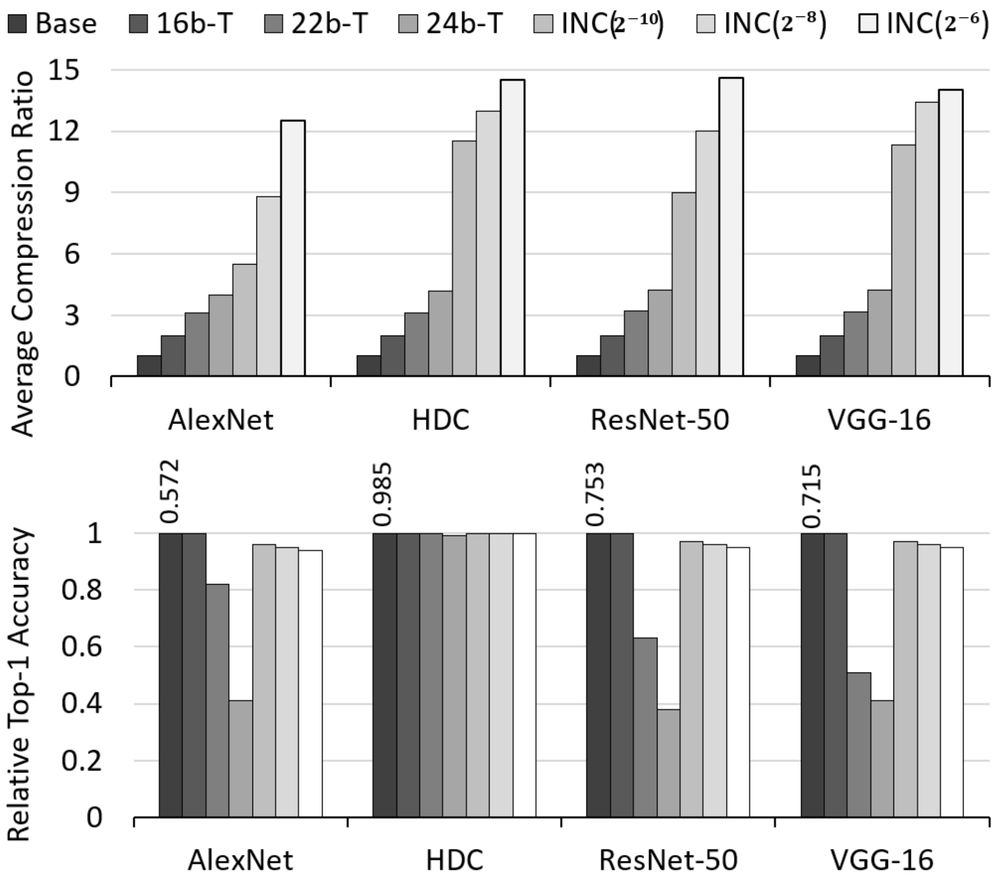


**Figure 3.12:** Comparison of (a) compression ratio and (b) impacts on prediction accuracy of DNNs trained by INCEPTIONN training algorithm with various lossy compression schemes. Note that the accuracy is based on the same epochs of training (without extra epochs) for each model. ("Base" denotes the baseline without compression. The number on top of each "Base" bar denotes the absolute prediction accuracy. $x$b-T represents truncation of $x$ LSBs. "INC" bars are the results of INCEPTIONN lossy compression with a given error bound.)

**Table 3.4:** The bitwidth distribution of compressed gradients. The compressed gradients are composed of two bits for indication tag and compressed data bits (0, 8, 16, or 32 bits).

| | | 2-bit | 10-bit | 18-bit | 34-bit |
|---|---|---|---|---|---|
| AlexNet | INC($2^{-10}$) | 74.9% | 3.9% | 21.1% | 0.1% |
| | INC($2^{-8}$) | 82.5% | 14.8% | 2.6% | 0.1% |
| | INC($2^{-6}$) | 93.0% | 7.0% | 0.0% | 0.1% |
| HDC | INC($2^{-10}$) | 92.0% | 6.5% | 1.5% | 0.0% |
| | INC($2^{-8}$) | 95.7% | 3.4% | 0.9% | 0.0% |
| | INC($2^{-6}$) | 98.1% | 1.6% | 0.4% | 0.0% |
| ResNet-50 | INC($2^{-10}$) | 81.6% | 17.9% | 0.5% | 0.0% |
| | INC($2^{-8}$) | 92.3% | 7.7% | 0.1% | 0.0% |
| | INC($2^{-6}$) | 97.6% | 2.4% | 0.0% | 0.0% |
| VGG-16 | INC($2^{-10}$) | 94.2% | 0.9% | 4.9% | 0.0% |
| | INC($2^{-8}$) | 96.2% | 3.8% | 0.0% | 0.0% |
| | INC($2^{-6}$) | 97.3% | 2.7% | 0.0% | 0.0% |

In contrast, our lossy compression shows much higher compression ratios (*i.e.,* up to 14.9×) as well as better preserves the training quality than the truncation cases even for those complex DNNs. Figure 3.12 shows that the errors induced by compression are well controlled by our algorithm and the average compression ratios are boosted by the relaxation of a given error bound. With the most relaxed error-bound ($2^{-6}$), almost all benchmarks demonstrate a compression ratio close to 15× and the final accuracies of trained DNNs are only degraded slightly, *i.e.,* $< 2\%$ in absolute accuracy. Note that this slight drop of accuracies incurs only when DNNs are trained for the same number of epochs as their lossless baselines and such drop can be easily compensated by negligible extra epochs of training, as discussed in Sec. 3.7.2.

To further understand the significant gains from our compression algorithm, we analyze the bitwidth distribution of compressed gradients. Table 3.4 reports the collected statistics. When the error bound is $2^{-6}$, for all the evaluated models, our algorithm compresses larger than 90% of gradients into two-bit vectors. Even with $2^{-10}$ as the error bound, 75% to 94% of gradients are compressed into the two-bit vectors. Leveraging this unique value property of gradients, our lossy compression algorithm achieves significantly larger compression ratio than general-purpose compression algorithms.

Lastly, we find that the compression ratio of the gradients is not necessarily proportional to the reduction in communication time, as shown in Fig. 3.11

where the compression with an error bound of $2^{-10}$ should have compressed the communication time by a factor of $5.5 \sim 11.6\times$. This is because we do not reduce the total number of packets and the network stack overhead such as sending network packet headers remains the same. Consequently, the use of more relaxed error bounds (*e.g.,* $2^{-8}$ and $2^{-6}$) only provides marginally additional reduction in the overall communication time.

### 3.7.4 Scalability Evaluation of INCEPTIONN Training Algorithm

We also evaluate the scalability of our `INCEPTIONN` training algorithm by extending our cluster up to eight worker nodes. Since we had only four GPUs available at our disposal, we only measured the gradient exchange time for the scalability experiments. The gradient exchange time consists of both gradient/weight communication and gradient summation time, and represents the metric in the scalability evaluation, because only communication and summation overheads scale with the number of nodes, while the time consumed by other DNN training steps such as forward pass, backward pass, weight update are constant due to their local computation nature.

Figure 3.13 compares the gradient exchange time between the `INCEPTIONN` baseline (**INC**) and the worker-aggregator baseline approach (**WA**), both without compression across different number of worker nodes. As shown in Fig. 3.13, the gradient exchange time increases almost linearly with the number of worker nodes in the WA cluster; however, it remains almost constant in the `INCEPTIONN` cluster, especially when training larger models such as AlexNet, VGG-16, and ResNet-50 where the network bandwidth is the bottleneck. This phenomenon seems intuitive, since in WA cluster both the communication and summation loads congest the aggregator node, while the `INCEPTIONN` approach balances these two loads by distributing them evenly among worker nodes. Analytically, by adopting the communication models in [96], the gradient exchange time in a WA cluster is: $(1 + log(p)) \cdot \alpha + (p + log(p)) \cdot n \cdot \beta + (p - 1) \cdot n \cdot \gamma$, where $p$ denotes the number of workers, $\alpha$ the network link latency, $n$ the model size in bytes, $\beta$ the byte transfer time, and $\gamma$ the byte sum reduction time. In practice, for distributed DNN training,

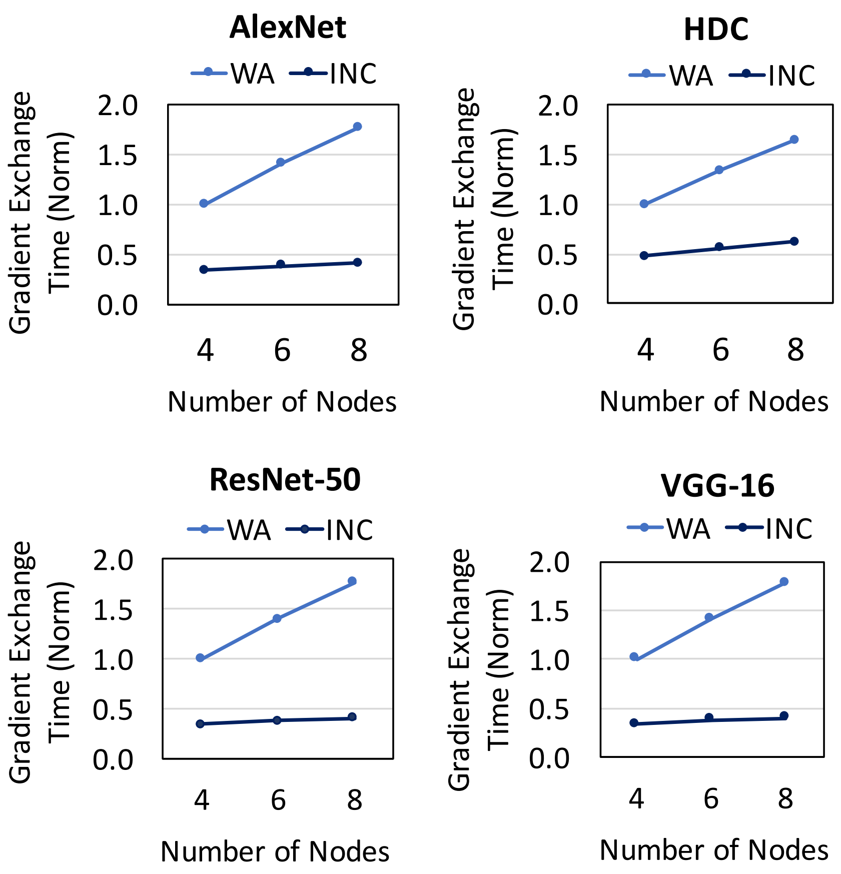


**Figure 3.13:** Scalability of `INCEPTIONN` training algorithm (INC) as compared to the conventional worker-aggregator based algorithm (WA) with different number of worker nodes. Gradient exchange time consists of both gradient/weight communication and gradient summation time. All values are normalized against four-node WA case.

the first term is negligible compared to the second and third term, due to the large model size $n$ and the limited network bandwidth $\beta$. The above equation explains clearly why the conventional WA approach is not scalable with increasing number of nodes $p$, *i.e.,* the gradient exchange time is linear in cluster size. In contrast, the communication-balanced `INCEPTIONN` offers the gradient exchange time of: $2(p-1)\cdot\alpha + 2(\frac{p-1}{p})\cdot n\cdot\beta + (\frac{p-1}{p})\cdot n\cdot\gamma$, where the effect of large cluster size $p$ cancels in the second and third terms, making `INCEPTIONN` much more scalable.

## 3.8 Related Work

**Acceleration for ML.** There has been a large body of work that leverage specialized accelerators for machine learning. Most of the work have concentrated on the inference phase [71, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112] while `INCEPTIONN` specifically aims for accelerating the training phase. Google proposes the TPU [71], which is an accelerator with the systolic array architecture for the inference of neural networks. Microsoft also unveiled Brainwave [110] that uses multiple FPGAs for DNN inference. Eyeriss is also an accelerator for CNN inference of which compute units set a spatial array connected through the reconfigurable multicast on-chip network to support varying shape of CNNs and maximize data reuse.

While the inference phase has been the main target of ML acceleration, the community has recently started looking into the acceleration of training phase [71, 113, 63, 114, 115]. ScaleDeep [113] and Tabla are ASIC and FPGA accelerators for the training phase while offering higher performance and efficiency compared to GPUs, which are the most widely used general-purpose processors for ML training. Google Cloud TPU [71] is the next-generation TPU capable of accelerating the training computation on the Google's distributed machine learning framework, Tensorflow [116]. CoSMIC [63] provides a distributed and accelerated ML training system using multiple FPGA or ASIC accelerators. Others [114, 115] focus on the acceleration of neural nets training with approximate arithmetic on FPGA. These ML training accelerators are either single-node solutions or accelerators deployed on the centralized training systems based on worker-aggregator approach, while `INCEPTIONN` provides a decentralized gradient-based training system and an efficient in-network gradient compression accelerators.

**Gradient reduction techniques.** There has been a series of work that proposes techniques for gradient reductions [73, 74, 75, 118, 62]. Quantization techniques for gradients [73, 74, 75, 118] provide algorithmic solutions to reduce the gradient precision while preserving the training capability. Deep Gradient Compression [62] is a complementary approach that reduces the amount of communication by skipping the communication of the gradients in each iteration. It will only communicate the gradients if the locally accumulated gradient exceeds a certain threshold. These works do not change the worker-aggregator nature of distributed training, nor propose in-network acceleration of compression.

## 3.9 Conclusion

Communication is a significant bottleneck in distributed training. The community has pushed forward to address this challenge by offering algorithmic innovations and employing the higher speed networking fabric. However, there has been a lack of a solution that conjointly considers these aspects and provides an interconnection infrastructure tailored for distributed training. `INCEPTIONN` is an initial step in this direction that co-design hardware and algorithms to provide an in-network accelerator for the lossy compression of gradients and maximize its benefits by introducing a decentralized distributed training algorithm.

# Part II

# Near-Memory Processing Using Commodity DRAM

Near-memory processing was in fact a hot research topic in late 1990 and early 2000 [119, 16, 120, 13]. Many papers has been published in that time frame, proposing to integrate DRAM and processor logic on a same die. However, none of those proposals got commercialized because of two main reasons. First, integrating DRAM and processing logic was too disruptive for chip manufacturing. The transistors parameters that are used for memory devices are different from the ones used for processing logic, and integration of DRAM and logic on a same chip compromises the reliability of the memory or the speed of the CPU [121]. Moreover, such proposals lack a standard interface for programming the processing logic on the memory which was unattractive for the industry [17]. Introduction of the 3D stack memory technologies in late 2000s renewed interest in the declining near-memory processing research. Researchers proposed to enable the logic layer in 3D DRAM produces to perform computation. The logic layer then could utilize the abundant memory bandwidth of Through Silicon Vias (TSVs) to accelerate memory intensive applications, such as graph processing or machine learning kernels, by offloading the memory intensive regions of the code to the near-memory processors. Although 3D stack memory technology resolves the difficulties of integrating memory and logic on a same die, it still suffers from lacking a standard interface for programming the near-memory processors. Moreover, Datacenters are optimized for cost while the cost per capacity of 3D stack memory devices cannot match that of the commodity DRAM products. Above all, the logic layer underneath the DRAM layers has a limited thermal design power (TDP) and area which makes it impossible to implement high-performance processing logic (such as out-of-order processors) in the logic layer [122]. All these reasons together makes near-memory proposals based on 3D stacked memories only attractive for mobile platforms with small memory capacity and computing requirements.

DRAM cost is in a race with processor costs these days and its hard to tell which one cost more when assembling a high-end server. This is why it is important to use commodity DRAM modules in datacenters to reduce the hardware cost. In this part, we introduce "memory module based" near-memory architecture that enables near-memory processing on commodity DRAM modules. The key idea is to enhance the buffer device of a commodity DRAM module (*i.e.,* Dual Inline Memory Module (DIMM)) with processing logic. These specialized DIMMs can perform computation on the data stored in the main memory independently form the host CPU. More specifically, we introduce Memory Channel Network (`MCN`) in Chapter 4. `MCN` seamlessly integrates near-memory processing within a server with scale-out processing of data-intensive applications across servers in a datacenter. The memory address space of the near-memory processors in `MCN` DIMMs is separate from the host CPU address space and `MCN` DIMMs communicate with the host CPU using a message passing model. The address space separation and message passing communication model is motivated by the execution model of large-scale data-intensive applications. Such applications are often programmed on top of popular distributed computing frameworks such as MPI, Hadoop, and Spark where each worker process has its own address space. In Chapter 5 we introduce `NetDIMM`, which complements `MCN` architecture. One shortcoming in `MCN` is that in an `MCN`-enabled cluster, if an application does not fit inside one server, then `MCN` DIMMs need to communication with each other over the long latency PCIe NICs. Our measurements shows that over 90% of the network hardware latency is contributed by the PCIe bus and frequent memory copies between NIC, DRAM, and CPU. `NetDIMM` integrates a full blown NIC inside the buffer device of a commodity DIMM. Compared to a conventional PCIe NIC, `NetDIMM` reduces data movement for networking, completely removes PCIe bus from networking data path, and accelerate memory copies in the network software stack by supporting in-memory buffer cloning. `MCN` and `NetDIMM` together propose a novel, scalable server architecture with a revolutionary network architecture.

# CHAPTER 4

# APPLICATION TRANSPARENT NEAR-MEMORY PROCESSING OF DATA-INTENSIVE APPLICATIONS

The physical memory capacity of servers is expected to increase drastically with the deployment of the forthcoming non-volatile memory technologies. This is a welcomed improvement for the emerging data-intensive applications. For such servers to be cost-effective, nonetheless, we must cost-effectively increase computation throughput and memory bandwidth commensurate with the increase in memory capacity without compromising the application readiness. Tackling this challenge, we present Memory Channel Network (MCN) architecture in this chapter. Specifically, first, we propose an MCN DIMM, an extension of a buffered DIMM where a small but capable processor called MCN processor is integrated with a buffer device on the DIMM for near-memory processing. Second, we implement device drivers to give the host and MCN processors in a server an illusion that they are independent heterogeneous nodes connected through an Ethernet link. These allow the host and MCN processors in a server to run a given data-intensive application together based on popular distributed computing frameworks such as MPI and Spark without any change in the host processor hardware and its application software, while offering the benefits of high-bandwidth and low-latency communication between the host and MCN processors over the memory channels. As such, MCN can serve as an application-transparent framework which can seamlessly unify the near-memory processing within a server and the distributed computing across such servers for data-intensive applications. Our simulation running the full software stack shows that a server with eight MCN DIMMs offers 4.56× higher throughput and consume 47.5% less energy than a cluster with nine conventional nodes connected through Ethernet links, as it facilitates up to 8.17× higher aggregate DRAM bandwidth utilization. Lastly, we demonstrate the feasibility of MCN with an IBM POWER8 system and an experimental buffered DIMM.

## 4.1 Introduction

The performance of servers running emerging data-intensive applications such as big-data analytic is often limited by the DRAM capacity and DDR bandwidth. The expected deployment of emerging memory technologies such as 3D XPoint [123] to servers will relieve the ever-increasing pressure on demanding larger memory capacity for such applications. However, for such servers to be cost-effective, we must increase the compute throughput and available memory bandwidth commensurate with the increase in memory capacity of servers.

As part of such effort, researchers have proposed various near-memory processing architectures that tightly integrate a processor with memory to expose higher bandwidth to the processor [15, 124, 125, 126, 127, 14, 128, 129, 130, 131, 132]. Such near-memory processing architectures, nonetheless, require significant changes in target applications especially to orchestrate the communication between the host and near-memory processors [16, 133, 15, 126]. This hurts application readiness and thus creates a big hurdle for wide adoption.

Tackling the application readiness challenge for near-memory processing, we start with an observation that many emerging data-intensive applications, which can benefit from near-memory processing, are often built upon distributed computing frameworks such as Hadoop [134], Spark [135] and MPI [136]. These distributed computing frameworks distribute given input data of an application and have many servers process the input data in parallel. As such, the high-level processing model of the recent near-memory processing architectures was inspired by the distributed computing frameworks [127, 15].

In this chapter, building on the distributed computing frameworks and exploiting high bandwidth and low latency of DDR interfaces, we propose Memory Channel Network (MCN). Specifically, MCN aims to give the host and near-memory processors connected through a DDR interface in a server the *illusion* that these processors are connected through Ethernet links. Therefore, MCN can provide a standard and application-transparent communication interface not only between the host and near-memory processors in a server, but also among such servers, seamlessly unifying near-memory processing with distributed computing for data-intensive applications. MCN consists of the following hardware and software components.

**(HW) MCN DIMM.** We architect an MCN processor and place it in a buffer device of a buffered DIMM (between a host-side Memory Controller (MC) and its associated DRAM devices on the DIMM). We refer to this buffered DIMM as MCN DIMM. For an MCN processor, we may take a small but capable Application Processor (AP), such as Qualcomm Snapdragon 835 [137], as the MCN processor and then implement a simple MCN interface in it. The MCN interface in the buffered device is similar to a network interface but takes a DDR PHY instead of an Ethernet PHY, interfacing between a host-side MC and an MCN processor. Lastly, each MCN processor runs a lightweight OS with the network software layers essential for running a distributed computing framework.

**(SW) MCN driver.** We develop a special device driver, referred to as MCN driver, for the host and MCN processors to give them the illusion that they are computer nodes connected through an Ethernet interface. An MCN driver is similar to a NIC driver, but it intercepts a network packet from the network software layer in the OS and redirects it to MCs instead of a NIC, if the network packet is destined to an MCN DIMM. Unlike a conventional NIC generating an interrupt to inform a host of new network packets, a memory interface (and MC) does not have a corresponding mechanism. Hence, we implement a special mechanism in the host-side MCN driver to determine whether any MCN DIMM is sending any network packet to the host or other MCN DIMMs.

These MCN DIMMs and associated drivers together allow a server to run an application based on a distributed computing framework *without any change in the host processor hardware, distributed computing middleware, and application software*, while offering the benefits of high-bandwidth and low-latency communications between the host and the MCN processors over memory channels. Furthermore, each MCN processor accesses its DRAM devices on the same MCN DIMM through its (local) memory channels isolated from the (global) memory channel shared with other DIMMs. Therefore, multiple MCN DIMMs can concurrently operate. That is, the aggregate memory bandwidth for processing is proportional to the number of MCN DIMMs. As such, MCN can serve as an application-transparent near-memory processing platform, as well as unify near-memory processing in a server with the distributed computing across multiple servers.

To further increase the utilized bandwidth and decrease the communication latency between MCN DIMMs, we propose optional software and hardware optimization techniques. Specifically, we optimize the MCN driver and some of the OS network layers, leveraging unique properties of MCN over the traditional Ethernet. We also show that additional communication efficiency can be achieved by changing the host-side MC to exploit a special signal of the recent and future memory interfaces. We use this signal to interrupt the host MC when an MCN DIMM has outgoing packets.

We model MCN DIMMs, develop MCN drivers, adapt some OS network layer, and demonstrate the full functionality in a full-system simulator running the entire software stack. Our evaluation shows that MCN offers 456.5% and 78.1% improvement in the network bandwidth and the latency compared with a conventional 10GbE network, respectively. Furthermore, an MCN-enabled server with 8 MCN DIMMs increases the aggregate memory utilization bandwidth for distributed applications by up to 8.17× compared with a conventional server.

Atop the simulation study, as a proof of concept, we take an experimental buffered DIMM which consists of an FPGA device and several DDR3 DRAM devices, implement an MCN processor on the experimental buffered DIMM, and demonstrate the feasibility of the MCN concept after plugging the DIMM into a memory channel of an IBM POWER8 system and installing the developed MCN drivers on the both IBM POWER8 host and the MCN DIMM.

## 4.2 Background

### 4.2.1 Memory Sub-System

**Buffered DRAM modules.** To strike a balance between memory capacity and bandwidth, multiple DRAM devices that operate in tandem compose a rank, and one or more ranks are packaged on a memory module. A popular memory module called Dual-Inline Memory Module (DIMM) has 64 data I/O (DQ) pins (plus 8 DQ pins for a DIMM supporting ECC). A memory channel connects an MC to one or more DIMMs. In a server class processor,

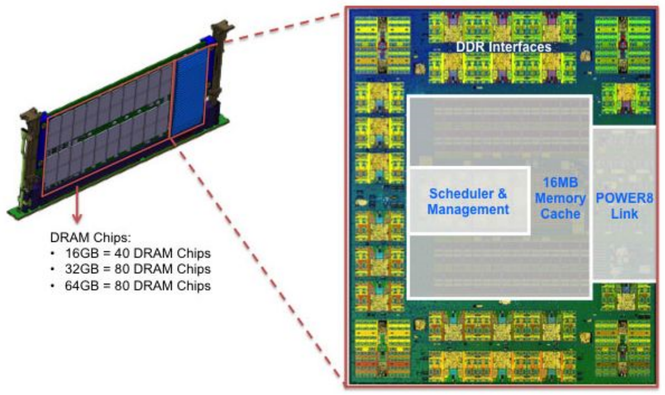


**Figure 4.1:** IBM Centaur DIMM.

an MC drives hundreds of DRAM devices and delivers Command/Address (C/A) signals through a memory channel to them. Considering the GHz operation frequency range of a modern DRAM device, this in turn leads to a serious signal integrity problem. For example, a C/A pin from a memory controller has to drive 144 DRAM devices (18×4 devices per rank supporting ECC multiplied by 8 ranks) when 8 ranks are populated per channel, whereas a data pin is connected to 8 DRAM devices, which is an order of magnitude fewer. Therefore, DIMMs for servers typically employ a buffer per DIMM, such as Registered DIMM (RDIMM) [138] or Load-Reduce DIMM (LRDIMM) [139], to reduce this huge capacitive load imposed to an MC and alleviate the signal integrity problem. Figure 4.1 depicts another DIMM type with a buffer, Centaur DIMM (CDIMM) [140]. Each CDIMM with a tall form factor comprises up to 80 commodity DDR DRAM devices and a Centaur device which provides a 16MB eDRAM L4 cache, memory management logic, and an interface between DDR and IBM proprietary memory interfaces. Note that the bandwidth available to the CPU remains constant as the memory channel is shared by all the DIMMs although the memory capacity increases with more DIMMs per channel.

**OS memory management for kernel space drivers.** For virtual to physical address mappings, an OS manages hierarchical page tables, each with two or more levels, depending on a processor architecture [141]. During the booting process, the Linux kernel is responsible for setting up page tables and turning on the Memory Management Unit (MMU). By default, the Linux kernel and users assume that any virtual page can be mapped to any physical page. However, it is desirable to **(D1)** reserve a specific range of physical memory space exclusively for an (memory-mapped) I/O device and its driver, and **(D2)** allow the driver to access this physical memory range with virtual addresses since every address issued by the processor is a virtual address after the MMU is turned on.

In the Linux kernel, we can accomplish **(D1)** by editing the Device Tree Blob (DTB). A DTB is a set of attributes of the hardware components in a given system and is fetched during the booting process. Specifically, a node in a DTB represents a hardware component and describes information such as the number and type of CPUs, base physical addresses and sizes of memory devices, I/O devices, etc. To reserve a specific region of physical memory, we create a new node in the device tree, wherein a physical address range is explicitly enumerated and is tagged as `reserved_memory`. At boot time, the kernel will exclude this physical address range from mapping to other processes, thereby creating a *memory map hole*. Later, the reserved memory region can be assigned to a device driver by setting the `memory_-region` parameter.

### 4.2.2 Network Sub-system[1]

**OS network layer.** TCP/IP is the most commonly used protocol for the distributed computing frameworks. An application sends and receives data through a TCP socket using `tcp_sendmsg()` and `tcp_recvmsg()` system calls, respectively. When a user application calls `tcp_sendmsg()`, the data is copied to a kernel buffer, fragmented into several segments of Maximum Transmission Unit (MTU) size, undergoes TCP/IP processing, and eventually sent to a NIC for transmission. The MTU limit exists since sending a packet a large data size at once is vulnerable to random transient errors in traditional physical links, such as the Ethernet links, and increases the probability and the overhead of re-transmitting the packet. In Linux, the default value of MTU is 1500 bytes. On the receiver side, the segments of a message are reassembled inside the Linux kernel and the complete message is copied to the user-space application.

**NIC and driver.** Figure 4.2 shows the interactions between a processor, memory, and a NIC when a network packet is received or transmitted. Once

[1]Please also refer to Sec. 2.2.2

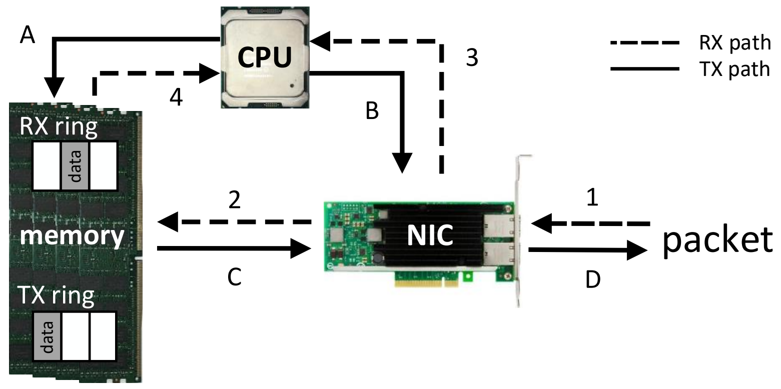


**Figure 4.2:** Network sub-system architecture.

an outgoing packet is processed in the TCP/IP stack, it is written to a transmission ring (TX ring) buffer (Ⓐ) in the physical memory. Then, the NIC driver informs the NIC of the available packets in the TX ring (Ⓑ). Later, the NIC reads the ready-to-transmit descriptors from the TX ring and DMA-transfers the data from the memory to the NIC buffers(Ⓒ). Finally, the packet is sent out (Ⓓ).

Similar to the TX ring, the NIC driver manages a circular ring buffer in the memory for the incoming packets (RX ring in Fig. 4.2). When a packet is received (❶), the NIC DMA-transfers the data to the next available buffer in the RX ring (❷). When the DMA-transfer is done, a HW interrupt is sent to the processor (❸). Upon receiving the HW interrupt, the NIC driver schedules a softIRQ. When the softIRQ handler eventually executes, it prepares a socket buffer by assembling the data inside the RX ring (❹) and send it to a higher network layer for further processing. Note that once a NIC starts receiving packets, switching to a polling-based approach is often preferred to a pure interrupt-based approach. This is because the performance cost of handling many hardware interrupts is notable which can bottleneck the throughput of a high bandwidth network [142].

## 4.3 Memory Channel Network

The overall design of MCN is depicted in Fig. 4.3. The MCN DIMMs and MCN drivers are designed with two key objectives in mind. First, they should run applications based on the existing distributed computing frameworks *without any change in the host processor hardware, distributed computing*

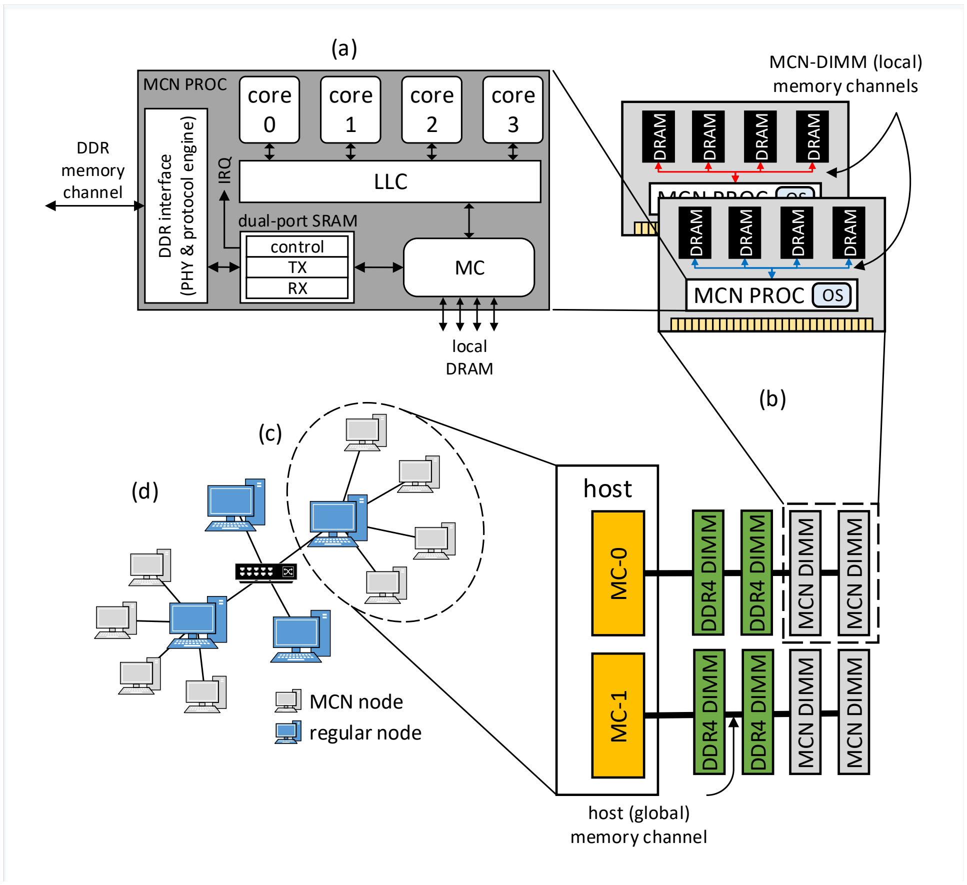


**Figure 4.3:** MCN Overview.

*middleware, and application software*. That is, MCN does not require any modification in the host processor and commodity DRAM architectures since it limits all hardware changes to a buffer device in a DIMM. Second, each MCN processor accesses its DRAM devices on the same MCN DIMM through its (local) memory channel isolated from the (global) memory channel which is shared with other DIMMs, as depicted in Fig. 4.3(b). Therefore, multiple MCN DIMMs can be concurrently accessed by an MCN processor through its local MCs, multiplying the aggregate memory bandwidth for processing [124, 143], as shown in Fig. 4.3(a) and (b). This is in contrast to a traditional memory sub-system, where the memory bandwidth for processing remains constant regardless of the number of DIMMs per memory channel; because multiple DIMMs share a global memory channel and the host processor can access only one DIMM at a time through the shared global memory channel. As such, MCN can serve as an *application-transparent near-memory processing platform*, as well as *unify the near-memory processing in a node* *with the distributed computing across multiple such nodes*, as illustrated in Fig. 4.3(d).

### 4.3.1 MCN DIMM Architecture

An MCN DIMM, also referred to as an MCN node, consists of an MCN processor and its associated DRAM devices. A host-side MC treats MCN DIMMs as buffered DIMMs (Sec. 4.2.1) and thus supports a mix of multiple MCN and conventional DIMMs per memory channel, as depicted in Fig. 4.3(b).

In this work, we propose to place a small low-power but capable mobile processor used in APs on a buffer device[2] of each DIMM. For example, four ARM Cortex A57 cores with 2MB LLC, implemented in Samsung Exynos 5433, consume ∼2mm×2mm space and Thermal Design Power (TDP) of ∼1.8W after scaling the size and power in 20nm technology [145] to the size and power in 10nm technology. A Qualcomm Snapdragon 835 AP incorporates a quad-core 2.45GHz ARM Cortex A57 CPU, a 710MHz Adreno 540 GPU, two 1866MHz (LP) DDR4 memory channels, and a UFS2.1 storage interface. Even with other components specific for mobile applications such as an LTE modem, a camera image signal processor (ISP), digital signal processors, etc., the Snapdragon 835 AP operates at TDP of 5W or less [137] and it is implemented on a small (∼8mm×8mm) die in 10nm technology [146]. Lastly, if the power constraint of DIMMs prevents us from taking more capable processors such as Tegra® SoC [147] for MCN DIMMs, then we can bring an external power cable to DIMMs as NVDIMMs [148] do.

Figure 4.3(a) depicts the MCN processor architecture which implements a DDR interface and a 96KB SRAM buffer in a typical quad-core mobile processor. A DDR interface consisting of DDR PHY and a protocol engine repeats DRAM C/A and DQ signals from/to a host MC as a typical buffer device does. It also performs two operations that are specific to the MCN. First, upon receiving a memory-write request from a host MC, it retrieves a command, a host physical memory address and 64-byte data from the captured C/A and DQ signals, translates the address to an SRAM address

[2]The size and TDP of an IBM Centaur buffer device is ∼10mm×10mm and 20W in 22nm technology [144]

and writes the data to the SRAM. Second, when servicing a memory-read request from a host MC, it performs operations similar to handling a memory-write request except that it reads data from the SRAM and generates DQ signals while following a given DDR protocol. Note that this DDR interface differs from the DDR interface between the MCN MC and DRAM devices on the MCN DIMM; we denote the former as the host DDR interface and the later as the MCN DDR interface. The SRAM serves as a communication buffer between the host and MCN processor, and is exposed to both the host and MCN processor as a part of their respective physical memory space, referred to as host and MCN physical memory space. The DDR interface and the SRAM together operate as an MCN interface similar to a NIC in a conventional node.

Figure 4.4 describes three regions of the SRAM buffer. We implement a circular TX buffer using `tx-start` and `tx-end` pointers, pointing to the start of the valid data and end of the valid data, respectively. Based on the area from McPAT in 22nm technology, we calculate that the size of this buffer is 0.074mm$^2$ in 10nm technology. TX and RX circular buffers store MCN messages which are sent to or received from the host processor, respectively. The `tx-poll` and `rx-poll` fields are used for handshaking between the host and MCN processors. We will describe the detailed usage of these control bits and the circular buffers in Sec. 4.3.2. When the OS network layer running on an MCN processor sends a network packet, the MCN driver, which is perceived as a regular Ethernet interface (Sec. 4.3.2), sends the network packet to a specific MCN physical memory address, where the SRAM buffer is mapped. When the MCN MC receives any memory request to the MCN physical memory space corresponding to the SRAM buffer, it re-directs the memory request to the SRAM buffer, which is connected to the MCN MC through an on-chip interconnect, instead of sending it to the DRAM devices on the MCN DIMM.

Lastly, similar to a conventional NIC, we implement a HW interrupt mechanism in the MCN interface to notify the MCN processor of any received packet in a SRAM RX buffer (`IRQ` in Fig. 4.3(a)). Upon receiving an interrupt from the MCN interface, the MCN processor starts a transfer of the packets from the RX SRAM buffer to the kernel memory space of the MCN driver using `memcpy` function. The memory copy operation can be accelerated using a custom DMA engine (Sec. 4.4.2).

### 4.3.2 MCN Drivers

The MCN drivers run on both the host and the MCN DIMMs to create an illusion of the existence of an Ethernet interface between the host and MCN processors. An MCN driver exposes itself as a regular Ethernet interface to the upper OS network layers, therefore, MCN does not require any changes in the OS network stack. This is a key advantage for MCN as there is a resistance towards the changes in the TCP/IP stack [149, 150].

**Network organization.** As shown in Fig. 4.3(b), we can populate a host memory channel with multiple MCN DIMMs (also referred to as MCN nodes). The host-side driver (*i.e.,* the driver running on the host processor), creates a virtual Ethernet interface for each MCN node installed on the host memory channels. That is, a virtual point-to-point connection is provided between the host and each MCN node, as shown in Fig. 4.3(c). We refer to each of the virtual Ethernet interfaces created on the host as a *host-side interface*. We then assign a MAC address, which is a unique 48-bit ID assigned to a network device, to each virtual Ethernet interface. Note that an MCN-side driver (*i.e.,* a driver running on an MCN processor) creates one virtual Ethernet interface, as an MCN node only has one point-to-point connection to the host. We refer to a single virtual Ethernet interface created on an MCN node as an *MCN-side interface*.

To facilitate the MCN communication, we assign an IPv4 address [151] to each of the host-side and MCN-side interfaces. From the host point of view, all of the MCN nodes are locally connected. We assign a unique IP addresses to each host-side interface (and the corresponding MCN node) and set the subnet mask of each interface to 255.255.255.255. This means that the host forwards a packet to a host-side interface if and only if the entire destination-IP address of the packet matches with the IP address of the interface. However, an MCN node does not have a direct connection to the other MCN nodes and outside world. Therefore, a packet that is generated from an MCN node and is destined to another MCN node (or outside world), has a different destination-IP address than the host's IP address. To support MCN to MCN and MCN to outside world accesses, we set the subnet mask of the MCN-side interfaces to 0.0.0.0. This means that all the outgoing packets from an MCN node is forwarded to the host, regardless of its IP address. Note that within an MCN node, a packet with its destination-IP

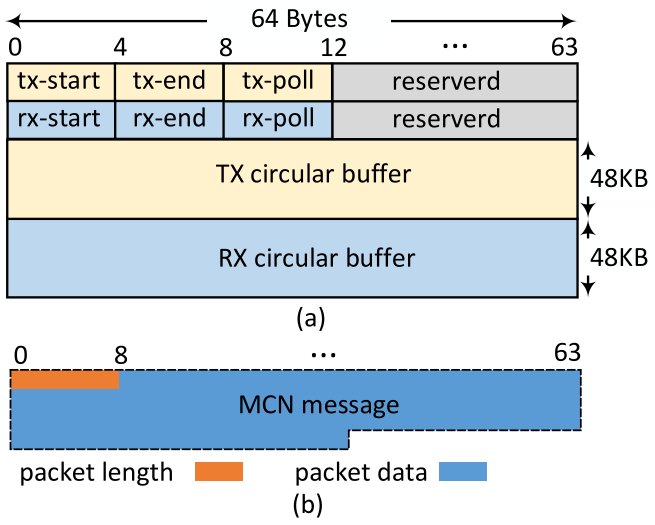


**Figure 4.4:** SRAM buffer of MCN interface.

set to localhost[3] does not get forwarded to the host as the kernel first checks if a packet belongs to a loopback network interface; if there is no match, then it enumerates the other available interfaces.

This setup ensures that the host arbitrates all the traffic to the MCN nodes, including the traffic between the MCN nodes. This network organization also supports the communication between MCN nodes connected to different hosts by having the source host to forward the packet to the host of the destination MCN node through a conventional NIC.

**Driver.** Figure 4.5 illustrates that the MCN-side driver is composed of three main components: **(C1)** a packet forwarding engine, **(C2)** a memory mapping unit, and **(C3)** a polling agent. Upon initialization, the network driver creates a network device object, sets it up as an Ethernet device, and registers the device with the kernel, thereby making a network interface visible to the host OS. The memory mapping unit accounts for the memory interleaving across different host memory channels and ensures that the physical address space of the SRAM buffers are accessible to the host and MCN processors through the virtual memory. Finally, the polling agent is responsible for periodically polling the SRAM buffers to check for new incoming packets.

**Packet transmission and reception.** In this sub-section we explain the

[3]In the IPv4 standard, 127.0.0.0 through 127.255.255.255 addresses are reserved for loopback purposes

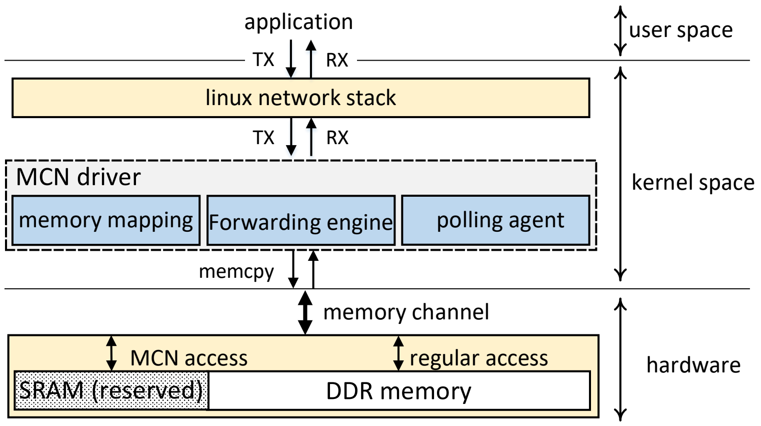


**Figure 4.5:** Overview of MCN driver.

flow of sending a packet from an MCN node and receiving it on the host. Since the host and MCN nodes run symmetric drivers, except for some minor differences, the flow is mirrored for a host to send a packet to an MCN node.

The transmission flow of a packet received at the MCN-side driver from the network stack is as follows (please refer to Fig. 4.4). (**T1**) Read `tx-start` and `tx-end` from the SRAM buffer. (**T2**) If there is enough space available in the TX buffer, write the packet length followed by the packet data to the TX memory space, starting from the address pointed by `tx-end`. As shown in Fig. 4.4, we call the combination of a packet length and data an *MCN message*. (**T3**) Update `tx-end` and also set `tx-poll` to a non-zero value, indicating that a new packet is enqueued in the TX buffer. Memory fences are used to ensure that the packet data has been copied correctly, prior to setting the control bits. Note that, if there is not enough space available in the TX buffer, the driver returns `NETDEV_TX_BUSY` as described in $< linux/netdevice.h >$.

Because a conventional DDR interface does not provide a signal that can serve as an interrupt or allow a transaction to be initiated by a DIMM, we propose to adopt a (host-side) polling agent to notify the host processor of incoming packets as a high-speed NIC do. The polling agent periodically reads the `tx-poll` field of the SRAM across all the MCN nodes, checking whether there are any pending packets in any of the MCN nodes. If a pending packet is detected, the host-side driver follows the following steps to receive a packet. (**R1**) Read the `tx-start` and `tx-end` pointers. (**R2**) Read the cache line at `tx-start`. (**R3**) Retrieve the packet length and the packet destination MAC address (`dst-mac`). In an Ethernet packet, the first six bytes of the data construct the destination MAC address [152]. (**R4**) Send the packet to the *packet forwarding engine*. (**R5**) If the `tx-start` pointer moved by the number of bytes read from the TX SRAM buffer is not equal to `tx-end`, it means that there are still available packets in the TX buffer, so start over from `R2`. Otherwise, reset `tx-poll`, and then exit.

**Packet forwarding engine.** As discussed earlier, the host processor is responsible for routing packets between MCN nodes. When the host receives a packet from either another host or an MCN node, it first inspects the packet to check its destination MAC address (`dst-mac`). Depending on the value of the `dst-mac`, we have one of the following scenarios. (**F1**) `dst-mac` matches the MAC address of the host-side interface: Allocate a socket buffer (`sk_buff`), copy the received packet data from the RX SRAM buffer to the `sk_buff`, and send it up to the network stack for processing. (**F2**) `dst-mac` is the reserved address for broadcast: Perform `F1` and `F4` actions. Also transmit the received packet to all the connected MCN nodes, except the source node, by performing `T1-T3` steps for each MCN node. (**F3**) `dst-mac` matches the MAC address of one of the MCN-side interfaces: Transmit the received packet to the destination MCN node by performing `T1-T3` steps. (**F4**) `dst-mac` does not match the host interface or any MCN-side interfaces: The packet is sent to a conventional NIC interface using `dev_queue_xmit` function in Linux kernel. If a packet is received at an MCN-side interface, MCN always sends the packet up to the network stack for processing by taking the actions explained in `F1`.

**Memory mapping unit.** By default, `ioremap` (Sec. 4.2.1) creates a page mapping that is tagged as `uncacheable` in the ARM architecture. While this prevents the coherency issues, the maximum size of a memory access to an uncacheable memory space is double word (*i.e.,* 64 bits). The memory access size along with the strict memory request ordering limit the memory bandwidth utilization. For the bulk memory transfers needed in MCN, it is desirable to access memory in cache line granularity. This can be done using `memremap` with a `MEMREMAP_WC` flag. This allows the MC's ability to perform a write combining which groups all consecutive write requests into a cache line granularity inside its write queue. On the other hand, read requests to consecutive memory addresses cannot be merged inside the MC read queue as it violates the memory consistency model. Thus, the MCN host-side driver

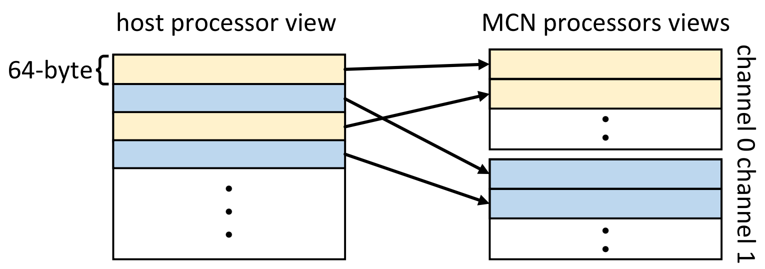


**Figure 4.6:** MCN memory interleaving for two channels.

uses an uncacheable memory mapping with the write combining support for the TX buffer and a cacheable memory mapping for the RX buffer. It explicitly invalidates the cache lines in the range of RX buffer after receiving a packet.

While accessing an MCN SRAM buffer, we must be cognizant of the memory channel interleaving performed by the memory sub-system, wherein the successive cache lines in the physical address space are mapped evenly across all the MCs of the host processor. This is to maximize the memory channel parallelism when there is spatial locality between the memory accesses. Without accounting for the memory interleaving, a naïve `memcpy` would incorrectly spread the packet data across MCN DIMMs in different memory channels although it should send them to a particular MCN DIMM's address space. To efficiently tackle this challenge, we propose `memcpy_to_mcn` and `memcpy_from_mcn` functions that perform memory copying such that the 64-byte blocks within the address space of the MCN DIMMs are interleaved in a manner that reflects the memory interleaving of the host processor. This allows the driver to send a packet data to an appropriate memory channel and thus MCN DIMM. Figure 4.6 illustrates an example of how `memcpy_to_mcn` and `memcpy_from_mcn` functions map a host processor view of the physical address space to an MCN processor view with two memory channels. As there is an MCN driver assigned to each memory channel and a typical distributed application sends packets to multiple (MCN) nodes, all the memory requests from these MCN drivers still concurrently utilize all the memory channels.

## 4.4 Bandwidth and Latency Optimization

In Sec. 4.3, we described a basic implementation to enable the MCN concept without any changes in the software stack and the host processor architecture. In this section, we identify some inefficiencies in the naïve MCN implementation and exploit some unique properties of a memory channel to further increase the bandwidth and decrease the latency of MCN. Specifically, we first propose to optimize the software stack which does not demand any hardware change. Second, we propose to optimize the memory sub-system architecture if we are permitted to slightly change the host processor architecture as well.

### 4.4.1 Software-Stack Optimization

In this section, we first exploit the features in the OS and conventional processors, and propose an efficient polling mechanism to reduce the communication latency between the host and MCN processors. Second, we exploit the fact that the Bit Error Rate (BER) of a memory channel is orders of magnitude lower than that of a network link and propose to bypass the checksum calculation to detect any error in a received packet and adopt a larger frame size for the packets.

**Efficient polling mechanism.** In Sec. 4.3.2, we proposed a naïve polling approach using a `tasklet`. However, a core running such a polling function can neither sleep nor accept a timer to reschedule. Consequently, it will overwhelm the core by rescheduling itself rapidly. To more efficiently support a polling mechanism, we propose to use a High-Resolution (HR) timer which reschedules a polling function call at a specific time with a nanosecond resolution. Specifically, whenever an HR-timer routine is invoked, it schedules a `tasklet` for running the polling function and then exits. Hence, any function called inside an HR-timer should be very short (*i.e.,* scheduling a `tasklet`). Note that a `tasklet` is interruptible and does not negatively impact a high priority process.

**Bypassing checksum.** The network stack inspects a Cyclic Redundancy Check (CRC) value or checksum of a packet to detect any error before it delivers the packet to the next network layer. Since the checksum calculation for each packet consumes the host and MCN processors cycles, it often limits the maximum bandwidth and the minimum latency. To reduce such an overhead, the network stack typically supports an interface to offload the checksum calculations to a hardware in the NIC. We propose a much simpler mechanism to efficiently handle checksum calculations. Since a memory channel is protected by an ECC (and CRC in DDR4), we do not need to redundantly generate a checksum value for an MCN message. Therefore, we disable the header checksum checking in the TCP/IP stack without affecting the reliability of the TCP packets.

**Large frame size and TCP segmentation offload (TSO).** The standard MTU of an Ethernet frame is 1.5KB, as discussed in Sec. 4.2.2. A larger MTU can better amortize the protocol processing software overhead and improve the network performance. Although the network stack can support a larger MTU, it often uses the default size as a larger packet going through the conventional Ethernet links is more likely to be corrupted and incur a higher cost for a re-transmission. However, MCN can efficiently deploy a larger frame size as the BER of a memory channel is typically multiple orders of magnitude lower than that of an Ethernet link. Exploiting such an advantage, we propose to increase the MTU of MCN to 9KB. This can be done by configuring the interface via the Linux `ifconfig` utility. The unique *MCN message* format described in Sec. 4.3.2 seamlessly supports any MTU size.

Even with a large MTU size, the network stack may still need to divide a bulk user data chunk into multiple MTU-sized packets. Each of these packets undergoes TCP/IP processing and pays the overhead of segmentation. To optimize bulk data transfer, modern NICs support TCP segmentation offload (TSO) [153], that offloads the segmentation to the NIC hardware. The driver of a TSO enabled NIC provides a TCP/IP header along with a large data chunk to the NIC. The TSO enabled NIC performs the following actions to send the data chunk. (**O1**) Divide the data chunk into several MTU sized segments. (**O2**) Copy the TCP/IP header at the beginning of each data segment. (**O3**) Calculate and set the `Total Length`, `Header Checksum`, and `Sequence Number` fields of each TCP/IP header [154, 151]. (**O4**) Send out each MTU sized packet. The MCN drivers support TSO by ensuring that there is sufficient space in the TX and RX buffers for the largest possible user data chunk allowed by the network stack. Since MCN can also bypass the checksuming, we simply set the `Total Length` field of the TCP/IP header to the user data chunk size and then transmit the unsegmented packet to the destination MCN node.

### 4.4.2 Memory Sub-System Optimization

In this section, we identify two bottlenecks to accomplish a higher bandwidth and lower latency in MCN: the lack of an interrupt mechanism to notify the host processor of the received packets from MCN DIMMs and a memory-to-memory copy accelerator to efficiently transfer the packet data from (to) the host processor to (from) an SRAM buffer in an MCN node. To tackle these limitations, we propose to slightly change the memory sub-system of the host processor as a set of optional optimization.

**Supporting interrupt from MCN DIMMs.** In Sec. 4.4.1, we proposed to adopt a high-resolution timer to more efficiently implement the polling agent. However, whenever the HR-timer is called, an interrupt is asserted, which incurs a performance overhead if the polling fails and no packet is received. If the timer interval is increased to minimize the overhead, then the average packet transmission latency increases as well. Additionally, upon receiving an HR-timer interrupt, the driver scans across all the MCN DIMMs on all channels, which further increases the overhead of the polling.

To further reduce the host-side polling overhead, we propose to leverage an existing interrupt-like signal (`ALERT_N` in the DDR4 standard [155]). Specifically, we may re-purpose an `ALERT_N` signal to serve as an interrupt from the MCN-DIMMs installed on a memory channel to the host processor. First, an MC receiving an `ALERT_N` from a memory channel must identify which DIMM on the channel has asserted it. Second, the MC relays the signal to a core as an interrupt. Third, the host-side drivers of the MCN DIMMs installed on the memory channel poll the MCN DIMMs on the channel, as for the polling case. This mechanism not only eliminates the need for periodic polling, but also allows the MCN drivers to immediately know which memory channel it should check.

**Memory-to-memory DMA.** The host (MCN) processor is responsible for copying packets between SRAM buffers and the host (MCN) physical memory space with the MCN specific `memcpy` functions. Consequently, the host and MCN processors issuing many memory requests often become a bottleneck, especially when they exchange many packets. This bottleneck can be resolved by implementing MCN DMA engines (`MCN-DMA`) in the memory controller of both the host and MCN processors.

More specifically, we use one DMA engine for each MCN node and one for each memory channel of the host processor. While an MCN side DMA engine maintains only one RX and one TX ring buffers, a host side DMA engine maintains separate ring buffers for each of the MCN nodes installed on the corresponding memory channel. For the packet transmission, the MCN driver initiates the DMA transfer by writing the destination MCN node number (always set to 0 in an MCN-side driver) and the transfer size to the corresponding DMA engine configuration space. The DMA engine is cognizant of the memory channel interleaving and writes a packet from the TX ring to the corresponding MCN node address space. When an MCN node has a new packet, utilizing the DIMM interrupt mechanism explained in the previous sub-section, the DMA engine receives an interrupt from the MCN DIMM, reads the available packets from its address space, and populates the RX ring with the received data. When the DMA transfer is finished, the host MCN driver is interrupted. The newly arrived packets in the RX ring are read and routed based on the *packet forwarding engine* explained in Sec. 4.3.2.

## 4.5 Methodology

**Proof of concept demonstration.** As a proof of concept, we developed a prototype MCN system using an experimental buffered DIMM [156] and an IBM POWER8 S824L system shown in Fig. 4.7(a) and (b), respectively. The prototype MCN DIMM couples two 32GB DDR3-1066 DIMMs with an Intel (Altera) Stratix V FPGA that directly interfaces with the host memory channel, the IBM Differential Memory Interface (DMI). We implemented an MCN DIMM architecture based on a soft IP core, NIOS II embedded processor [157] acting as an MCN processor in the FPGA. We also implemented the MCN SRAM buffer with BRAM blocks, custom glue logic to connect the buffer with DMI/Avalon interface, and used Intel's Avalon [158] as the internal bus in the FPGA. Finally, we developed the MCN drivers for the IBM host processor and the NIOS II processor based

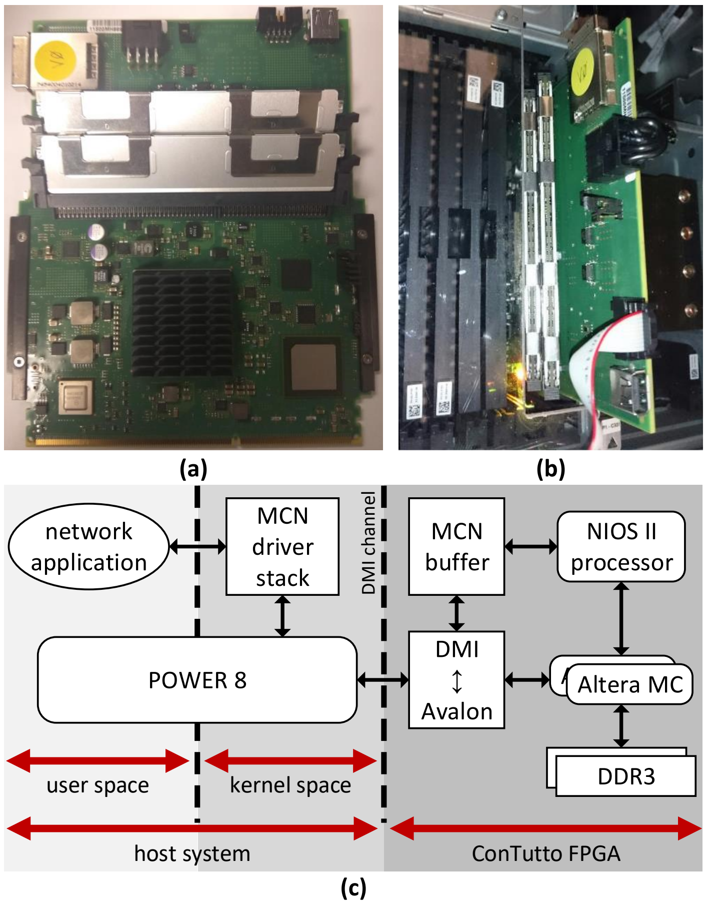


**Figure 4.7:** (a) ConTutto FPGA board (b) plugged into an IBM S824L system alongside regular CDIMMs. (c) MCN implementation block diagram.

on the descriptions in Sec. 4.3. Figure 4.7(c) depicts the prototype system architecture.

We use `McPAT` [47] in 22$nm$ technology for power estimation.

**Simulator and benchmarks.** A NIOS II processor implemented with FPGA and operating at 266MHz has very limited computing capability. Besides, we have only one experimental buffered DIMM. This prevents us from evaluating the full potential of MCN. Thus, we take a full-system simulation approach to further evaluate the effectiveness of MCN. To model a baseline distributed system with multiple nodes connected by 10GbE network, we first take `dist-gem5` [23, 159, 160] and run the entire software stack in the full-system mode. Second, we implement the MCN DIMM architecture in `dist-gem5` and port the MCN drivers implemented for the prototype system to a simulated processor architecture (*i.e.,* ARMv8 ISA).

**Table 4.1:** System configuration.

| **Parameters** | **Values** |
|---|---|
| Cores (# cores, freq): MCN/Host | (4, 2.45GHz)/(8, 3.4GHz) |
| Superscalar | 3 ways |
| ROB/IQ/LQ/SQ entries | 40/32/16/16 |
| Int & FP physical registers | 128 & 192 |
| Branch predictor/BTB entries | BiMode/2048 |
| MCN Caches (size, assoc): I/D/L2 | 32KB,2/32KB,2/1MB,16ways |
| Host Caches: I/D/L2/L3 | 32KB,2/32KB,2/256KB,16/8MB,16 |
| L1I/L1D/L2 latency,MSHRs | 1/2/12 cycles, 2/6/16 MSHRs |
| DRAM | DDR4-3200MHz/8GB |
| Operating system | Linux Ubuntu 14.04 (kernel 4.3) |
| Network | 10GbE/1$\mu$s link latency |

**Table 4.2:** Different MCN configurations.

| | |
|---|---|
| *mcn0* | baseline MCN with HR-timer polling implementation |
| *mcn1* | *mcn0* + MCN DIMM interrupt mechanism |
| *mcn2* | *mcn1* + IPv4 checksum bypassing |
| *mcn3* | *mcn2* + MTU increasing to 9KB |
| *mcn4* | *mcn3* + enabling TSO |
| *mcn5* | *mcn4* + enabling `MCN-DMA` |

Table 4.1 summarizes the `dist-gem5` system configuration.

Various optimization levels we used for the evaluation is described in Table 4.2. The same notations are used in the figures in Sec. 4.6.1. We use `iperf` [161] to compare the achieved bandwidth of MCN with the baseline 10GbE network. We run one `iperf` server and four `iperf` clients that simultaneously communicate with the server. To the collect round-trip latency, we run `ping` with various payload size. We evaluate communication intensive benchmarks from `NAS Parallel Benchmark` (`NPB`) [162], `CORAL` [163], and `BigDataBench` [164] benchmark suites.

## 4.6 Evaluation

### 4.6.1 Network Bandwidth and Latency

Figure 4.8 shows the achieved `iperf` bandwidth of MCN with different optimization levels, normalized to that of 10GbE. We show the bandwidth

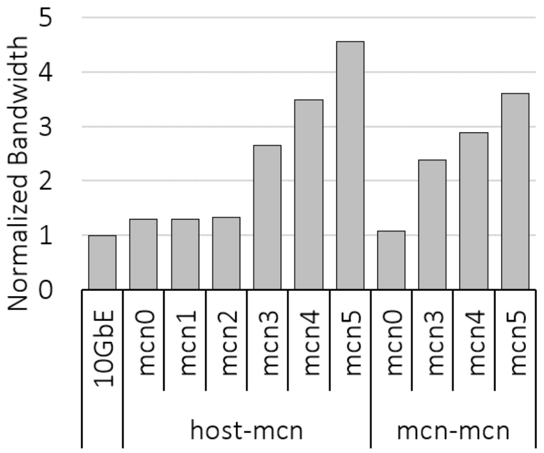


**Figure 4.8:** Network bandwidth for various MCN configurations when running iperf, normalized to 10GbE network.

for the following `iperf` setups: *host-mcn* runs the `iperf` server on the host and runs the `iperf` clients on the MCN DIMMs; *mcn-mcn* runs the `iperf` server on an MCN DIMM and runs the `iperf` clients on the host and the remaining MCN DIMMs. Compared with 10GbE, *mcn0*, which is the basic MCN implementation, improves the bandwidth by 30.3% and 7.8% for the *host-mcn* and *mcn-mcn* configurations, respectively. Replacing polling mechanism with interrupt does not have a notable effect on the achieved `iperf` bandwidth as `iperf` is not compute intensive and the polling agent does not interfere with the `iperf` processes. However, disabling IPv4 checksum calculation, increasing MTU size from 1.5KB to 9KB, enabling TSO, and enabling `MCN-DMA`, each in turn offer extra 3.0%, 99.6%, 31.4%, and 30.6% bandwidth improvements for *host-mcn* configuration.

In general, the achieved bandwidth of *host-mcn* is higher than *mcn-mcn* configuration. The reason is that there is no point-to-point communication channel between two MCN DIMMs and the *mcn-mcn* traffic have to be routed through the host-side MCN driver (Sec. 4.3.2). The achieved bandwidth of *mcn-mcn* is 10.5%, 17.2%, and 20.1% lower than *host-mcn* configuration when employing *mcn3*, *mcn4*, *mcn5* optimization levels, respectively.

Figure 4.9 illustrates the round-trip latency of a `ping (ICMP)` request, with various payload size, from host to an MCN DIMM (*i.e., host-mcn*), normalized to the round-trip latency of a 16-byte `ping` request between two hosts connected with a 10GbE network. As shown, MCN significantly

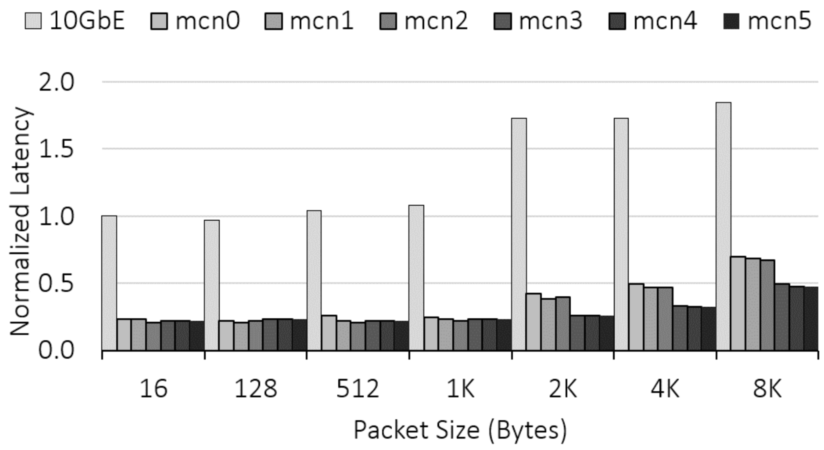


**Figure 4.9:** Round-trip latency between host and an MCN node normalized to 10GbE for different MCN configurations across various packet sizes.

reduces the latency between the nodes. For *host-mcn*, compared with 10GbE, *mcn0* reduces the round-trip latency by 62.2-75.4% across different packet sizes. Although the *mcn-mcn* communication is less efficient than the *host-mcn* configuration, the optimized MCN implementations always offers lower latency than 10GbE. As shown in Fig. 4.10, for *mcn-mcn* configuration, *mcn5* reduces the round-trip latency by 52.2-79.0% across different packet sizes, compared with 10GbE.

Table 4.3 reports the latency breakdown of different hardware/software components for 10GbE and *mcn0* when transmitting and receiving a TCP

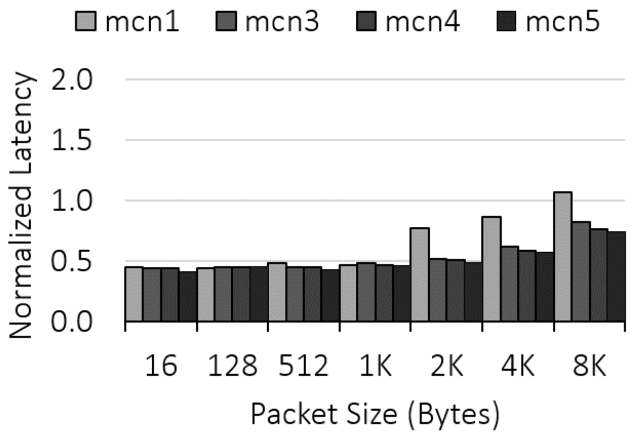


**Figure 4.10:** Round-trip latency between two MCN nodes normalized to 10GbE for different MCN configurations across various packet sizes.

Table 4.3: Breakdowns of the end-to-end latencies for transmitting and receiving a single TCP 1.5KB/9KB packet.

| Size | Type | Driver-TX | DMA-TX | PHY | DMA-RX | Driver-RX | Total |
|---|---|---|---|---|---|---|---|
| 1.5KB | 10GbE | 0.017 | 0.001 | 0.479 | 0.001 | 0.500 | 1 |
| | MCN-0 | 0.075 | 0 | 0 | 0 | 0.245 | 0.320 |
| 9KB | 10GbE | 0.019 | [0.503 (MEM-to-MEM)] | | | 0.478 | 1 |
| | MCN-0 | 0.073 | 0 | 0 | 0 | 0.692 | 0.765 |

packet with different size. For 10GbE, `Driver-TX` includes the time for setting up DMA and notifying NIC about a packet ready for transmission; `DMA-TX`/`DMA-RX` includes the time for transferring a packet from DRAM/NIC to NIC/DRAM; `PHY` includes the time spent in the PCIe bus, the NIC hardware, the Ethernet link, and the Ethernet switch; `Driver-RX` includes interrupting processor, clearing the RX ring buffer, and sending packet up to the network stack for processing. For MCN, `Driver-TX` includes the time that takes to write the packet to the RX buffer. MCN does not have `DMA-TX`, `PHY` and `DMA-RX` components and `Driver-RX` includes the overhead of polling and reading from the RX buffers. We normalize all the latency components to the 10GbE case. As illustrated in the Table 4.3, removing `PHY` is the biggest contributor to the end-to-end latency reduction for MCN. `Driver-TX` and `Driver-RX` are higher than 10GbE because now the host/MCN CPU manually copies the packets inside the drivers to/from the SRAM buffer instead of using a DMA engine.

### 4.6.2 Performance and Energy

Figure 4.11 shows the normalized aggregate memory bandwidth utilization of an MCN-enabled server with different number of MCN DIMMs. We normalize the aggregate bandwidth to the utilized memory bandwidth of the application when running on a conventional server. Across all applications, on average, a server with 2, 4, 6, and 8 MCN DIMMs improves the aggregate memory bandwidth by 1.76×, 2.6×, 3.3×, and 3.9× compared with a conventional server. This shows the effectiveness of MCN in scaling the aggregate memory bandwidth of a server as a near-memory processing framework. Note that adding regular DIMMs to the server just increases the memory capacity and the aggregate memory bandwidth remains unchanged.

Figure 4.12 shows the energy efficiency of MCN compared with a

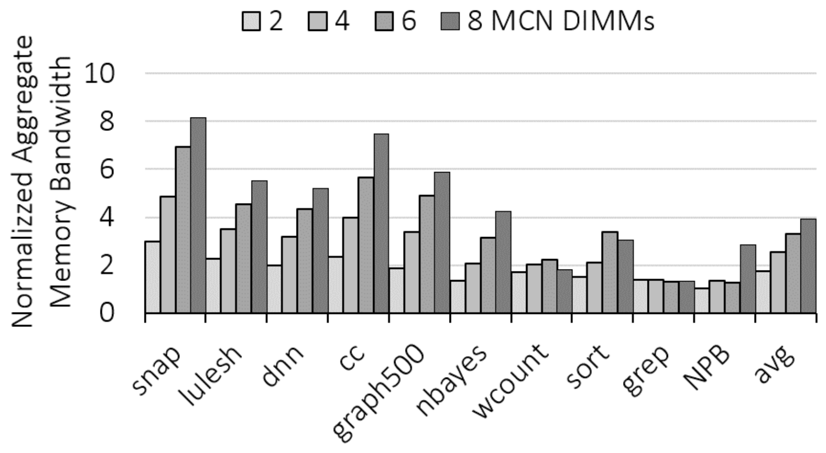


**Figure 4.11:** Aggregate memory bandwidth of an `MCN`-enabled server.

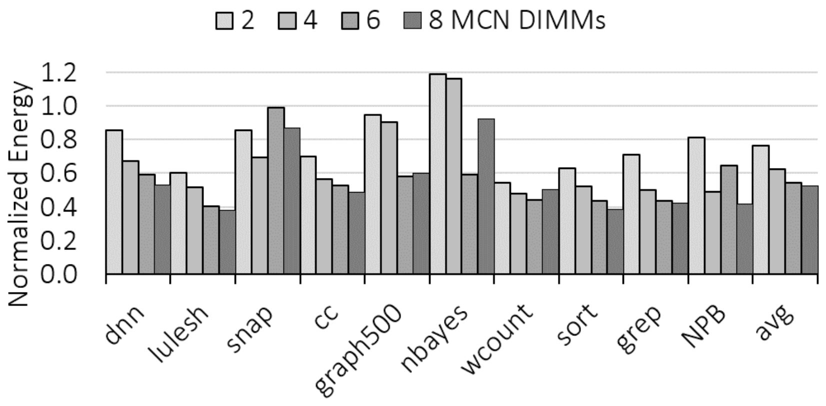


**Figure 4.12:** Energy efficiency of MCN.

conventional scale-out cluster connected through the 10GbE network. A corresponding scale-out system for an MCN server with 2, 4, 6, and 8 MCN DIMMs has 2, 3, 4, and 5 nodes configured with the parameters shown in Table 4.1, respectively; that is to compare the energy consumption of a scale-out system with an MCN server when the total number of cores in both setups is the same. We evenly distribute MCN DIMMs on the host memory channels. Across all the applications, on average, MCN offers 23.5%, 37.7%, 45.5%, and 57.5% lower energy consumption compared with a 2, 3, 4, and 5 node 10GbE scale-out cluster, respectively. Most of the data-intensive distributed applications are more energy-efficient to run on an MCN server with a mix of high-performance and near-memory mobile processors, compared with a scale-out counterpart with an identical number of cores [165]. However, not all the benchmarks show energy improvement when running on an MCN server. This shows the importance of supporting the conventional scale-out distributed computing and near-memory acceleration at the same time.

To show the effectiveness of MCN in scaling memory bandwidth, memory capacity, and compute capability of a conventional server all together, we compare the execution time of running `NPB` on a conventional server with running `NPB` on an MCN-enabled server with the same number of cores in both configurations. Figure 4.13 shows the execution time of MPI applications from `NPB` running on an scale-up setup, where it has 4, 8, 12, and 16 cores on a single chip, and on an MCN-enabled server, where it has 1, 2, or 3 MCN DIMMs installed. The execution time is normalized to running the application on a conventional server with 4 cores. "0," "1," "2," and "3" on the $x$-axis represents the baseline server with 4, 8, 12, and 16 cores and an MCN-enabled server with 0, 1, 2, and 3 MCN DIMMs, respectively. As shown in Fig. 4.13, on average, a server with 1, 2, and 3 MCN DIMMs improves the execution time of `NPB` applications by 27.2%, 42.9%, and 45.3% compared with running them on a scale-up setup with 4, 8, 12, and 16 cores. Note that increasing the number of MPI processes of an application does not always improves the execution time of the application, as for `mg` and `lu`. However, here our focus is on how the higher aggregate memory bandwidth of an MCN-enabled server impacts the execution time of an MPI application.

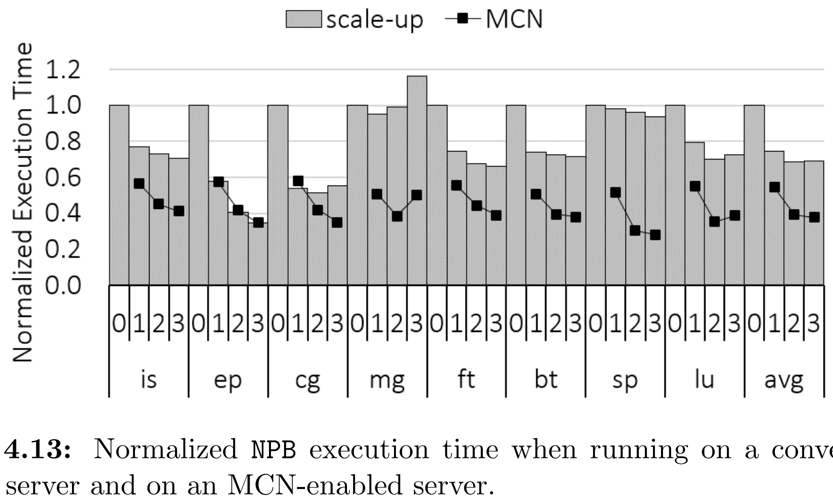


**Figure 4.13:** Normalized `NPB` execution time when running on a conventional scale-up server and on an MCN-enabled server.

We noticed that the performance of `ep` (Embarrassingly Parallel) is not sensitive to the memory bandwidth and it only scales with the number of MPI processes. Therefore, MCN does not provide any speedup for `ep`. Also, a scale-out server with 8 cores executes `cg` faster than a server with one MCN DIMM. `cg` performs many irregular communication between MPI processes. As the overhead of inter-process communication in a scale-up server is lower than an MCN server, the speedup of having higher aggregate memory bandwidth is offset by the overhead of frequent MCN to host communication.

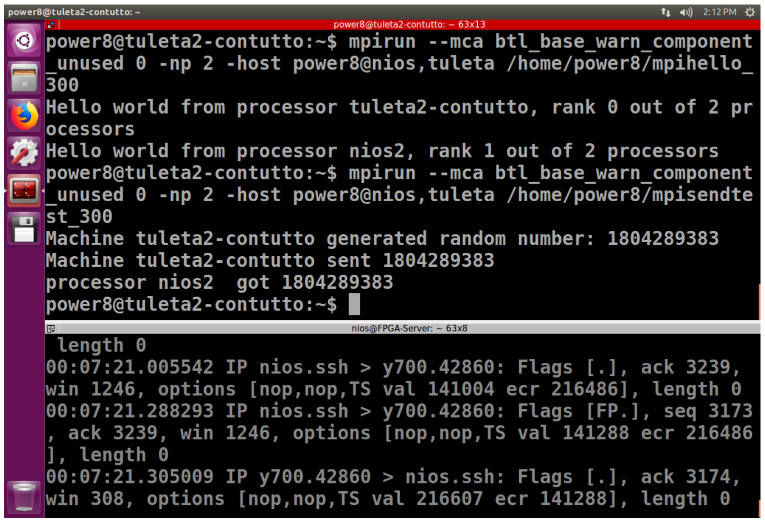
### 4.6.3 Demonstration with ConTutto Platform

To demonstrate support for existing distributed computing framework APIs, we have cross-compiled the latest version of `OpenMPI` (v3.0.0) for the NIOS II ISA. We successfully tested various MPI applications over MCN by treating the MCN node as a regular worker node with its own IP address in the local area network. The MPI application needs to be compiled separately for the POWER8 and NIOS II processors, but no modification is needed in the application's source code and the execution process is entirely transparent from the programmer's perspective. Figure 4.14 shows a screenshot of our prototype system running MPI. The NIOS II terminal is running `tcpdump` at the bottom half of the screen. Note that the purpose of building this proof-of-concept system is to demonstrate that the concept and developed driver work on a commercial system. As the prototype MCN DIMM is built using an FPGA and the MCN processor is a very low-performance soft IP core (NIOS II), we cannot obtain any meaningful performance evaluation.

### 4.6.4 Discussion

In Sec. 4.6.1, we demonstrated that MCN is capable of surpassing the performance of a conventional 10GbE network. However, with the given bandwidth and latency of the memory channel, we can potentially improve the MCN bandwidth to surpass higher bandwidth networks too.

A NIC employs several techniques to achieve high bandwidth: (T1) utilizes several offload engines [153, 166, 167]; (T2) uses highly optimized driver and OS software stack (*e.g.,* DPDK [168] or mTCP [169]), with special



```
power8@tuleta2-contutto:~$ mpirun --mca btl_base_warn_component
_unused 0 -np 2 -host power8@nios,tuleta /home/power8/mpihello_
300
Hello world from processor tuleta2-contutto, rank 0 out of 2 pr
ocessors
Hello world from processor nios2, rank 1 out of 2 processors
power8@tuleta2-contutto:~$ mpirun --mca btl_base_warn_component
_unused 0 -np 2 -host power8@nios,tuleta /home/power8/mpisendte
st_300
Machine tuleta2-contutto generated random number: 1804289383
Machine tuleta2-contutto sent 1804289383
processor nios2  got 1804289383
power8@tuleta2-contutto:~$ 
```

```
 length 0
00:07:21.005542 IP nios.ssh > y700.42860: Flags [.], ack 3239,
win 1246, options [nop,nop,TS val 141004 ecr 216486], length 0
00:07:21.288293 IP nios.ssh > y700.42860: Flags [FP.], seq 3173
, ack 3239, win 1246, options [nop,nop,TS val 141288 ecr 216486
], length 0
00:07:21.305009 IP y700.42860 > nios.ssh: Flags [.], ack 3174,
win 308, options [nop,nop,TS val 216607 ecr 141288], length 0
```

**Figure 4.14:** Screenshot of an MPI "Hello World" program running on our proof-of-concept system.

purpose network processing libraries such as RDMA; (T3) distributes the packet processing tasks on several CPU cores; and (T4) uses the aggregate memory bandwidth of a processor by interleaving DMA data across multiple memory channels.

We identified two bottlenecks toward utilizing MCN to its full capabilities. First, the TCP congestion control is implemented for slow, long latency network connections and sometimes takes several seconds to reach to the full bandwidth utilization. Also, TCP frequently sends ACK messages to the sender. Sending and receiving ACK messages consumes both CPU cycles and network bandwidth. Based on our evaluation results, sending and receiving ACK messages incurs up to ∼25% overhead in a TCP connection, which is aligned with the previous studies [170]. Second, an MCN DIMM can only use a single channel bandwidth and cannot interleave the memory accesses across multiple memory channels. That being said, the maximum theoretical MCN bandwidth is 12.8GB/s, which is the maximum bandwidth of a single memory channel. Although the bandwidth of each MCN node is limited to the bandwidth of a single memory channel, it is far from being a bottleneck as the bandwidth of a single memory channel alone is more than 100Gbps. Nonetheless, each MCN DIMM can communicate with the host or each other independently, providing aggregate bandwidth proportional to the total number of memory channels in the system.

As a future work, we consider the use of an specialized TCP/IP stack for MCN that resembles a *user space TCP* stack such as `mTCP`. When communicating between MCN DIMMs, the MCN network stack does not rely on the conventional TCP/IP stack and instead resembles a shared memory communication channel between the host and MCN nodes.

The network architecture of the current datacenters follows a hierarchical model with the servers as the leaf nodes. A rack, as the basic building block of a datacenter, consists of several servers connected together using a top of rack switch. As reported in several industry papers, the bandwidth of a top of rack switch ranges from 1 to 10Gbps, while the top of rack switches are connected together through 40 to 100Gbps connections [6, 171]. As shown in Sec. 4.6.1, even a basic MCN implementation provides higher bandwidth and lower latency than its 10GbE counterpart. We propose to replace a rack with an MCN-enabled server that interconnect leaf nodes (*i.e.,* MCN nodes) using a low-cost, energy-efficient interconnect to improve the energy efficiency of running IO intensive applications (Sec. 4.6.2) while reducing the datacenter cost.

## 4.7 Related Work

**Near-DRAM processing.** The traditional processor-in-memory (PIM) architectures integrate a processor and DRAM onto a single die [172, 173, 174, 13, 16, 119, 12]. These architectures can reduce energy consumption and increase the throughput of data transfers between the processor and DRAM, but suffer from high fabrication costs and low yields [175, 133]. The integration issue was mitigated by the emerging 3D die-stacking technology, reopening opportunities for near-DRAM processing architectures [176, 177, 143, 178, 127, 179, 180, 14, 15, 181, 128, 182, 183]. Among these architectures, NDA [178] 3D-stacks accelerators atop a DRAM device and is similar to MCN because both build on a standard DRAM interface and DIMM architecture. However, NDA requires a programmer to manually handle the communication between the host processor and accelerators using a dedicated programming model for the accelerators. As a cheaper alternative to using 3D die-stacking technology and providing large memory capacity, Chameleon [124] proposes to place accelerators in the buffer devices of DIMMs. It is similar to MCN because accelerators are integrated in a buffer device, but it suffers from the same limitation as NDA. In contrast, MCN is unique because it does not require technology integration, 3D die-stacking, or a new programming model.

**Cache coherent interconnect.** IBM's Coherent Accelerator Processor Interface (CAPI) [184] is a high speed communication standard, designed for I/O attached accelerators to work in a cache-coherent fashion with traditional processors. As CAPI leverages existing point-to-point PCIe I/O channels, it needs to rely on a customized hardware to manage the coherency and it cannot provide the bandwidth scaling benefit with more accelerator modules like MCN. Besides, CAPI relies on kernel extensions and a CAPI application library to expose the accelerator to the host application, thus requiring the user to modify or rewrite the application in order to leverage the accelerators. Intel Quick Path Interconnect(QPI) [185] supports a cache coherency protocol for attached devices. Although QPI is a high speed point-to-point interconnect, it suffers from the same limitations as PCIe (long latency, limited to one device per channel, etc.). Intel HARP [186] architecture leverages the QPI and couples an FPGA with an Intel processor. Like CAPI, accelerators in HARP have access to the cache coherency mechanisms and unified address space, but leveraging accelerators in HARP also requires using Intel provided APIs and libraries or using OpenCL, which would once again require modifying or rewriting the target application.

## 4.8 Conclusion

In this chapter, we proposed MCN consisting of MCN DIMMs and MCN drivers. MCN allows us to run applications based on distributed computing frameworks, such as MPI, without any change in the host processor hardware, distributed computing middleware and application software, while offering the benefits of high-bandwidth/low-latency communication between host and MCN processors. Furthermore, MCN can serve as an application-transparent near-DRAM processing platform since the memory bandwidth for processing multiplies with the number of MCN DIMMs. As such, MCN can unify the near-DRAM processing in a node with the distributed computing across multiple nodes. Our evaluation showed that a node with MCN can provide up to 58.7% higher performance than multiple conventional nodes connected by a 10GbE network when running various MPI-based distributed applications. Lastly, we demonstrated a proof of MCN concept with an IBM POWER8 system and an experimental buffered DIMM.

# CHAPTER 5

# ULTRA-LOW LATENCY NEAR-MEMORY NETWORK INTERFACE ARCHITECTURE

Optimizing bandwidth was the main focus of designing scale-out networks for several decades, and this optimization trend has served the traditional Internet applications well. However, the emergence of datacenters as single computer entities has made latency as important as bandwidth in designing datacenter networks. PCIe interconnect is known to be latency bottleneck in communication networks as its latency overhead can contribute to up to ∼90% of the overall communication latency. Despite its overheads, PCIe is the de facto interconnect standard in servers as it has been well established and maintained for more than two decades. In addition to PCIe overhead, data movements in network software stack consume thousands of processor cycles and make ultra-low latency networking more challenging. Tackling PCIe and data movement overheads, in this chapter, we propose `NetDIMM`, a near-memory network interface card capable of in-memory buffer cloning. `NetDIMM` places a network interface card chip into the buffer device of a dual in-line memory module and leverages the asynchronous memory access capability of DDR5 to share the memory modules between the host processor and near-memory NIC. Our evaluation shows `NetDIMM`, on average, improves per-packet latency by 49.9% compared with a baseline network deploying PCIe NICs.

## 5.1 Introduction

Traditionally, the main design requirement for scale-out networks was high bandwidth. To ensure fairness and avoid congestion, network transport protocols such as TCP [187] have thrived. For the past three decades, such network architecture has served well throughput oriented Internet applications such as file and email servers. Even for interactive web applications, such as web search, that are sensitive to the per packet delivery time, a response time of several hundreds of milliseconds is considered acceptable as long as it can satisfy a service level objective, often defined as $99^{th}$ percentile response time. This throughput oriented network design has driven the development of high bandwidth network devices such as 100Gb+ Ethernet network interface cards (NIC).

The proliferation of datacenters and emerging applications over the past few years has changed network design requirements. In addition to high bandwidth, low-latency communication has become a primary metric for evaluating the next generation of scale-out networks. Ultra-low latency applications such as in-memory caching, high-performance computing, and financial trading [188, 46, 189] benefit from even sub microsecond latency improvements in the network hardware and software stack.

Ethernet, as the backbone of datacenter networking technology, is tightly coupled with the TCP/IP protocol to ensure reliable and fair communication between nodes in a datacenter. The deployment of TCP offload engines [190, 191, 192, 193] along with more efficient implementation of the software stack [194, 195, 169, 196, 168, 197, 198] has significantly reduced the computational overhead in the software stack of Ethernet networks. For instance, RDMA over converged Ethernet (RoCE) protocol technically offloads the whole network software stack to the Ethernet NIC device by implementing a priority flow control inside the NIC to make the Ethernet lossless [199]. A RoCE network can achieve node to node latency as low as $\sim 1.3\mu$s [200] by minimizing the software stack overhead. These technological advancements have made it possible to achieve end-to-end network latency that is close to hardware limits.

PCIe is a widely used and well-established server I/O interconnect technology. PCIe is used to connect off-chip storage, network, and accelerator devices to the processor chip. A bleeding edge $\times 16$ PCIe Gen 4.0 provides a theoretical bandwidth of 31.51GBps. PCIe has a layered architecture and the protocol overhead at each layer reduces the usable bandwidth and adds to the latency overhead [201]. Therefore, PCIe interconnect is known to be the bottleneck especially in low-latency communication networks [202, 203, 204, 38, 8]. Frequent transactions over PCIe interconnect are the main contributor to the end-to-end network latency of software-stack optimized networks. For example, the PCIe sub-system contributes to

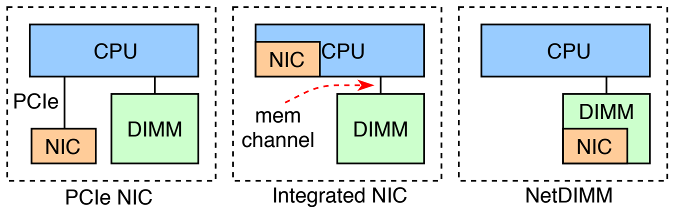


**Figure 5.1:** State-of-the-art network interface architectures vs. `NetDIMM`.

77.2∼90.6% of the overall network latency for sending packets of various size over an ExaNIC 10Gbps NIC [8]. Besides the PCIe overhead, data copying from DMA buffers to application memory space is a major bottleneck in network sub-system that can constitute 18∼92% of the per-byte operation overhead for different network protocols [9, 205].

To confront the PCIe and memory movement bottlenecks, previous works have proposed several solutions: (S0) reducing the number of PCIe transactions needed for packet transmission and reception [202, 203, 38], (S1) integrating the network interface card to the processor chip [206, 122], (S2) integrating a large memory buffer to the NIC [207, 208], (S3) adding processing units to the NIC and offloading the software to the NIC logic [209, 25], and (S4) developing zero copy networking [205]. Although these techniques can alleviate some of the network overhead, each has several drawbacks. S0 proposals mostly minimize the number of PCIe transactions for small packets. Moreover, the NIC is still connected to the processor through a PCIe interconnect and at least one round-trip over PCIe is needed for sending or receiving a packet. S1 designs are costly due to the area and power overhead for the processor chip. Furthermore, NIC and processor chips are often manufactured by different vendors and it is not practical to integrate them into one chip. Lastly, integrated NIC can pollute on-chip CPU resources when receiving large packet sizes (Sec. 5.5.3). Even though S2 and S3 can accelerate some applications, these techniques cannot benefit general-purpose applications and are often hard to manage/program. Moreover, such NIC architectures still suffer from PCIe overhead. Regarding S4, although zero copy networking eliminates the data copying from DMA to application buffers, it introduces several problems including security breaches, main memory exhaustion, and extra virtual memory operation overheads that can nullify its benefits [9, 210].

In this work, we propose **Net**work attached **DIMM** (`NetDIMM`), a novel near-memory network interface card that utilizes a high speed DDR5 channel to interconnect a near-memory NIC to the processor. `NetDIMM` integrates a NIC into the buffer device of a dual inline memory module (DIMM) and uses the low-latency, high-bandwidth memory channel to communicate with the processor. `NetDIMM` leverages the asynchronous memory access support of DDR5 specification to seamlessly expose its local memory capacity to the host processor as if it is part of the host processor address space. Furthermore, `NetDIMM` supports in-memory buffer cloning that provides the performance of zero copy networking without its drawbacks. More specifically, `NetDIMM` makes the following contributions:

- *Eliminate the PCIe bottleneck in the network sub-system.* `NetDIMM` uses the memory channel instead of PCIe link to interconnect a NIC to the processor.

- *In-memory acceleration of network stack data movements.* `NetDIMM` accelerates the DMA between NIC and DRAM by placing the NIC close to the DRAM modules. Furthermore, `NetDIMM` performs in-memory buffer cloning to accelerate data movements in the network stack.

- *Application-transparent network stack acceleration.* `NetDIMM` runs the kernel software stack with minimal modification in the Linux kernel. Therefore `NetDIMM` can run unmodified userspace applications.

- *Reducing memory interference from the network traffic.* `NetDIMM` reduces the host memory channel utilization by using the local memory channels of `NetDIMM` for transferring packets between the memory and NIC. `NetDIMM` also split header and payload of packets that reduces on-chip resource pollution.

Figure 5.1 compares `NetDIMM` with the state-of-the-art NIC architectures. `NetDIMM` significantly improves the communication latency by eliminating costly PCIe transactions and leveraging the physical proximity of NIC and DRAM for data movement. Based on our evaluation results, across various packet sizes, `NetDIMM` on average reduces the one-way network latency between two servers by 49.9% and 25.9% compared with servers employing PCIe and integrated NICs, respectively. We also replay traces from three Facebook production clusters and observe 25.3∼40.6% average per packet latency reduction when replacing PCIe NICs with `NetDIMM` across different clusters. Lastly, we show that depending on the network application running on a server, co-running applications that use the same memory channel as `NetDIMM` can experience up to 30.9% lower memory access latency while in worst case experiencing 15.4% higher memory access latency compared with running the workloads on a system with an integrated NIC.

## 5.2 Background

### 5.2.1 Network Architecture

Despite a large body of research, the innovations in Internet network architecture have been limited to incremental updates and its architecture has remained more or less the same since the creation of the Internet. The main reason for this resistance to changes is the multi-provider nature of the network ecosystem that any change in the existing architecture needs a consensus among several stakeholders. Moreover, this network architecture has been reliably working for several decades and radical changes in it have become increasingly difficult.

Figure 5.2 shows the overall network hardware architecture of a server. A NIC is connected to a processor over a PCIe link. Modern NICs use the Data Direct I/O (DDIO) technology [211, 212] to reduce memory bandwidth utilization when sending and receiving network packets. That is, when a packet is received at a NIC, a DMA engine transfers the packet to a buffer inside processor's last level cache (LLC) instead of moving it all the way to DRAM. When transmitting a packet with a DDIO enabled NIC, the packet buffer is allocated in LLC and DMA engine reads the packet from LLC. However, the DDIO share is usually 10% of the LLC capacity [212] (*i.e.,* a few megabytes) and often this space is exhausted by a NIC at high RX/TX rates. Moreover, sharing the DDIO space between several network functions can result in a phenomenon known as DMA leakage [213]. The DDIO can cause cache pollution for other applications if there is no upper limit for its LLC share [214].

An Ethernet NIC employs a circular ring buffer (*i.e.,* descriptor ring) inside the main memory to let the processor and NIC produce and consume packets

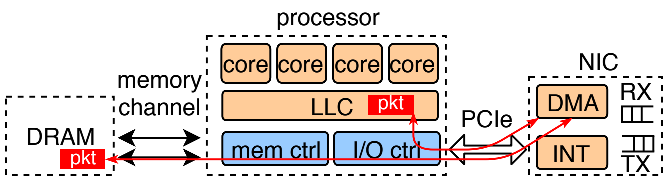


**Figure 5.2:** Server network architecture.

at different rates. Because interrupt handling and interrupt moderation can delay the packet processing for several microseconds, ultra-low latency networks are usually deployed in (adaptive) polling mode [168, 215]. Here we explain NIC, CPU, and memory interactions when transmitting (TX) and receiving (RX) a packet using an Ethernet NIC with a polling driver. Before any transmission or reception (*i.e.,* during the system boot up), the NIC driver allocates RX and TX descriptor rings, initializes them and sends their information to the NIC. (T1 - @Driver) The transmit function of the driver is called and the driver checks the status of the NIC. (T2 - @Driver) The driver sets up a DMA transfer by writing into a NIC configuration register. (T3 - @NIC) The DMA device fetches the next available TX descriptor from DRAM (or LLC if the DDIO is enabled) and then performs another DMA to transfer the packet to the NIC. (T4 - @NIC) The packet is transmitted over the Ethernet link and the TX ring tail pointer is updated. (R0 - @NIC) The packet is received at the destination NIC. (R1 - @NIC) The next available RX descriptor is fetched from DRAM or LLC (R2 - @NIC) The packet is DMAed to the RX descriptor buffer. (R3 - @NIC) The RX descriptor ring information is updated. (R4 - @Driver) The polling driver is notified of a new packet reception. (R5 - @Driver) A new socket buffer (*i.e.,* SKB) is created and initialized with the data in the RX ring buffer. The Ethernet header is removed, and the rest of the packet is sent to an upper network layer.

## 5.2.2 Asynchronous Memory Access

In this subsection, we discuss nonvolatile dual-inline memory module (NVDIMM) protocols. We specifically talk about NVDIMM-P and how DDR5 specification manages to interact with such memory technology. The NVDIMM technology offers persistence and high memory capacity while using the memory channel, that is the fastest interconnect in the system, to interface with the processor. Based on JEDEC standard, there are three types of NVDIMMs:

(1) NVDIMM-N consists of byte-addressable DRAM modules and a backup NAND flash device. In NVDIMM-N, the host DDR memory controller only addresses the DRAM part of the NVDIMM-N. NVDIMM-N has the access time of a regular DDR DIMM from the host perspective.

(2) NVDIMM-F directly exposes the NAND flash storage to the processor and removes the DRAM devices. NVDIMM-F cannot be accessed with regular DDR timing and the memory channel has to slow down to meet the NVDIMM-F timing.

(3) NVDIMM-P uses a novel memory channel protocol that allows asynchronous, out-of-order completion of the memory accesses to have the best features of both NVDIMM-N and NVDIMM-F. NVDIMM-P exposes both DRAM and NAND flash to the host processor address space. Because the NAND flash (or any other persistence memory technology such as 3D-XPoint [123]) has different access timing compared with DRAM, a conventional DDR protocol cannot be used to access the persistent memory region. DDR5 specification is designed to comprehend the heterogeneous media type and support a mixture of convectional DIMM and NVDIMM-P. To facilitate NVDIMM-P accesses, DDR5 specification supports asynchronous memory transactions [216]. Figure 5.3(b) compares the timing of a cacheline read from DRAM and NVDIMM-P in the DDR5 standard. As shown, to access a cacheline from NVDIMM-P, depending on the location of the data (if it is cached in the buffer device of NVDIMM-P or not), a read access has non-deterministic latency. A read request to NVDIMM-P starts with a read request (*i.e.,* XRD in Fig. 5.3(b)) command that includes the full address of the requested data and a request ID. Unlike DRAM operations, each NVDIMM-P request has an ID to facilitate out-of-order access completion. When the XRD command is received at NVDIMM-P, the media data read command is immediately issued. Once the data is ready in the media, a ready command (*i.e.,* RDY) is issued on the response pins (*i.e.,* RSP) with the ID of the original request. The memory controller then issues a send (*i.e.,* SEND) command to read the data. The data appended with the request ID is available on the data bus (*i.e.,* DQ) after a specific amount of time.

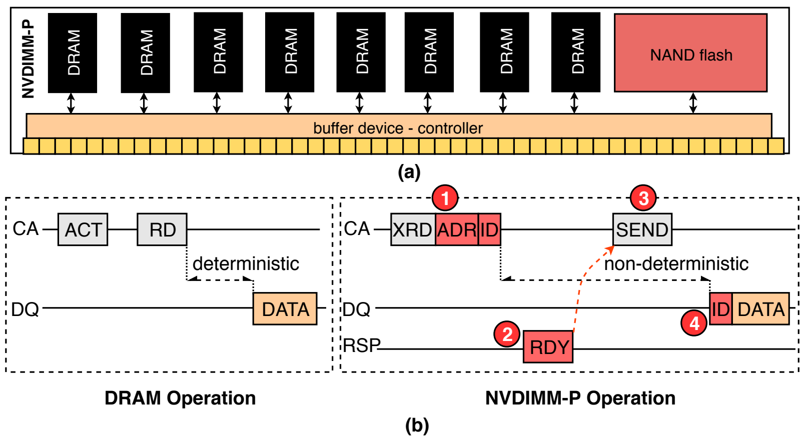


**Figure 5.3:** (a) NVDIMM-P architecture, (b) asynchronous memory access for NVDIMM-P.

### 5.2.3 Linux Memory Management

**Memory Address Mapping.** Different systems use different physical memory address mapping and decode different bits in the physical address to calculate the channel, rank, bank, row and column of the address location in the DRAM. If there are DIMMs installed on multiple memory channels, then the memory mapping can operate at three different modes as follows: single channel, multi-channel, and flex channel modes. In single channel mode, the memory channel bits are mapped to the most significant bits of the physical address and sequential addresses are mapped to one memory channel. In multi-channel mode, sequential memory addresses are interleaved between multiple memory channels. Flex mode provides a flexible memory mapping configuration where a part of address space can work in multi-channel mode and the rest in single channel mode. Flex mode is especially useful in asymmetric memory configuration where different DIMM types (*e.g.,* DDR5 or NVDIMM-P) are installed on memory channels [217].
**Linux Kernel Memory Allocation.** Due to hardware limitations, different parts of physical memory should be treated differently by the Linux kernel. Linux groups physical memory locations into four primary memory zones: ZONE_DMA: contains pages that can be used for DMA; ZONE_DMA32: contains pages that can be used for DMA by 32-bit devices; ZONE_HIGHMEM: contains "high memory" [218] pages that cannot be mapped into the kernel address space in 32-bit machines; ZONE_NORMAL: contains regularly mapped pages in the system.

`kmalloc()` is used to allocate memory in kernel, similar to `malloc()` in userspace. `kmalloc()` can allocate memory from a specific memory zone based on the input arguments. There are also several APIs for allocating memory in page granularity in Linux. These APIs are especially used in the network stack for allocating the paged area of the network socket buffers [219]. The core function for page allocation is `__alloc_pages()`. There are several wrapper APIs to allocate pages from a specified NUMA node and/or memory zone.

## 5.3 Motivation

As we discussed in Sec. 5.2.1, to send a packet over a conventional NIC, several PCIe and memory channel transactions need to take place. More specifically, in a client-server application, 16 one-way PCIe transactions are needed for completing one request-response transfer. Several research studies have proposed new NIC and DMA architectures to reduce the number of PCIe transactions when sending and receiving network packets, especially for small packets [202, 203, 38]. Although such architectures improve the network latency, they still require several PCIe round-trips to send and receive packets to and from the NIC, respectively.

CPU and NIC integration is a promising approach for solving the overheads mentioned above. Figure 5.4 shows the one-way latency of sending packets of different size from one node to another through a 40Gb Ethernet link. For more information on our evaluation methodology please refer to Sec. 5.5.1. We evaluate four different NIC configurations: discrete NIC (`dNIC`), which represents a conventional PCIe Gen3×8 NIC (*i.e.,* Fig. 5.1(left)); `dNIC` with zero copy transmission and reception (`dNIC.zcpy`); a NIC integrated into CPU chip (`iNIC`) (*i.e.,* Fig. 5.1(middle)); and `iNIC` with zero copy transmission and reception (`iNIC.zcpy`). The figure also shows PCIe contribution to the overall packet transmission and reception (`pcie.overh` in Fig. 5.4). As shown, `iNIC` improves the network latency by 21.3∼38.6% compared with `dNIC`. The latency improvement is more signified for smaller packets and mainly comes from faster accesses to the I/O registers. Figure 5.4 clearly shows the benefit of removing PCIe link between the CPU and NIC for low-latency networking.

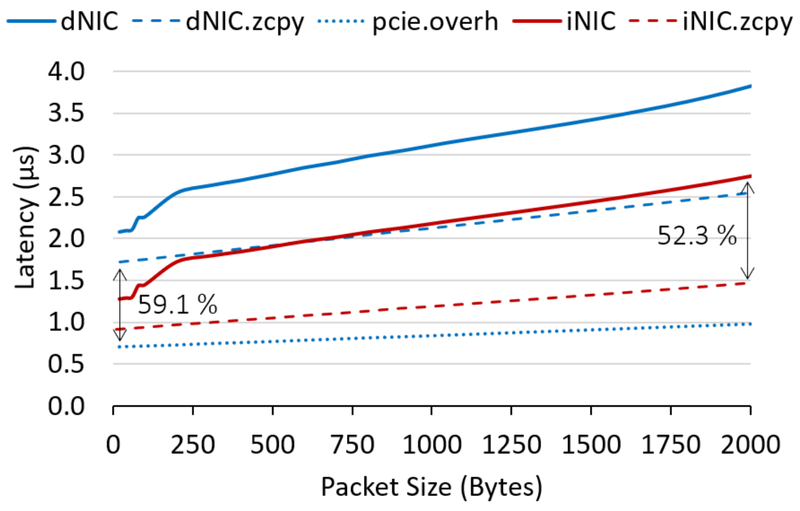


**Figure 5.4:** One-way latency comparison of different NIC configurations for packets of various sizes: discrete NIC (`dNIC`), discrete NIC with zero copy (`dNIC.zcopy`), integrated NIC (`iNIC`), and integrated NIC with zero copy (`iNIC.zcpy`). `pcie.overh` shows the overhead of PCIe interconnect for discrete NIC configurations.

We enable zero copying by letting NIC to access application buffers as DMA buffers. Zero copy improves `iNIC` network latency by 28.8% and 52.3% for 10Byte and 2000Byte packets, respectively. As expected, memory copy overhead increases with packet size and larger packets benefit more from zero copy networking. On the other hand, the PCIe overhead is more for smaller packets. For `dNIC.zcpy`, 40.9% and 34.3% of the overall network latency is spent in PCIe interconnect when transferring 10Byte and 2000Bytes packets, respectively.

Although `iNIC.zcpy` seems to be an ideal ultra-low latency network architecture, it has several limitations: (**L1**) Zero copy networking can introduce security breaches [210]. Also pinning application pages to the memory can cause main memory exhaustion and the overhead of virtual memory operations and buffer management can nullify the gains of zero copy networking [9]. (**L2**) Integrating a full-blown NIC into CPU significantly increases the area and power of the processor. It is specifically challenging as often NIC and CPU are manufactured by different vendors. (**L3**) Most importantly, `iNIC` can pollute on-chip resources, such as LLC, at high network rates or cause memory interference for co-running applications. Furthermore, storing the payload of received packets on the processor chip

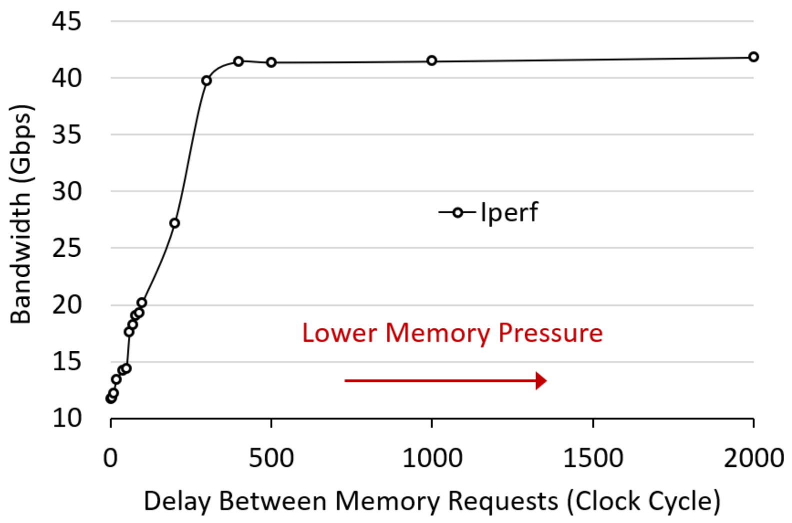


**Figure 5.5:** `iperf` bandwidth at different memory pressure levels.

is waste of precious on-chip resources for network functions that only require packet header to be processed by the CPU [220]. Note that (L3) is not specific to `iNIC` and `dNIC` also has the same problem.

To illustrate the memory and cache interference caused by network packets, we study the sensitivity of network bandwidth to the cache and memory interference. Figure 5.5 depicts the sensitivity of network bandwidth to the pressure on the memory system. In this experiment, we use two machines, each equipped with a Xeon E5-2660 processor, three DDR4 memory channels, and an Intel 40Gbps XL710-QDA1 NIC. We use Intel Memory Latency Checker (MLC) [221] tool to inject dummy memory requests to the memory sub-system at different rates. We set the ratio of memory read to write requests to 1. In Fig. 5.5, the X-axis shows the delay between injected memory requests (higher values lower the interference at the memory sub-system) and Y-axis shows the achieved `iperf` [161] TCP bandwidth at different memory interference levels. `iperf` bandwidth significantly drops when the memory pressure from MLC increases. For example, at the maximum memory pressure, which corresponds to 15.1GBps per memory channel, `iperf` only delivers ∼27.9% of the achieved bandwidth without any interference from MLC. This experiment shows how sensitive network bandwidth is to the interference at the memory sub-system. Moreover, Fig. 5.5 can be interpreted from another angle: the network traffic can cause severe interference at the memory sub-system. However, here we could not show that because TCP flows from `iperf` regulate the transmission rate based on the processing capability of the receiver node. Therefore, before we see any major degradation on the local application performance, the `iperf` bandwidth decreases.

Figure 5.4 and Fig. 5.5 illustrate the inefficiencies in the network architecture of current servers. Ideally, we want to completely remove the PCIe transactions and exchange data between the processor and NIC over an interconnect with lower latency without jeopardizing the network bandwidth. Furthermore, to reduce the memory interference, we want to decrease the host memory sub-system utilization when sending and receiving packets to and from NIC; which involves preventing a NIC from injecting all the received traffic to LLC. Instead, we want a mechanism which collectively brings different bytes of a received packet to the processor on the application's demand. PCIe is a standard and well-developed interconnection technology that has been around for three decades. One key requirement for a replacement is that it should be a standard and well-established interconnection technology. Introducing a new and specialized interconnect is costly and error prone. Also, the new interconnect should seamlessly work with memory channel and processor cache hierarchy to facilitate quick data delivery to the CPU.

Memory channel has the lowest latency among off-chip interconnects in a modern server. Besides low latency, memory channel provides high bandwidth. For example, a DDR4 channel provides 12.8GBps (*i.e.,* 102.4Gbps) bandwidth. The latency of transferring a 4KB page over a DDR4 channel and a $\times 8$ PCIe link are $\sim 200ns$ and $\sim 2\mu$s, respectively. More importantly, the memory channel is a standard and well maintained interconnect that can be find on the motherboard of any server. We leverage these unique features of the memory channel and propose a near-memory network interface card architecture by placing a NIC into the buffer device of a DIMM. This design solves all the limitations of `dNIC` and `iNIC`: (1) eliminating the PCIe overhead by utilizing memory channel and internal DIMM interconnects for packet transmission and reception; (2) supporting in-memory buffer cloning to copy packets from application to DMA buffers and vice versa; (3) decoupling header and payload of packets to reduce LLC pollution and (4) using a separate memory channel to access network buffers in the DRAM to reduce the host memory channel interference.

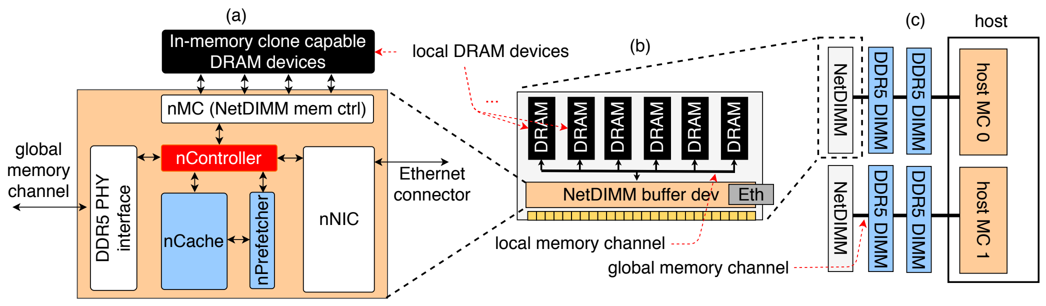


**Figure 5.6:** NetDIMM architecture.

## 5.4 Network-Attached DIMM

Motivated by the explanation in Sec. 5.3, we propose NetDIMM, a low-latency, near-memory network architecture. Building atop the NVDIMM-P architecture (Sec. 5.2.2) and based on the near-memory processing concept, NetDIMM improves the data transfer latency between the processor, memory, and NIC. In this section, we explain the hardware and software components of NetDIMM in detail.

### 5.4.1 NetDIMM Hardware Architecture

Inspired by the asynchronous, out of order memory access support of DDR5 specification (Sec. 5.2.2), we architect a NIC that is placed on the buffer device of a DIMM. Figure 5.6 overviews the overall architecture of NetDIMM. Figure 5.6(c) shows a system with two memory channels where each memory channel is occupied with three DIMMs in total. Out of these three DIMMs, there are two conventional DDR5 DIMMs, and one NetDIMM. Note that the figure only shows an example system and there is no requirement for the number of NetDIMMs on a memory channel. For example, a system can have one NetDIMM installed on one of the DDR5 slots. The DDR5 support of asynchronous memory request completion allows mixing DRAM and NetDIMM on a same memory channel [222]. As shown in Fig. 5.6(b), the organization of NetDIMM is similar to the organization of an NVDIMM-P depicted in Fig. 5.3(a).

Figure 5.6(a) shows the internal architecture of NetDIMM buffer device. It consists of the following main components: (nNIC) An integrated network interface card; (nMC) one (or several) memory controller(s) to access

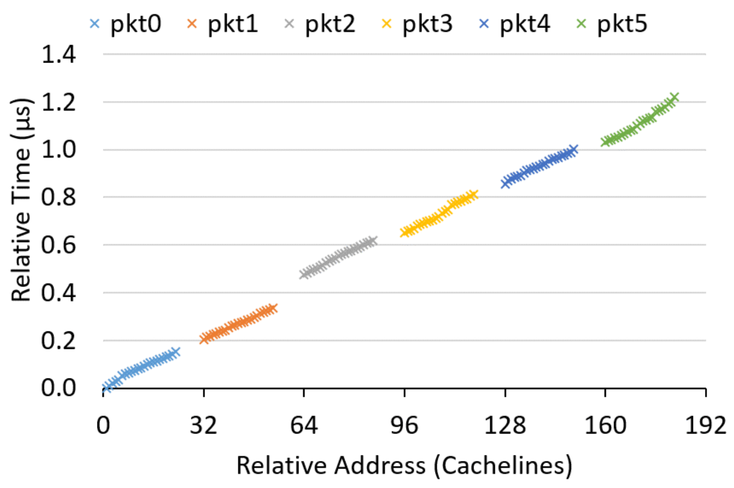


**Figure 5.7:** Spatial and temporal locality of NIC memory accesses from host processor perspective.

the `NetDIMM` local DRAM modules; (nController) logic that extends the NVDIMM-P controller with `NetDIMM` routing and management logic; (DDR5 PHY interface) DDR5 physical interface and protocol engine. The DDR5 physical interface contains a protocol engine that repeats DRAM CA, DQ, and RSP signals similar to a typical NVDIMM-P device; (nCache) a dual-port SRAM buffer for caching RX data resided in the local DRAM modules; (nPrefetcher) a next-line prefetcher for pre-loading RX packets to nCache from the local DRAM modules; (RowClone enabled DRAM) DRAM devices that support in-memory data copying.

We expose the local DRAM capacity of `NetDIMM` to the host memory address space, therefore, the local `NetDIMM` memory is managed by the host operating system. This is similar to the unified address space of NVDIMM-P. We explain `NetDIMM` memory management in Sec. 5.4.2. Because both nNIC and PHY can independently access the local DRAM modules through nMC, we need arbitration between the memory accesses from nNIC and PHY. nController does this arbitration by giving priority to the nNIC accesses. Because of the following reasons, the access time to the local DRAM from the host MC is non-deterministic: (R1) the host MC does not know the state of the `NetDIMM` local DRAM modules; (R2) nMC is shared between nNIC and PHY. Thus, the access time of the local DRAM modules depends on the current state of the local DRAM modules, the current nNIC traffic, and the current requests from PHY.

We make a key observation that the memory access pattern between the host processor and NIC is very regular and has spatial and temporal locality. Figure 5.7 plots the relative address and relative arrival time of memory requests, generated by the DMA engine of a 40GbE NIC, when receiving six 1514 Byte packets. For detailed experimental setup please refer to Sec. 5.5.1. As illustrated, each packet arrival generates a burst of memory requests to DMA buffers. Each burst consists of 24 cachelines[1] (24 * 64 = 1536 Bytes) that arrive at the host memory controller in a short time interval, which for example is 143ns for the third packet. nCache and nPrefetcher components exploit the unique characteristics of this memory access pattern to improve the host MC access latency to the `NetDIMM` address space.

Once a packet is received at nNIC from the outside, nNIC notifies nController. nController implements the same functionality of a DMA engine in a conventional NIC. Upon receiving the notification from nNIC, nController reads the next available descriptor buffer from nMC and depletes the RX buffer of nNIC to the descriptor ring resided in the `NetDIMM` local DRAM modules. In Sec. 5.4.2 we explain how the descriptor ring is allocated on `NetDIMM`. While transferring the RX packets to the `NetDIMM` local DRAM space, the nController writes the first cacheline of each received packet to nCache. The rationale for only caching the first cacheline of received packets is that for all transport protocols, the header size is less than 64 Bytes (*i.e.,* one cacheline) and only the header of a received packet is needed for processing the packet in the network software stack[2]. Moreover, as explained in Sec. 5.3, some network functions, such as forwarding and firewall, do not need the packet payload as the application makes forwarding decisions only based on the header information. The maximum header size of a TCP/IP packet is 52Bytes [154], so caching the first 64Bytes of a received packet includes all the headers. The rest of the packet is only accessed when copying it to a userspace buffer. Storing an entire received packet in nCache is not efficient as the reuse distance of the payload of a received packet is much longer than its header.

Assuming a maximum transmission unit (MTU) size of 1500 Bytes, each Ethernet packet can carry 1∼24 cachelines. When the payload of a received packet is accessed (*e.g.,* to be copied to an application buffer), a stream

[1]We assume that the cacheline size is 64Bytes

[2]Assuming that nNIC has checksum offloading support

of consecutive read requests is received to `NetDIMM` PHY, similar to the access pattern shown in Fig. 5.7. This access pattern is easy to predict by a simple next-line prefetcher. We add this prefetcher, shown as nPrefetcher in Fig. 5.6(a), to `NetDIMM`. nPrefetcher prefetches the next $n$ cachelines and stores them in nCache. Therefore, even if `NetDIMM` does not cache the payload of RX packets in nCache, in the worst case, reading an entire RX packet may only experience one nCache miss. We disable nPrefetcher for the first cacheline of RX packets which contains the header. This is because we do not want to pollute nCache when only the header of a received packet is accessed by the host processor. We add a one-bit flag for each cacheline of nCache, that is set when the first cacheline of a newly arrived packet is stored in nCache. nPrefetcher checks this flag and prefetches next $n$ cachelines if the flag is not set. nCache resets the flag after the first access to each cacheline.

When a read request is received from the global memory channel, nController checks if the requested data is cached in nCache. If it is a hit, the data is read from nCache and immediately sent to the host MC. Otherwise, nController creates a read request and sends it to nMC. Once the data is read from the local DRAM through nMC, it will be sent to the host using the asynchronous protocol explained in Sec. 5.2.2. When a write request is received from the global memory channel, nController constructs a memory write request and send it to nMC. The write requests do not use nCache as they are immediately queued in the nMC write queue upon arrival.

nCache is an inclusive, set associative cache structure. nCache is more like a large data buffer and its data is removed from it once it is accessed. This is because once the RX packet is read from `NetDIMM`, it is going to be stored in a host processor cache or in another location in the main memory. In either case, that memory address is unlikely to be accessed in a near future. Therefore, there is no value in keeping that data in nCache. We use random replacement policy to make space in an nCache set if all the blocks in the set are occupied. Note that all cachelines in nCache are clean and there is no need for writing a victim cacheline back to nMC. To ensure the coherency of nCache with local DRAM data, nController snoops the addresses of write requests received from PHY or nNIC and invalidates the matching cachelines in nCache.

Conventionally, copying one memory location to another involves a

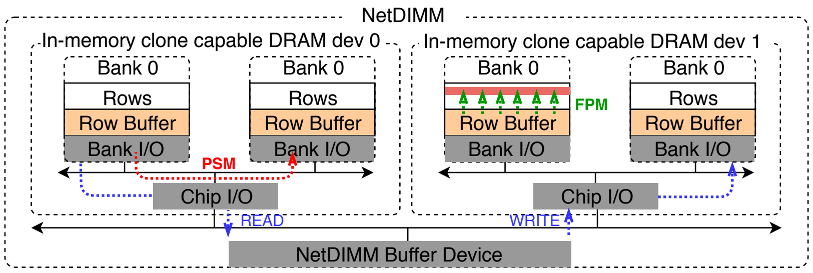


**Figure 5.8:** In-memory buffer cloning acceleration.

processor to read data over its memory channels into its cache hierarchy and then write it back through the memory channels to the destination memory location. This makes memory copying an expensive operation. For example, copying a 4KB page over a DDR3 memory channel takes ~1$\mu$s [223]. Because of the limitation of zero-copy drivers (discussed in Sec. 5.3), we envision an in-memory data copy acceleration mechanism to swiftly clone application buffers to DMA buffers and vise versa on `NetDIMM`. To this extent, we utilize an extended implementation of RowClone [223] mechanism. RowClone is an in-memory bulk data copying mechanism that utilizes DRAM internal architecture to accelerate memory-to-memory copying on a single DIMM. Figure 5.8 illustrates a high-level overview of in-memory clone-capable DRAM devices. Depending on the location of the source and destination addresses, there are three modes for cloning a page: Fast parallel mode (FPM): source and destination pages share a bank sub-array. In this case buffer cloning can be done by two back to back activation of the source and destination pages. FPM mode is highlighted with green arrows in Fig. 5.8; Pipeline serial mode (PSM): source and destination pages are on different banks but on a same DRAM device. In this case cloning happens by pipelining cacheline copy operations over the internal bus of DRAM chips. PSM mode is highlighted with the red arrow in Fig. 5.8; General cloning mode (GCM): otherwise, `NetDIMM` reads source to the `NetDIMM` buffer device and writes them back in pipeline mode to the destination address (highlighted by blue arrows in Fig. 5.8). GCM is similar to the operation of a conventional DMA engine near the memory chips. FPM is the fastest while GCM is the slowest and most general mechanism. That being said, it is important to intelligently allocate source and destination pages to a same sub-array within a DRAM device in order to extract the maximum benefit from the in-memory page cloning. In Sec. 5.4.2 we explain how `NetDIMM` implements an intelligent memory allocation scheme to efficiently move data from DMA buffers to application buffers.

### 5.4.2 `NetDIMM` Software Architecture

In this subsection, we explain required software stack changes to enable `NetDIMM`. Overall, we try to have the minimum amount of changes possible in the network software stack and Linux kernel. The changes in the software stack includes implementation of a new Linux memory allocation API, changing the physical memory address mapping, and implementation of a `NetDIMM` driver. The TCP/IP layers remain unchanged except for the API for SKB allocation. Note that we developed a userspace `NetDIMM` driver for our evaluations. However, to show the feasibility and generality of our implementation, we also developed a Linux kernel `NetDIMM` driver that runs the full Linux kernel software stack and unmodified userspace applications. We use our Linux kernel driver for explanation here.

**Handling `NetDIMM` local memory region.** Before we talk about `NetDIMM` driver, we first need to discuss how we use the local DRAM modules on `NetDIMM`. To leverage the operating systems memory management functionality, keep the amount of changes in the software stack at minimum, and make `NetDIMM` application-transparent, *we expose the local memory capacity of `NetDIMM` to the host processor as if it is part of the host physical memory address space.* The local memory capacity of a `NetDIMM` can be seen as a memory node in a NUMA system, and despite different access timing, `NetDIMM`'s memory space is part of the host (global) address space. We reveal this heterogeneity in the memory system to Linux by creating a new memory zone called NET$i$ where $i$ is the `NetDIMM` number in the system. Note that a system can have multiple `NetDIMM`s installed on memory channels and each need a different memory zone. Defining a memory zone in Linux is not expensive and new memory zones have been added to Linux when necessary [224].

In addition to defining new memory zones, it is also important to intelligently allocate DMA and application buffers on a same bank and sub-array to extract the maximum performance out of `NetDIMM`'s in-memory buffer cloning capability (*cf.* Fig. 5.8). To achieve this, we need to expose

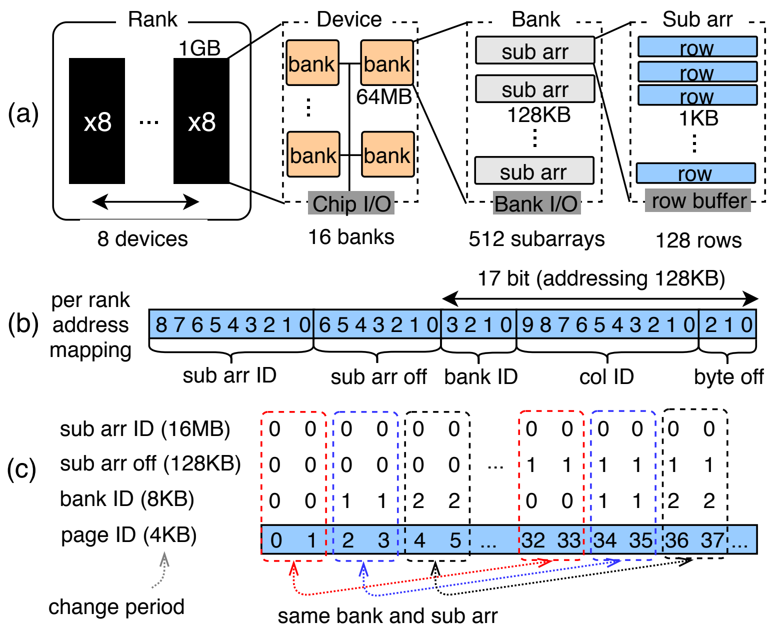


**Figure 5.9:** (a) Configuration of a memory rank in a NetDIMM; (b) physical memory address mapping; (c) illustration of the physical location of pages.

the internal memory organization of NetDIMM to the memory scheduler. Figure 5.9(a) shows our assumptions about the size and organization of a memory rank in NetDIMM, which is based on a Micron MT40A512M16 DRAM device [225]. Each rank consists of eight ×8 DRAM devices, each device consists of 16 banks, each bank is divided into 512 sub-arrays, and each sub-array consists of 128 rows. The capacity of each rank, device, bank, sub-array, and row is 8GB, 64MB, 128KB, and 1KB, respectively. Based on this organization, the physical memory address mapping for NetDIMM looks like Fig. 5.9(b). Assuming a page size of 4KB, Fig. 5.9(c) illustrates the geometric location of consecutive pages stored in a memory rank. As shown, the pages that are physically stored on a same bank and sub-array are spaced every 128KB (or 32 pages). Thus, it is easy to check if two pages are on a same sub-array and bank. We implement `__alloc_netdimm_pages(zone, hint)` that allocates a page on NetDIMM `zone` and the same sub-array as `hint` address. If `hint` is set to -1, then the API only considers the `zone` requirement. Note that this is a best effort API and it is possible that the allocated pages are not on the same sub-array as the `hint` address.

Another complexity in handling the local memory area of NetDIMM is the memory channel interleaving of physical addresses in a systems with

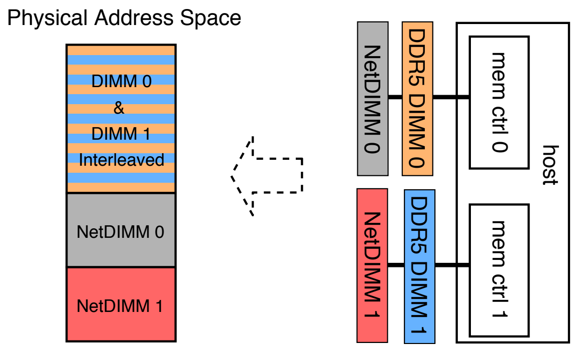


**Figure 5.10:** The memory address space and channel interleaving mode for a mixture of DDR5 DIMMs and `NetDIMM`s.

multiple memory channels. Memory channel interleaving increases the memory throughput by parallelizing memory accesses over several memory channels. However, we need to disable the memory channel interleaving for the `NetDIMM` address space because the global memory channels are not visible to nNIC (Fig. 5.6). Therefore, the `NetDIMM` address space should be exposed to the host in single channel mode so the host processor sees the `NetDIMM` physical address as a continuous memory chunk. We leverage the Flex channel interleaving mode (*cf.* Sec. 5.2.3) to divide the physical address space into two parts. One part contains all conventional DDR DIMMs that operate in multi-channel mode and another part contains `NetDIMM`s address space operating in single channel mode. Figure 5.10 depicts the unified address space of the conventional DIMMs and `NetDIMM`s and their memory channel interleaving modes.

`NetDIMM` **Driver.** We use Intel e1000 GbE driver as a base to develop `NetDIMM` driver. Because `NetDIMM` is not a PCIe device, `ioremap()` API is used to create a configuration space for `NetDIMM` similar to the configuration space of a conventional PCIe NIC. Using this techniques, we can configure all the features of a full-blown NIC without the need for writing a new driver from scratch.

When a NIC interface is initialized, it creates transmit (TX) and receive (RX) descriptor ring buffers and initializes their buffer pointers by allocating DMA buffers. Moderns NICs support scatter-gather DMA operation, so a DMA buffer can span over multiple pages that are not physically contiguous. `NetDIMM` requires that the physical location of descriptor rings and their corresponding DMA buffers to be on the memory zone of the corresponding

NetDIMM. To benefit from in-memory cloning acceleration, applications are also required to allocate their network data buffers on NetDIMM memory zone. For both TX and RX rings, we use __alloc_netdimm_pages(zone$i$, -1) to allocate descriptor ring data structures for NetDIMM$i$. For RX and TX DMA buffers, we allocate them on the fly based on the location of application buffers. However, calling __alloc_netdimm_pages for each packet can deteriorate the network latency and bandwidth. As shown on Fig. 5.9(a) each NetDIMM rank has 512 * 16 = 8K distinct sub-arrays. To accelerate the on-demand memory allocation, NetDIMM pre-allocates two pages from each distinct sub-array and stores them in a hash table called allocCache. Considering that NetDIMM has two memory ranks, each NetDIMM pre allocates 32K pages (*i.e.,* 128MB) for on-demand DMA buffer allocation. This corresponds to 0.8% of capacity overhead for a 16GB NetDIMM. allocCache immediately returns a page allocated on a specific sub-array. NetDIMM driver refills allocCache concurrently in the background, thus, the on-demand allocation of DMA buffers are not in the critical path of packet RX and TX.

One complication here is that an application should have knowledge about the physical layer to know which NetDIMM is serving its packet streams. To resolve this, we add a flag to the SKB header (or any other type of network data structure used for networking) called COPY_NEEDED. We allocate the SKBs that belong to the connection establishment on the regular kernel address space and set the COPY_NEEDED flag in the SKB header. At the transmit function of NetDIMM driver, if COPY_NEEDED flag is set, the driver first copies the SKB data to an allocated TX DMA buffer on the corresponding NetDIMM and then initiates the packet transmission. Each SKB has a pointer to the socket that the packet is associated with. We add a new field to "struct sock" called "struct zone_struct skb_zone" and set it to NET$i$ in the NetDIMM driver. Therefore, after the first packet transmission, each connection has enough information to allocate the SKB and paged buffers of the TX packets on a corresponding NetDIMM memory zone. Note that COPY_-NEEDED flag is also used as a fallback mechanism in case the memory space on a NET$i$ zone is exhausted and the SKB and TX buffers are allocated on different memory zones. This is a rare event and does not happen frequently.

When receiving a packet from NetDIMM, similar to a PCIe NIC, once nNIC finished moving a received packet to a DMA buffer, it needs to notify the

**Algorithm 5:** Packet TX and RX handling at `NetDIMM` driver.

```
TX:
 $txDesc[next].dma = allocCache[txSKB.data]$ // DMA buffer allocation
 if $txSKB.COPY\_NEEDED$ then
     copy $txDesc[next].dma \leftarrow txSKB.data$    // slow path
     set skb_zone to NETi
     flush $txDesc[next].dma$ to memory
 else
     flush $txSKB.data$ to memory                  // fast path
 set $txDesc[next]$ size and flags         // total size is 64 bits
 flush $txDesc[next]$ size and flags    // kick off transmission
RX:
 invalidate $rxDesc[next]$      // to fetch fresh data from NetDIMM
 $rxSKB.data = allocCache[rxDesc[next].dma]$  //RX buffer allocation
 $netdimmClone(rxSKB.data, rxDesc[next].dma, rxDesc[next].size)$ //
   in-memory buffer cloning
 send rxSKB to upper network layers for processing
Polling Agent:
 clean TX buffers after a successful transmission
 if newly arrived packet then
     call RX
```

host processor. To notify the processor about newly received packets or packet transmission completions, a NIC typically uses an interrupt signal or a polling agent. The interrupt approach is mostly used for high bandwidth network connections where the network latency is not critical. On the other hand, a polling mechanism is mainly used by userspace network stacks and low-latency networks to prevent interrupt processing and context switching overheads (*cf.* Sec. 5.2.1).

`NetDIMM` driver implements an efficient polling agent using a high-resolution kernel timer. Note that polling `NetDIMM` is more efficient than polling a PCIe NIC as accessing I/O registers on a `NetDIMM` is much faster than a PCIe NIC. After the polling driver detects a packet arrival, it calls the RX routine of the driver as shown in Alg. 5. `NetDIMM` uses memory flush and invalidate instructions to enforce coherency between processor caches and `NetDIMM` local memory. The `netdimmClone(dst, src, size)` function shown in Alg. 5 is the API for in-memory buffer cloning. It writes `dst`, `src`, and `size` values to a set of `NetDIMM` registers and `NetDIMM` clones `src` to `dst` buffer inside the memory.

**Table 5.1:** System configuration.

| **Parameters** | **Values** |
|---|---|
| Cores (# cores, freq): | (8, 3.4GHz) |
| Superscalar | 3 ways |
| ROB/IQ/LQ/SQ entries | 40/32/16/16 |
| Int & FP physical registers | 128 & 192 |
| Branch predictor/BTB entries | BiMode/2048 |
| Caches (size, assoc): I/D/L2 | 32KB,2/64KB,2/2MB,16ways |
| L1I/L1D/L2 latency,MSHRs | 1/2/12 cycles, 2/6/16 MSHRs |
| DRAM | DDR4-2400MHz/16GB/2 channels |
| Network/Switch latency/#`NetDIMM` | 40GbE/100ns/1 |
| PCIe performance | ×8 PCIe 4 [8] |

### 5.4.3 Physical Feasibility of `NetDIMM`

One question that we still need to answer is how feasible it is to integrate a full-blown NIC into the buffer device of a DIMM in terms of power and thermal specifications. There are products [140, 156, **?**, 226, 148] and academic research proposals [178, 21] that add processing power to the buffer device of conventional DIMMs. Centaur DIMM (CDIMM) [140] is a buffered DIMM, designed by IBM to scale the memory capacity of POWER processors. CDIMM comprises of up to 80 DDR DRAM devices and a Centaur device that consists of a 16MB L4 cache, four memory controllers, and other controlling logic. The TDP of an IBM Centaur buffer device is 20W in 22$nm$ technology. On the other hand, a modern XXV710 Intel PCIe Ethernet controller incorporating 2×40Gbps ports has a TDP of 6.5W [227]. Therefore, considering the specification of the current DIMM products, it is feasible to integrate a NIC chip into the buffer device of a DIMM. Lastly, we always can connect an external power cable to DIMMs similar to an NVDIMM [148]. Moreover, we can use a similar connector for the network cable in `NetDIMM`.

## 5.5 Evaluation

### 5.5.1 Methodology

We evaluated `NetDIMM` using `gem5` [160] along with analytical models for PCIe interconnect [8, 228] and memory controller [229]. Because the overhead of

Linux kernel software stack fades the latency improvements of `NetDIMM`, we implement a set of bare-metal drivers for our PCIe NIC, integrated NIC and `NetDIMM` models using `gem5` that resemble low-latency userspace drivers and use them for latency evaluations. We configure `gem5` as shown in Table 5.1.

To model `NetDIMM` memory access latency, we instantiate an isolated memory controller that models nMC shown in Fig. 5.6(c). The nMC model is used to access the `NetDIMM` memory zone. A memory request from host to `NetDIMM` is first queued in a host MC. Once it is chosen to be sent to the DRAM, instead of performing a regular memory access, after a `tCMD` delay, the host MC forwards the memory request to a corresponding nMC. The memory request access is completed once the nMC sends a response to the host MC. For the network DMA operations, the memory accesses are directly sent to the nMC model.

For performance evaluations, we use network traces from three Facebook production clusters. Each cluster has different packet size and traffic patterns: first cluster is for `database` applications with their packet size uniformly distributed between 64 Bytes and 1514 Bytes (MTU is set to 1514 Bytes), second cluster is for `webserver` where $\sim$90% of the packet sizes are smaller than 300 Bytes, and third cluster is uses for `hadoop` servers where $\sim$41% of packets are less than 100Bytes and $\sim$52% are 1514 Bytes [230]. The traffic pattern of `database` cluster is mostly inter-cluster and inter-datacenter, `webserver` is mostly inter-cluster but intra-datacenter, and `hadoop` is intra-cluster. The traces are publicly available by Facebook [231]. We randomly pick one node in each cluster and use several dummy nodes to replay the ingress and egress data traffic to and from the node under test. We simulate the Clos network topology of Facebook datacenter using the `dist-gem5` [23] switch model. We assume all the network devices in the datacenter has a bandwidth of 40Gbps. We implement an L3 Forwarding (`L3F`) and a deep packet inspection (`DPI`) network functions as two network functions with extremely different packet processing behaviors to evaluate the impact of `NetDIMM` on the performance of server memory sub-system. We use the Facebook traces to exercises these network functions. `L3F` forwards received packets only based on their header information while the `DPI` processes the entire header and payload to make a forwarding decision.

### 5.5.2 Network Latency and Bandwidth

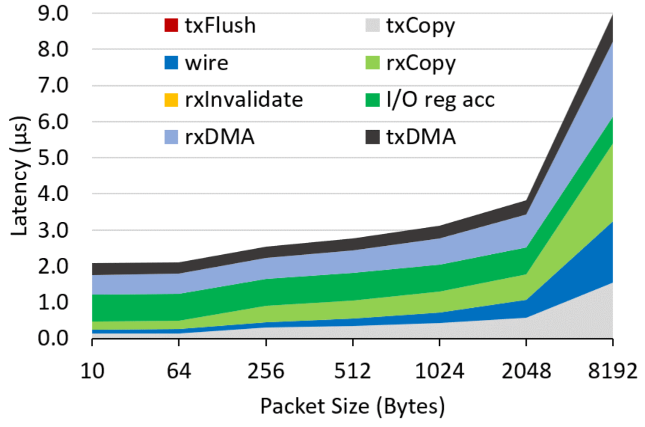


**Figure 5.11:** One-way network latency breakdown for packets of various sizes when using a PCIe NIC. X-axis is not drawn to scale.

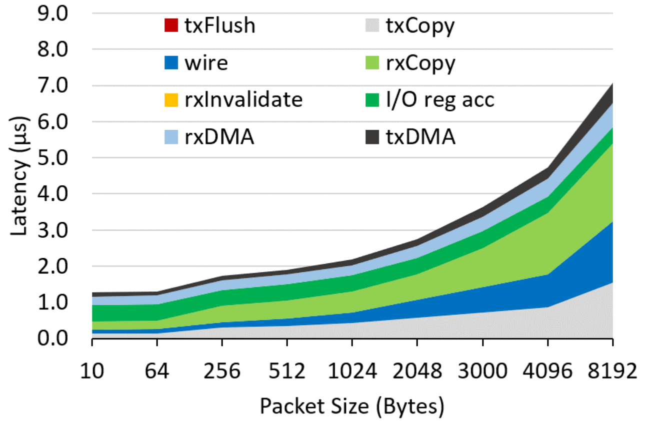


**Figure 5.12:** One-way network latency breakdown for packets of various sizes when using an integrated NIC. X-axis is not drawn to scale.

Figures 5.11-5.13 show one-way network latency breakdown of various sized packets between two nodes directly connected together by PCIe NICs, iNICs, and `NetDIMM`s, respectively. `rxCopy` and `txCopy` respectively show the overhead of memory copy and allocation at RX and TX drivers, `rxDMA` and `txDMA` show the DMA overhead at NIC hardware, `wire` shows the physical layer overhead, and `I/O reg acc` represents the overhead of CPU/NIC register accesses. `txFlush` and `rxInvalidate` represent cache flush and cache invalidate overheads of `NetDIMM` driver, respectively. `NetDIMM` reduces the one-way network latency by 46.1%, 52.3%, and 49.6% for 64B, 256B,

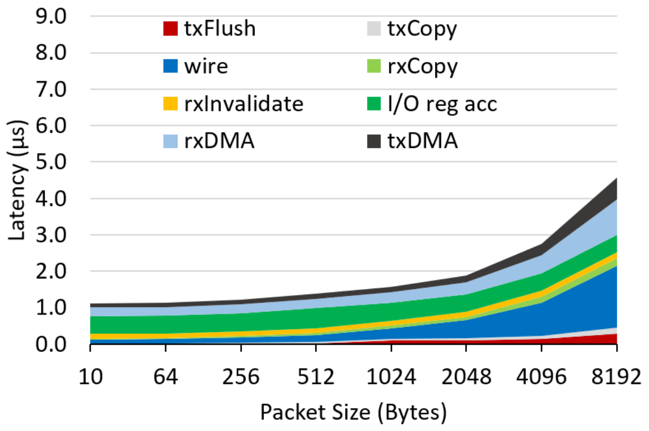


**Figure 5.13:** One-way network latency breakdown for packets of various sizes when using `NetDIMM`. X-axis is not drawn to scale.

and 1024B packets compared with a PCIe NIC, which translates to 0.97$\mu$s, 1.33$\mu$s, and 1.54$\mu$s lower network latency, respectively. As shown in Fig. 5.12 and Fig. 5.13, because of eliminating the PCIe interconnect, `I/O reg acc` is significantly reduced for iNIC and `NetDIMM` compared with that of PCIe NIC. `NetDIMM` adds `txFlush` and `rxInvalidate` overheads to the end-to-end network latency. These two components combined add 9.7∼15.8% overhead to the total network latency. Nonetheless, on average `NetDIMM` delivers 26.0% lower latency than iNIC across different packet size. This shows that the in-memory buffer cloning not only makes up for the overhead of CPU cache operations, but also improves the overall network latency compared with an integrated NIC.

One caveat of `NetDIMM` is that unlike a PCIe NIC, it is located on one memory channel and it cannot utilize multiple memory channels when communicating with the host processor and memory. However, our simulation results show that `NetDIMM` delivers 40Gbps bandwidth just like our PCIe and integrated NIC models. This is not a surprise as the nominal bandwidth of a DDR4 memory channel is 12.8GBps or 102.4Gbps, which is far more than 40Gbps. In fact DDR5 memory channel's projected bandwidth is twice more than that of a DDR4 channel which can sustain any bandwidth of what the current or under development PCIe NICs can deliver.

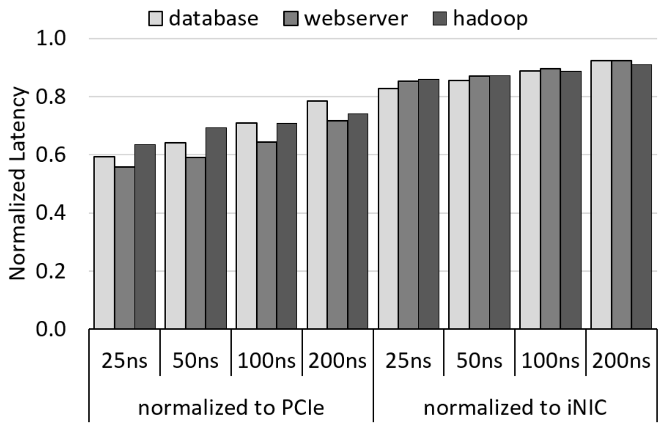


**Figure 5.14:** Per packet normalized network latency for different network switching latency, using servers with `NetDIMM` and replaying Facebook cluster traces.

### 5.5.3 Performance Evaluation

Figure 5.14 shows the average per packet network latency for each cluster with servers using `NetDIMM` normalized to the latency of PCIe NIC and iNIC configurations. We set the latency of network switches inside the simulated Clos network to 25ns, 50ns, 100ns, and 200ns to measure the performance sensitivity of `NetDIMM` to different network configurations. On average, across different clusters, `NetDIMM` improves the end-to-end packet latency of PCIe NIC configuration by 40.6%, 36.0%, 33.1%, and 25.3% when switch latency is 25ns, 50ns, 100ns, and 200ns, respectively. `NetDIMM` improves the average end-to-end packet latency of different clusters employing iNIC by 8.1∼15.3% for different switch configurations. As expected, `NetDIMM` latency reduction is more highlighted when lower latency network switches are used. Fortunately, the latency of network switch products is improving and today's ultra-low latency network switches offer port to port latency of less than 6ns [232].

Among all clusters, `webserver` benefits the most from `NetDIMM` because over 90% of its packets are less than 300Bytes and `NetDIMM` is more effective when transferring small packets. In addition, `webserver` traffic is within the datacenter and it traverses lesser hops to reach a destination compared with `database` traffic that is mostly inter-datecenter. Although `hadoop` traffic is local to the cluster, its packets are skewed to either small- or MTU-sized packets, therefore, `NetDIMM` latency reduction is the lowest for `hadoop` amongst the other two clusters.

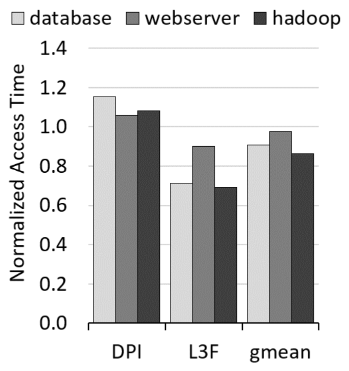


**Figure 5.15:** Normalized memory access latency observed by a co-running application when running deep packet inspection (`DPI`) and L3 forwarding (`L3F`); using servers with `NetDIMM` and replaying Facebook cluster traces.

Figure 5.15 shows the normalized memory access latency observed by a co-running application when running a `DPI` and `L3F` on servers with `NetDIMM`. The values are normalized to that of iNIC. Because `DPI` makes forwarding decisions based on the packet payload, the processor should fetch the entire packet to its caches and process both header and payload. Because an iNIC directly brings the received packets to the LLC, it does not consume memory channel bandwidth and if the processor is not congested, each received packet can be processed and forwarded before it gets evicted to the DRAM. However, `L3F` only needs packet header to decide where to forward a received packet, which is naturally done by nCache at `NetDIMM`. Based on this packet processing behavior, `DPI` and `L3F` are two ends of packet processing spectrum and any other applications falls between these two. Figure 5.15 shows that `NetDIMM` increases the memory access time by 5.7%∼15.4% when running `DPI` and improves it by 9.8%∼30.9% when running `L3F` compared with iNIC configuration. On average, `NetDIMM` improves the memory access latency by 9.3%, 2.4%, and 13.6% for `database`, `webserver`, and `hadoop` clusters respectively.

## 5.6 Related Works

**Novel Network Architecture.** Kim et al. [207, 233] proposed a caching mechanism inside NIC to reduce data communication over the PCI channel. The NIC cache is implemented using on-board DRAM devices and is managed by the operating system. Although this network architecture reduces the PCIe traffic, incoming packets still need to traverse the PCIe interconnect to reach the CPU. Furthermore, designing an efficient software managed data cache is challenging. Flajslik et al. [38] performed a detailed study on different sources of latency overhead in the network stack and found that minimizing the number of PCIe transactions is the key in designing a low latency NIC. They proposed a new NIC architecture called NIQ to reduce the communication latency, especially for small packets, by employing techniques such as embedding packets inside the buffer descriptors, custom polling, and creative use of caching policies. FlexNIC [209] is a network DMA interface design that reduces the packet processing overhead by enabling NIC to perform simple operations on the packets while exchanging them with the main memory. Offloading optimization is orthogonal to `NetDIMM` design and can be applied to `NetDIMM` to further improve the network performance. Liao et al. [202] proposal decouples the DMA descriptor management from other NIC functionality and moves it to processor side. This design aims to reduce the number of PCIe transactions and handle DMA buffers more efficiently. Larsen et al. [203] also introduced an integrated DMA engine to minimize the descriptor management overhead and PCIe transactions. Binkert et al. [206] proposed SINIC, which integrates a simple NIC into the processor die. SINIC uses PIO to exchange data between the processor, main memory and NIC. Although SINIC is effective in reducing the network latency, it has a high area cost. Furthermore, it is not suitable for high bandwidth communication due to the lack of a DMA engine and other capabilities of modern NICs. Compared with these works, `NetDIMM` completely removes the PCIe link between NIC and the processor, places a full-blown NIC near memory, and implements in-memory buffer cloning which in turn solves all the overheads of a conventional network sub-system.

Minnich et al. [234] proposed a memory-integrated NIC called MINI, that places a NIC behind the main memory DRAM modules. MINI implements a pseudo dual-port DRAM to share the DRAM space between the host and NIC. This requires arbitration signals between the host and NIC memory controllers. MINI need to redesign DRAM and memory controller interfaces to port to a new system architecture. MEMONet [235] and DIMMNET-2 [236] plug a NIC into a memory channel slot. Although these designs solve the PCIe bottleneck, they do not share the NIC and host address space and explicitly copy packets over the host memory channel for packet transmission and reception. Furthermore, these NICs can be used on a single memory channel system. On the other hand, `NetDIMM` seamlessly exposes its local memory address space to the host, minimizes the data movement between host and NIC, supports multi-channel memory systems, and lastly, `NetDIMM` does not require any change to the processor architecture and memory subsystem.

**Novel Interconnection Technology.** Alian et al. [21] introduced memory channel network (MCN) concept where they add a general-purpose mobile processor to a DIMM and expose the near-memory processors to the host processor as if they are connected through an Ethernet interface. They use memory channel to interconnect the remote nodes to the host processor. `NetDIMM` uses a similar concept to connect NIC, processor, and memory together. Open Coherent Accelerator Processor Interface (OpenCAPI), Cache Coherent Interconnect for Accelerators (CCIX), and Gen-Z are new interconnect standards under development that are mainly used to tightly couple processors and accelerators such as GPUs and FPGAs. CCIX is developed based on PCIe specifications and has PCIe drawbacks. The combination of DDR and such interconnection technologies provides unprecedented bandwidth and reduces data movement overhead by directly accessing the memory. Although CCIX, OpenCAPI and Gen-Z are three different standards, these are introduced and emerged to solve similar problems and they may merge into each other in the future. However, DDR standard is maintained and developed for over two decades and is the standard interconnection technology for memory. Moreover, the serial interconnects such as CCIX, OpenCAPI and Gen-Z cannot match the latency of a parallel DDR memory channel.

## 5.7 Conclusion

For decades, the focus of scale-out network system design was to optimize its bandwidth. However, with the emergence of ultra-low latency datacenter applications, a need for low latency scale-out networks has unfolded. In this chapter, building upon the near-memory processing concept and leveraging the asynchronous memory access of NVDIMM-P protocol, we designed and evaluated a near-memory NIC architecture called `NetDIMM`. `NetDIMM` integrates a full-blown NIC into the buffer device of an in-memory buffer-cloning capable DIMM. We developed supporting logic and a device driver to make the near-memory NIC available to applications running on a host processor. Finally, we implemented a new memory zone for `NetDIMM`'s local memory space and developed Linux kernel APIs to facilitate memory allocation from these memory zones. Such memory allocation significantly reduced the amount of data movement when processing network packets. Compared with a conventional PCIe NIC, `NetDIMM` improves the network latency by up to 52.9% without compromising the network bandwidth.

# Part III

# System Simulation and Modeling

# CHAPTER 6

# PARALLEL/DISTRIBUTED SIMULATION OF COMPUTER CLUSTERS

When analyzing a distributed computer system, we often observe that the complex interplay among the processor, node, and network sub-systems can profoundly affect the performance and power efficiency of the distributed computer system. Therefore, to effectively cross-optimize hardware and software components of a distributed computer system, we need a full-system simulation infrastructure that can precisely capture this complex interplay. Responding to the aforementioned need, we present `dist-gem5`, a flexible, detailed, and open-source full-system simulation infrastructure that can model and simulate a distributed computer system using multiple simulation hosts. Then we validate `dist-gem5` against a physical cluster and show that the latency and bandwidth of the simulated network sub-system are within 18% of the physical one. Compared with the single threaded and parallel versions of `gem5`, `dist-gem5` speeds up the simulation of a 63-node computer cluster by 83.1× and 12.8×, respectively.

## 6.1 Introduction

Single-thread performance of processors has not significantly improved as technology scaling has approached the fundamental physical limit. Meanwhile, emerging applications require computation across larger data sets. Consequently, distributed computing models such as MapReduce and MPI have thrived. This has increased the importance of building efficient distributed computer systems. The complex interplay among processor, node, and network sub-systems can strongly affect the performance and power efficiency of a distributed computer system. In particular, we observe that all the hardware and software layers of the network, including interface technology (*e.g.,* Ethernet, RapidIO, and InfiniBand), switch/router capability, link bandwidth, topology, traffic patterns, and protocols, significantly impact the processor and node utilization. Therefore, to build distributed computer systems that provide high-performance and power efficiency, a system architect must develop optimizations that cut across processor, node, and network sub-systems. Such cross-optimizations requires a detailed full-system simulator that can precisely models the entire distributed system. Currently, our community lacks a proper research infrastructure to study the interplay of these sub-systems. That is, we can evaluate them independently with existing tools, but not together.

`gem5` is one of the most widely used full-system simulators [160]. `gem5` can boot an operating system (OS), allowing researchers to evaluate various processor architectures while reflecting complex interactions between the processor and the OS. Several research groups have actively enhanced `gem5` to support various important features such as DVFS [41] and GPU models [237]. These features are now part of the official release of `gem5`. While the current official release of `gem5` supports thread-based parallelism within a single host process , it does not support parallelism across multiple simulation hosts yet. Consequently, it can only model and simulate a limited number of nodes.

In this chapter, we present `dist-gem5`, a distributed version of `gem5`, consolidating two independent development efforts, `pd-gem5` [159] and `multi-gem5` by the University of Illinois and ARM Ltd, to support the simulation of multiple nodes using multiple simulation hosts. The primary distinctions between `dist-gem5` and `pd-gem5` are as follows. (1) `dist-gem5` uses a single channel (TCP socket) to forward both synchronization and data messages between a switch node and a full-system node. This prevents data messages from bypassing synchronization messages (due to the strict ordering between TCP packets) and thus any straggler packets [159, 238]. (2) `dist-gem5` improves the check-pointing feature of `pd-gem5`. (3) `pd-gem5` is tightly coupled with the Ethernet protocol, whereas `dist-gem5` is protocol agnostic and can be easily extended to work with other network technologies.

Our goal is to develop `dist-gem5` that can provide a scalable, fast and detailed simulation infrastructure for modeling and evaluating large computer clusters. We observe that it is not scalable to use one physical simulation host to model and evaluate a large computer cluster, limiting the number of nodes that we can model and evaluate. In simulating a computer cluster, if the number of simulated nodes is more than that of physical cores of a simulation host, we demonstrate that `dist-gem5` with multiple simulation hosts can offer considerably lower simulation time than `dist-gem5` with a single simulation host (parallel/distributed simulation using many cores across multiple simulation hosts versus parallel simulation using multiple cores in a simulation host). A parallel/distributed simulation of a 63-node computer cluster offers 12.8× lower simulation time that running the same simulation using one simulation host. This shows that parallel simulation that has been proposed in previous works [239, 240, 241] is not sufficient for simulating emerging large-scale computing systems.

In this chapter, we first introduce `dist-gem5` and describe its components in detail (Sec. 6.2). Then we verify that modeling a computer cluster using `dist-gem5` generates exactly the same results as if we model the cluster using a single-threaded `gem5` with the same configurations (Sec. 6.3.1). That is, `dist-gem5` simulation is deterministic although it uses multiple simulation hosts connected by physical network with varying latency and bandwidth over time. Note that the goal of this writing is not to validate the existing performance models of `gem5`. Therefore, we focus on demonstrating that `dist-gem5` can precisely model and evaluate the network sub-system performance of a physical computer cluster (Sec. 6.3.2). Then we evaluate the speedup and scalability of `dist-gem5` by simulating 3- to 63-node computer clusters (Sec. 6.3.3). To evaluate the synchronization overhead of `dist-gem5`, we simulate a 16-node computer cluster and sweep synchronization quantum from 0.5 to 128$\mu$s (Sec. 6.3.4). In Sec. 6.4, we use `dist-gem5` and evaluate the scalability of the main five kernels of MPI implementation of NAS parallel benchmarks [162], and study `dist-gem5`'s speedup and scalability. Section 6.5 discusses related works to `dist-gem5`.

## 6.2 `dist-gem5` Architecture

Figure 6.1 shows `dist-gem5` simulating an eight-node server rack connected by a Top-of-Rack (TOR) network switch using three simulation hosts. Each `gem5` instance (process) is shown as a box labeled with the `gem5` logo, which can simulate a node in a full-system mode or a network switch. In Fig. 6.1 "p" prefix stands for "physical" and "s" prefix stands for "simulated." For example, "sNIC" stands for "simulated NIC." `dist-gem5` simulates such a

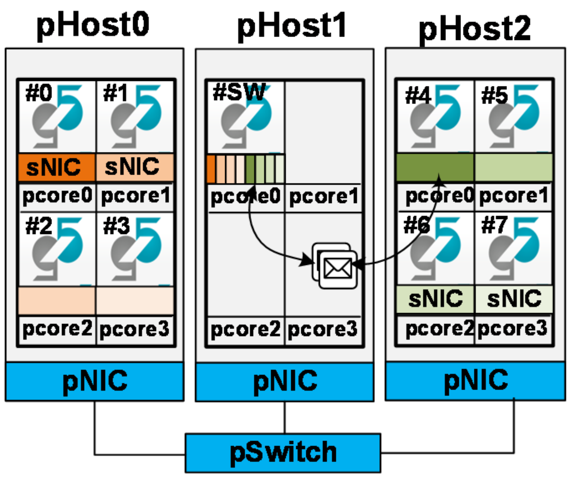


**Figure 6.1:** `dist-gem5` modeling a computer cluster using 3 simulation hosts.

computer cluster as follows: One `gem5` instance simulates a network switch on one simulation host ("pHost 1" in Fig. 6.1). Eight `gem5` instances running on the other two simulation hosts ("pHost0" and "pHost1" in Fig. 6.1) simulate eight nodes of the simulated cluster. As is the case in Fig. 6.1, `dist-gem5` may dedicate a physical core of a simulation host to run just a network switch to prevent a process simulating the network switch from being the simulation bottleneck, or it may run a network switch model within a `gem5` process that simulates a node in full-system mode. In this way, `dist-gem5` can model and simulate a distributed computer system in any given network topology (*e.g.,* star, ring, and mesh topology) at any scale.

### 6.2.1 Core Components

In this sub-section, we will describe four core components of `dist-gem5`: (1) packet forwarding, (2) synchronization, (3) distributed checkpointing, and (4) the network switch model. These enhancements allow us to precisely model and efficiently simulate a distributed computer system in a desired network topology.

**Packet forwarding.** `dist-gem5` forwards each network packet generated by a simulated NIC device to a port of a simulated network switch. The forwarded packets travel through TCP sockets, which establish physical communication channels between `gem5` processes. When a packet arrives at

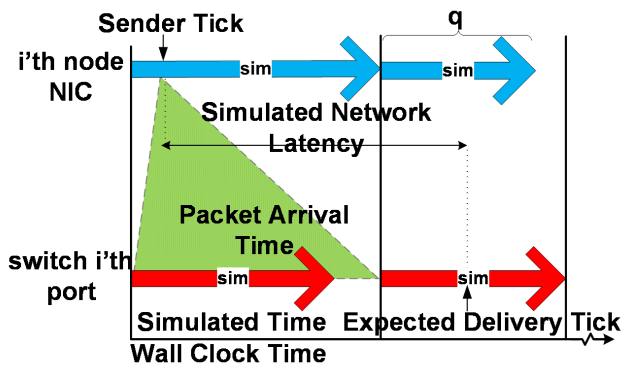


**Figure 6.2:** Distributed check-pointing in `dist-gem5`.

a `gem5` process simulating a network switch, it traverses through the entire simulated network topology (*i.e.,* one or more simulated Ethernet switches connected through simulated Ethernet links). After that, the simulated packet is forwarded to the target simulated NIC device(s) through a TCP socket of a simulation host. Figure 6.1 depicts in-flight forwarded packets (represented by small envelopes) between the simulated NIC of node#4 and port#4 (dark green port) of the simulated switch.

`dist-gem5` launches a dedicated receiver thread that runs in parallel with the main simulation thread within each `gem5` process. The main simulation thread is responsible for progressing the simulation by processing the event queue. The receiver thread processes all incoming packets and inserts the corresponding receive frame events into the event queue without interrupting the main simulation thread. Note that receiver thread does not perform any work independent from the main simulation thread. It is just an auxiliary thread that frees the main simulation thread from polling on a TCP connection. Such asynchronous processing of incoming packets helps to hide the extra overhead of packet forwarding with respect to the total simulation time.

**Simulation accuracy and packet forwarding.** The simulated time of parallel `gem5` processes may progress at different rates. It is due to the varying numbers of events that need to be processed by each `gem5` and the potential differences in the physical hardware resources (including varying load levels of shared resources). We must ensure that each network packet created by a simulated NIC arrives at the target `gem5` process(es) "on time" (*i.e.,* before the simulated time in the target `gem5` process passes the expected receive time). The expected time of receiving a packet (denoted by $er$) depends on the simulated time at which the packet is sent ($st$) and the latency ($lat$) and bandwidth ($bw$) of the simulated network link. We can compute $er$ as follows:

$$er = st + lat + \frac{ps}{bw}$$

where $ps$ is the packet size.

Figure 6.2 illustrates a quantum-based synchronization technique that we use to ensure timely packet delivery between `gem5` processes. The x-axis is the wall-clock time while the thick horizontal arrows represent the progress in simulated time of two `gem5` processes (where the upper one sends a simulated packet to the lower one). The quantum ("q") defines the amount of simulated time that any `gem5` process can proceed freely before it has to wait for all the others at a global synchronization barrier. The synchronization barrier must also flush the inter `gem5` processes communication channels. That is, every in-flight forwarded packet must arrive at the target `gem5` process before the barrier completes. These imply the following invariant property: a forwarded packet always arrives at the target before the target `gem5` completes simulating the quantum in which the simulated send time of that packet falls.

If we keep the quantum equal to (or less than) the simulated network link latency, then the expected delivery time will always fall in a future quantum when the packet arrives at the target. This ensures that no packet will miss the expected delivery time. However, synchronization barriers incur runtime overhead. That is, the less frequent the synchronization is, the smaller the overhead is. This implies that the optimal choice for the quantum size is the simulated network link latency.

**Distributed checkpointing.** Checkpoints can substantially reduce the simulation time needed for system explorations. We can start the simulation of an application in fast-forward mode (*i.e.,* fast functional simulation mode) and dump a checkpoint when we reach a Region of Interest (ROI). The checkpoint stores all the pertinent simulation states. Then we can restore the simulation states at the beginning of a ROI and run simulations in detailed mode.

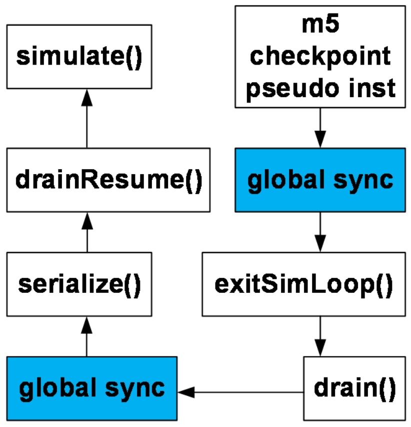


**Figure 6.3:** Distributed check-pointing in `dist-gem5`.

In case of `dist-gem5`, a global checkpoint consists of all the per-process checkpoints taken by the parallel `gem5` processes. However, a per process checkpoint captures only the internal state of a single `gem5` process, while `dist-gem5` also contains external states on inter-process communications. In-flight forwarded packets between `gem5` processes may cause the `dist-gem5` checkpoint to be incomplete. Hence, dumping a checkpoint needs coordination among the parallel `gem5` processes. The white boxes in Fig. 6.3 show the key steps to create a checkpoint in `gem5`. The "serialize()" step performs the actual checkpoint write. Prior to serialization, the simulator needs to be in a consistent state. The "drain()" step progresses the simulation as little as possible just to reach the next consistent state.

`dist-gem5` needs two extra synchronization steps shown in the blue boxes in Fig. 6.3. First, we need a global synchronization barrier to notify every `gem5` process that they need to dump a checkpoint. When a global synchronization completes, the inter-process communication channels are empty. However, during the subsequent `drain()` step simulation may proceed and new packets may get forwarded. Therefore, `dist-gem5` uses another global synchronization just before `serialize()` to flush the inter-process communication channels again. As an optimization for simulation speed during fast-forwarding, `dist-gem5` provides an option to disable synchronization and start it after restoring from a checkpoint or before entering a ROI.

**Network switch.** We implement an Ethernet switch device ("EtherSwitch") in `gem5`, which operates at layer 2 of the Open Systems Interconnection (OSI)

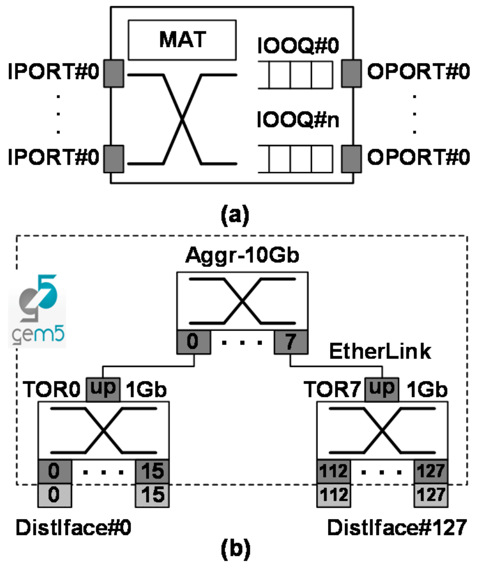


**Figure 6.4:** (a) EtherSwitch architecture. (b) Sample hierarchical tree topology modeling with `dist-gem5`.

model. Figure 6.4(a) shows the architecture of `EtherSwitch`. `EtherSwitch` learns [242] the MAC address of the connected device to each port and constructs a MAC Address Table (MAT). Then it uses MAT information to forward incoming packets to the correct port(s) based on the packet's destination MAC address. If there is no entry for a MAC address in MAT, `EtherSwitch` broadcasts the packet to all of its ports. Each MAT entry has a Time-to-Live (TTL) parameter. If a MAT entry is not used for TTL time units, then `EtherSwitch` removes that entry from MAT. Users can configure the switching delay, switching bandwidth, FIFO size of output port, and TTL as command line parameters.

To guarantee deterministic simulation, we have to make sure that packets get routed in the same order (with respect to simulated time) inside `EtherSwitch`. As soon as a packet is received on an input port ("IPORT" in Fig. 6.4(a)), the packet is enqueued in one or several in-ordered output queues ("IOOQ" in Fig. 6.4(a)). IOOQ is different from a simple FIFO in a sense that it tags each packet with its entry time and its associated input port. If two packets have a same entry time stamps, then IOOQ reorders them based on their input ports. With this, it is guaranteed that there is no reordering happening inside `EtherSwitch` when two packets reach `EtherSwitch` at a same simulated time.

**Network topology exploration.** EtherSwitch enables users to construct different Ethernet network topology by instantiating several EtherSwitch simObjects and connect them together via Ethernet link ("EtherLink") simObjects in gem5. Figure 6.4(b) shows how a two-level hierarchical network topology can be modeled using dist-gem5. The entire network topology can be modeled inside one gem5 process (what we have in Fig. 6.4(b)) or be distributed over several gem5 processes that each can potentially simulates a full-system node. In the Fig. 6.4(b), we have nine EtherSwitches, *i.e.,* 1× eight-port 10GbE aggregate ("Aggr") switch and 8× seventeen-port (sixteen downlink ports, one uplink port) 1GbE network switches. Each downlink port of TOR switches is connected to a "DistIface" which connects the downlink port to a node's NIC.

Although the current networking protocol supported by dist-gem5 is Ethernet, because its synchronization and packet delivery are protocol agnostic, dist-gem5 can easily be extended to model other networking technologies (*e.g.,* InfiniBand).

### 6.2.2 Deterministic Execution

Non-determinism in simulation may jeopardize confidence in results. The released gem5 provides deterministic simulation execution. dist-gem5 introduces a new source of potential non-determinism in the form of non-deterministic physical communication among gem5 processes. The delivery time for messages traveling through TCP sockets can vary substantially depending on the current state of the physical system. We need to make sure that non determinism in the physical arrival time of any forwarded packet does not affect the simulated timing at the target gem5.The quantum-based synchronization flushes the in-flight messages at each global barrier. This implies that we only need to consider non-deterministic message arrivals between two synchronization barriers (*i.e.,* within the "active quantum"). As discussed in the previous section, expected packet delivery always falls into a future quantum so the order of simulated receive events are not affected by the physical arrival time of the forwarded packets. In other words, the total order and simulated timing of all receive events is independent from that of the physical packet arrivals. This ensures that dist-gem5 simulations are

**Table 6.1:** Parameters of each simulated node.

| **Parameters** | **Values** |
|---|---|
| Number of cores | 4 |
| Superscalar | 4 ways |
| Integer/FP ALUs | 3/2 |
| ROB/IQ/LSQ entries | 128/36/72/42 |
| Branch predictor | Bi-Mode |
| L1I/L1D/L2 size (KB) | 64/64/2048 |
| L1I/L1D/L2 associativity | 2/4/8 |
| DRAM | 8GB DDR3_1600 |
| Network interface driver | Intel 82574GI Gigabit Ethernet |
| Network link | 10Gbps with 1$\mu$s latency |
| Operating System | Linux Ubuntu 14.04 |

deterministic.

## 6.3 Evaluation: Determinism, Network Sub-System Validation, and Speedup

In this section, we show evaluation results for `dist-gem5` runs. First, we demonstrate that `dist-gem5` is deterministic and its simulation results are identical to the single threaded `gem5` simulation model. Second, we show that `dist-gem5` can accurately simulate a physical cluster. Third, we study the speedup improvement of `dist-gem5`. Lastly, we vary the synchronization quantum size to analyze the impact of the synchronization overhead of `dist-gem5` on the speedup. Unless stated otherwise, we configure `dist-gem5` nodes with the parameters tabulated in Table 6.1 for our experiments.

### 6.3.1 Verification of Determinism

We verify `dist-gem5`'s synchronization, in-order packet delivery, and deterministic simulation using an equivalent `gem5` model. Currently, `gem5` can simulate a two-node computer cluster connected with an `EtherLink` together ("dual" mode configuration). We construct an identical simulated clusters to `dist-gem5` by instantiating several full-system nodes and connect them to an `EtherSwitch` model using several `EtherLinks`. Figure 6.5 illustrates our `gem5` model for simulating an eight-node computer cluster

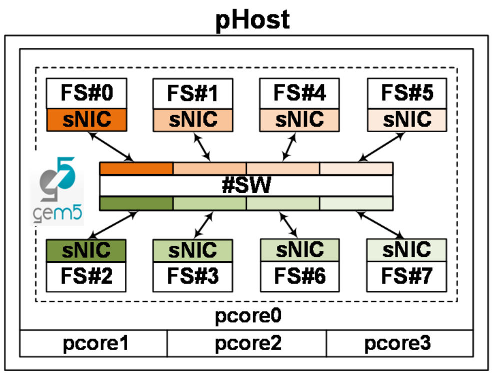


**Figure 6.5:** `gem5` modeling of an eight-node computer cluster.

(same cluster size as Fig. 6.1). As shown in the figure, all the nodes and the networking sub-system of the eight-node cluster are modeled using one `gem5` process running on one of the cores of a physical host. We configure `dist-gem5` and `gem5` to simulate identical 4-, 8-, and 16-node computer clusters and run `Memcached` and `httperf` on them. For each cluster configurations, we run one (`Memcached` or `httperf`) server on one of the nodes and run one (`Memcached` or `httperf`) client per node on the remaining nodes. We configure each node with the parameters tabulated in Table 6.1. The exact configuration of `Memcached` and `httperf` servers are explained in Sec. 6.3.2. For all the `dist-gem5` and `gem5` experiments with different configurations, we get exactly the same simulation results. Simulation results are all the `gem5` statistics that report the activities of different `gem5` hardware models, *e.g.,* simulated time, total number of committed instructions, bandwidth consumption by DRAM and NIC, etc.

Beside verification purposes, the seamless integration of `dist-gem5` into `gem5` framework is valuable for simulating a hierarchical network, using a combination of multi-threaded `gem5` processes and `dist-gem5`, *e.g.,* model a rack in multi-threaded[1] mode on a multi-processor machine and use `dist-gem5` to build up a simulation of multiple racks. This can reduce the overhead of synchronization among `gem5` processes, further improving `dist-gem5`'s scalability (Sec. 6.3.3)

[1]The multithreaded `gem5` is not publicly used by the `gem5` community yet

## 6.3.2 Validation of Network Sub-System Model

We validate `dist-gem5` against a physical computer cluster consisting of four nodes, each of which has one quad-core AMD A10-5800K, two 8GB DDR3 DIMMs and one Intel 10-Gigabit x540-AT2 Ethernet controller. Unfortunately, `gem5` lacks a 10 Gigabit Ethernet card model, preventing us from validating it against a 10-Gigabit physical cluster. Therefore, we use an HP 1410 Gigabit Ethernet switch to connect nodes together and force our Ethernet network to work with 1-Gigabit per second speed[2]. We use Intel DPDK [168] to accurately measure per Ethernet packet latency on our physical setup. We run five network-intensive applications, including `iperf` [161], `Memcached` [244], httperf [245], tcptest [246], and netperf [247], and report the average request latency and achieved bandwidth results. We run three clients (one client per node) which send requests to one server running on a separate node. `Memcached` server has 1GB caching capacity and is warmed up with a 1.2GB scaled dataset. Each of `Memcached` and `httperf` clients sends request to the server at 20% of the maximum load that the server can sustain. Thus, the total load on the server from three clients is 60% of the maximum sustained load.

Before validating `dist-gem5`, we run SPEC CPU2006 benchmarks on `gem5` and the physical machine and tune `gem5` parameters to closely model the physical node. On average, the performance of the tuned `gem5` is 6% higher/lower than the physical setup.

On both physical (denoted by `phys`) and `dist-gem5` clusters, we run three `iperf` clients on three separate nodes which send 500B UDP packets to the fourth node in the cluster. In the physical setup, in order to minimize the effect of the software network stack on the latency measurements, we use DPDK to measure round-trip latency of packets at layer 2 of the OSI model. We sweep `iperf` client's bandwidth (using the "–bandwidth" option) and measure the round-trip latency at different load levels. Figure 6.6 shows the roundtrip latency versus bandwidth for `dist-gem5` and `phys`. As expected, the round-trip latency dilates when the aggregated client's load reaches to 1 Gigabit per second (Gbps). This happens because the packet buffers in the Ethernet switch start to fill up. We also observe packet drops in both

[2]Because of the Autonegotiation [243] feature, NICs and the Ethernet switch agree on a common transmission rate which is the minimum rate supported by all network devices (here 1Gbps)

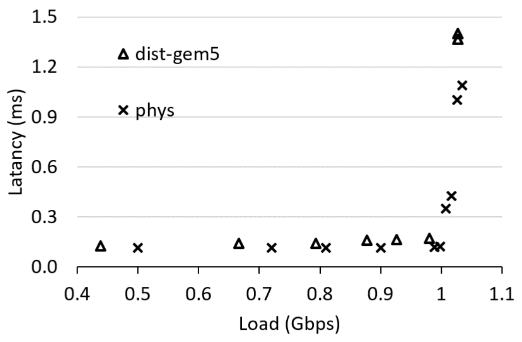


**Figure 6.6:** Comparison of latency versus bandwidth.

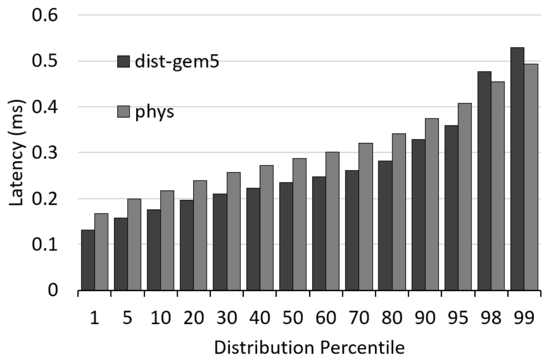


**Figure 6.7:** `Memcached` response time distribution.

`dist-gem5` and `phys` at load levels close to 1Gbps. As shown in Fig. 6.6, `dist-gem5` can accurately model the network queuing latency and closely follows the behavior of the physical cluster.

Figure 6.7 shows the distribution of response time of `Memcached` for `dist-gem5` and `phys`. Up until 95th percentile response time, on average, `dist-gem5` has 17.5% lower response time compared with `phys`. 98th and 99th percentile response times of `dist-gem5` are 6% higher than `phys`. The response time that we get from `dist-gem5` is constantly lower than `phys` except for a few requests. We track down the network delay for these requests and confirm that network latency does not contribute to their high response time. Instead, these few requests experience an unexpectedly high delay in the `Memcached` server. We suspect that `Memcached` thread load imbalance causes these unexpected spikes in the response time [248]. Because our focus for validating `dist-gem5` is particularly on the network sub-system, we did

**Table 6.2:** Latency and bandwidth comparison.

| Benchmark | dist-gem5 | phys | Error |
|---|---|---|---|
| httperf | 3.62(ms) | 3.68(ms) | – 1.6% |
| tcptest | 897.2(Mbps) | 958.6(Mbps) | −6.4% |
| netperf | 865.5(Mbps) | 942.2(Mbps) | −8.1% |

not strive to tune each node's parameters to exactly model the physical systems. Thus, we believe the mismatch in node's configuration (both hardware and software) is the main contributor to the discrepancy between `dist-gem5` and `phys` results.

In addition to `iperf` and `Memcached`, we run `httperf`, `tcptest`, and `netperf` on physical cluster and `dist-gem5`. `httperf` is a networking benchmark tool which implements a Hypertext Transfer Protocol (HTTP) client that sends request to an Apache web server. We run `tcptest` and `netperf` to measure the maximum sustainable TCP bandwidth of `dist-gem5` and `phys` clusters. Similar to `iperf`, we run three (`httperf`, `tcptest`, and `netperf`) clients on three separate nodes in which each of the clients send requests to one (`httperf`, `tcptest`, and `netperf`) server node. Table 6.2 shows the average `httperf` request response time and TCP bandwidth measured by `tcptest` and `netperf` for `dist-gem5` and `phys`. Compared with `phys`, `dist-gem5` just has 1.6% lower latency, and 6.4% and 8.1% lower bandwidth for `httperf`, `tcptest`, and `netperf`, respectively.

### 6.3.3 Comparison of Speedup

We evaluate `dist-gem5`'s speedup by comparing the simulation time of modeling 3- to 63-node computer clusters using the following simulation techniques.

**Single threaded** `gem5` (`st-gem5` configuration in Fig. 6.8). We model full-system nodes using one single threaded `gem5` process running on one dedicated physical machine (AMD machine described in Sec. 6.3.2). Figure 6.5 illustrates an eight-node `st-gem5` model.

**Parallel** `gem5` (`p-gem5` configuration in Fig. 6.8). We use `dist-gem5` to model each full-system node under a separate `gem5` process, but rather than distributing `gem5` processes on several physical hosts, we use one physical machine to simulate the entire cluster. That is, `gem5` processes can run in

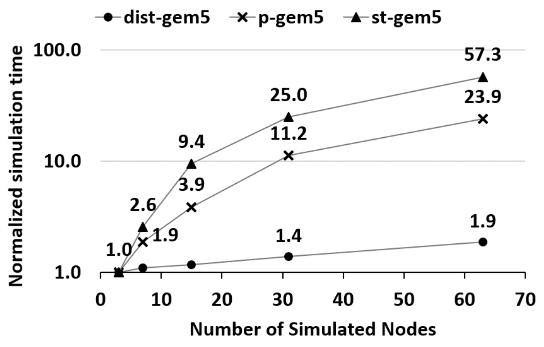


**Figure 6.8:** Speedup comparison of single threaded, parallel and parallel-distributed `gem5` simulation.

parallel on each of the four cores of the physical machine.

**Parallel-distributed** `gem5` (`dist-gem5` configuration in Fig. 6.8). This is the same as `p-gem5`, but we use as many physical machines as we need to assign each `gem5` process to one physical core. Table 6.3 shows the number of physical nodes and cores that we use to simulate the computer cluster summarized in the first row ("xnys" translates to `x` nodes and `y` `EtherSwitches`) with `st-gem5`, `p-gem5` and `dist-gem5`. As we dedicate one `gem5` process to run a `EtherSwitch`, we set the simulated cluster size equal to $2^m - 1$ where $m$ is an integer value to utilize all the cores of all the physical hosts used in `dist-gem5` configuration.

We run one `httperf` client on each simulated node and configure it to send requests to one unique node in the cluster. This way, independently of cluster size, the load per node at different cluster sizes is always constant. Therefore, we can have a fair scalability and speedup evaluation by comparing the simulation time of different configurations when they perform exactly the same number of work-units; in our experiments, number of requests serviced

**Table 6.3:** Breakdowns of the end-to-end latencies for transmitting and receiving a single TCP 1.5KB/9KB packet.

| | | 3n1s | 7n1s | 15n1s | 31n1s | 63n1s |
|---|---|---|---|---|---|---|
| `st-gem5` | #`gem5`/core | 1 | 1 | 1 | 1 | 1 |
| | #phyNodes | 1 | 1 | 1 | 1 | 1 |
| `p-gem5` | #`gem5`/core | 1 | 2 | 4 | 8 | 16 |
| | #phyNodes | 1 | 1 | 1 | 1 | 1 |
| `dist-gem5` | #`gem5`/core | 1 | 1 | 1 | 1 | 1 |
| | #phyNodes | 1 | 2 | 4 | 8 | 16 |

per second represents the work-unit.

We run one `httperf` client on each simulated node and configure it to send requests to one unique node in the cluster. This way, independently of cluster size, the load per node at different cluster sizes is always constant. Therefore, we can have a fair scalability and speedup evaluation by comparing the simulation time of different configurations when they perform exactly the same amount of work-units; in our experiments, number of requests serviced per second represents the work-unit.

Figure 6.8 shows the simulation time (wall-clock time) of `st-gem5`, `p-gem5`, and `dist-gem5` for simulating three seconds of `httperf` runtime, normalized to simulation time of `3n1s` configuration. Note that each data-series is normalized to the simulation time of `3n1s` configuration of the same data-series to show how simulation time scales when scaling simulated cluster size for each simulation technique. As depicted in Fig. 6.8, simulating `63n1s` takes 57.3× and 23.9× longer than simulating `3n1s` with `st-gem5` and `p-gem5`, respectively. On the other hand, it just takes 1.9× longer than `3n1s` to simulate `63n1s` with `dist-gem5`, showing the great scalability of `dist-gem5`. Note that the load on the switch process is not constant and linearly scales with cluster size although we keep the load per full-system node constant when scaling up the simulated cluster size. Therefore, the slowdown of `dist-gem5` is caused by both synchronization and network switch process overhead.

One important observation from Fig. 6.8 is that simulation time of `p-gem5` scales linearly with number of nodes until `15n1s`; 1.9× and 3.9× for `7n1s` and

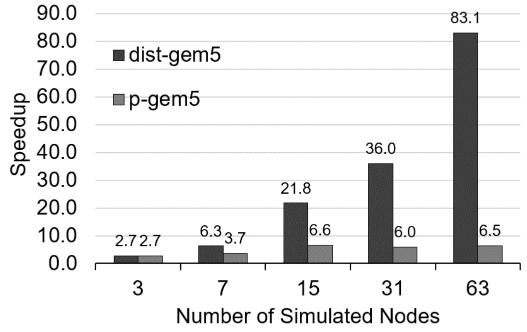


**Figure 6.9:** Speedup of parallel `gem5` (*i.e.,* `p-gem5`) and parallel-distributed `gem5` (*i.e.,* `dist-gem5`) simulations compared to single-threaded `gem5`.

15n1s configurations, respectively, which in turn has approximately 2× and 4× more nodes than 3n1s. However, for 31n1s and 63n1s configurations, because p-gem5 uses all the available memory in one physical system, it experiences memory thrashing and its simulation time increases super linearly; p-gem5 simulation time increases 11.2× and 23.9× for 8× and 16× increase in the simulated cluster size, respectively. This shows the necessity of distributed simulation for simulating large-scale clusters.

Figure 6.9 shows the speedup of p-gem5 and dist-gem5 over st-gem5. Note that p-gem5 is identical to dist-gem5 for 3n1s configuration as dist-gem5 uses only one physical node to simulate the cluster. For 63n1s configuration, p-gem5 and dist-gem5 provides 6.5× and 83.1× speedup over st-gem5, respectively. p-gem5 speedup saturates and does not go over 6.6× due to the thrashing phenomena that we explained earlier in this section.

### 6.3.4 Overhead of Synchronization

To evaluate the sensitivity of dist-gem5's speedup to synchronization quantum size (hereafter sync-quantum), we run httperf (with the same setup explained in Sec. 6.3.3) on dist-gem5 simulating a 16-node computer cluster and sweep Ethernet link delay parameter from 0.5$\mu$s to 128$\mu$s. Changing Ethernet link delay effectively changes sync-quantum as the sync-quantum of dist-gem5 is always set to Ethernet link delay to preserve the deterministic simulation (Sec. 6.2.2).

The bars in Fig. 6.10 show the normalized simulation time of httperf

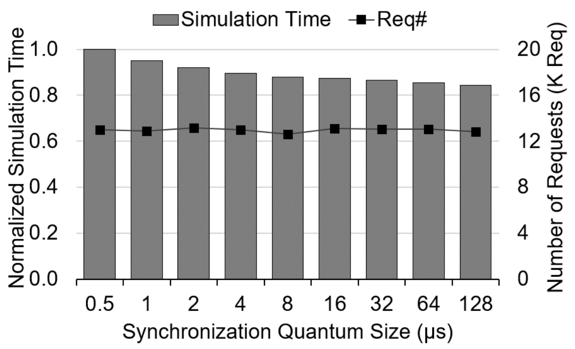


**Figure 6.10:** Simulation time of a 16-node computer cluster running httperf for different synchronization quantum sizes.

for different `sync-quantum`. The simulation times are normalized to the simulation time of the run with the smallest `sync-quantum` (0.5$\mu$s). Because we change Ethernet link delay to change `sync-quantum`, the simulation results of different quantum sizes are not identical. The line graph in Fig. 6.10 shows the total number of HTTP requests that are sent to the Apache servers for each `sync-quantum`. The variance from average of total simulated requests across different `sync-quantums` (*i.e.,* Ethernet link delays) is at most 2.6%. This small variation indicates that `httperf`'s performance is not sensitive to the Ethernet link delay (at least for this specific experiment that we are running) and the simulated work-units are approximately the same with different `sync-quantums`.

As shown in the Fig. 6.10, increasing `sync-quantum` consistently improves `dist-gem5`'s simulation time. This observation is expected as increasing `sync-quantum` lowers the synchronization frequency and consequently the overall synchronization overhead. However, the speedup gains from doubling `sync-quantum` decrease as it gets larger. Doubling `sync-quantum` from 0.5$\mu$s to 1$\mu$s, 1$\mu$s to 2$\mu$s, and 2$\mu$s to 4$\mu$s improves simulation speed by 4.9%, 3.1%, and 2.3%, respectively. Increasing `sync-quantum` by a factor of 256× (from 0.5$\mu$s to 128$\mu$s) improves simulation speed just by 15.7%. This experiment shows that `dist-gem5` synchronization is efficient and does not incur significant overhead even with small a `sync-quantum`.

## 6.4 In-Depth Evaluation: Scalability

In this section, we use `dist-gem5` to further evaluate the scalability of `dist-gem5` using the MPI implementation of NAS Parallel Benchmark (NPB) [162].

We run the small dataset size of NPB comprised of five kernels (*i.e.,* Integer Sort (`IS`), Embarrassingly Parallel (`EP`), Conjugate Gradient (`CG`), Multi-Grid on a sequence of meshes (`MG`), and discrete fast Fourier Transform (`FT`)). We compile each with 4 to 128 MPI processes ("NPROC" parameter). We run each binary on as many simulated nodes as needed to keep the MPI processes per simulated core equal to one in all configurations. Because each simulated node has four cores (Table 6.1), we use 1, 2, 4, 8, 16, and 32 nodes for running a binary with 4, 8, 16, 32, 64, and 128 MPI processes, respectively. Except

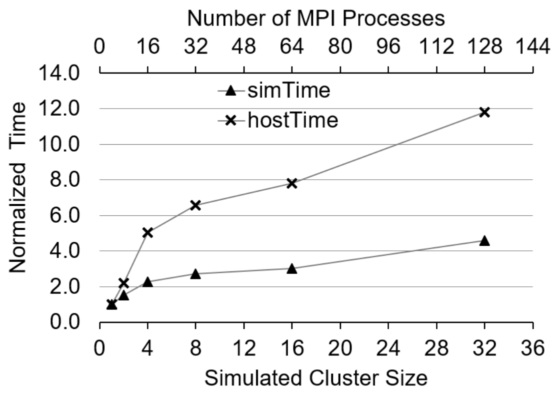


**Figure 6.11:** Simulation time (Host) and simulated time (Sim) versus number of processes (*i.e.,* cluster size) for CG.

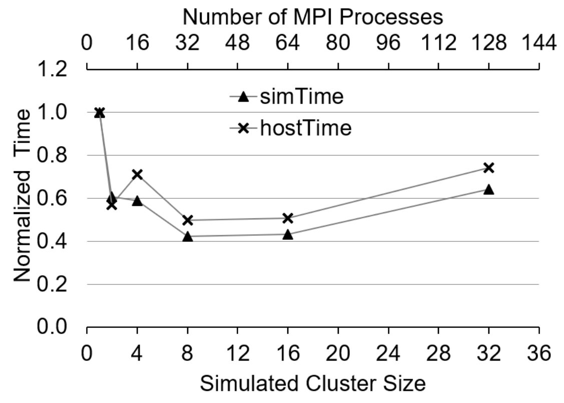


**Figure 6.12:** Simulation time (Host) and simulated time (Sim) versus number of processes (*i.e.,* cluster size) for EP.

for 1 node simulation, we use `dist-gem5` to run the kernels.

Figure 6.11 shows *simTime* (the simulated time that takes to run one kernel from beginning till end) and also *hostTime* (wall-clock time that takes to simulate one kernel), normalized to the single node (*i.e.,* 4 MPI processes) kernel, of two representative NPB kernels for different simulated cluster sizes (and consequently different number of MPI processes). As shown in Fig. 6.11, `CG` with 4 MPI processes offers the shortest *simTime*. Increasing the number of MPI processes (and consequently increasing the cluster size) leads to a higher *simTime* for `CG`. As expected, because of the increase in *simTime*, *hostTime* of `CG` increases for larger cluster sizes. Compared with the single node (4 MPI processes) data point, *simTime* and *hostTime* of a 32-node configuration increase by 4.6× and 11.8×, respectively.

Figure 6.12 shows the scalability results for `EP` kernel. Unlike `CG`, `EP` benefits from more MPI processes and larger cluster size. *simTime* of `EP` constantly decreases from single-node to eight-node cluster size. Its *simTime* starts to increase from 16-node cluster size and beyond. Compared with single node cluster, `EP` running on an eight-node cluster offers 58% lower *simTime*.

Except for four-node data point, *hostTime* follows the exact same trend as *simTime*: an increase in *simTime* results in an increase in *hostTime* and vice versa. However, at the four-node data point, we see an unexpected increase in *hostTime*. The reason for this anomaly roots from the core microarchitecture of the AMD machines (AMD Bulldozer microarchitecture) that we use to run our experiments (Sec. 6.3). These machines have two clusters and two cores per clusters. The cores within a cluster share the front end, the floating-point unit and the L2 cache. Therefore, running two processes on different cores of a same cluster can hurt the performance of each processes due to contention on the share resources. When we simulate a two-node computer cluster using `dist-gem5`, we have two `gem5` processes simulating full-system nodes and one simulating an `EtherSwitch`. The `EtherSwitch` process is a very light weighted process compared to the two full-system processes and effectively we are running two `gem5` processes on one physical host, and consequently one `gem5` process per AMD Bulldozer cluster. For simulating a four-node configuration, we run four full-system `gem5` processes on one AMD machine and run one switch process on a separate machine, using two physical machines to run the simulation. With this setup, each pair of full-system `gem5` processes shares one AMD Bulldozer cluster which degrades per `gem5` process performance. Furthermore, simulating all `gem5` processes in one physical host (two-node simulation) is more efficient than running on multiple physical hosts (four-node simulation and higher) due to lower synchronization overheads (off-chip versus on-chip communication). Sharing processor resources and higher synchronization overhead are the two contributors to the unexpected increase in the *hostTime* of the four-node cluster size. Compared with the four-node, *hostTime* of the eight-node configuration is lower as its *simTime* is significantly lower. After all, `gem5` is an event based simulator and *hostTime* is a function of the event frequency and event type. Technically, it is not true to tie *hostTime* to *simTime* and always expect that increasing (decreasing) *simTime* will increase (decrease) *hostTime*, especially for full-system simulation when we have complex interactions between the various components within a system.

The other three kernels in NPB exhibits the same scalability behavior of either `CG` or `EP`. `MG` and `IS` behave exactly the same as `CG` and their optimum *simTime* point is running on a single node using 4 MPI processes. `FT` follows the same scalability pattern as `EP` and its optimum *simTime* is when running on eight nodes using 32 MPI processes.

## 6.5 Related Work

Wisconsin Wind Tunnel (WWT) was developed to simulate a multi-processor system using multi-processors [249], and it pioneered the quantum-based technique to synchronize simulation processes and perform parallel simulation. SST [250] is an open-source, multi-scale, multi-component parallel architectural simulator. By importing `gem5` along with several other device models, such as NICs and routers, as components, SST can simulate HPC systems. However, a proper integration of `gem5` with SST is challenging as `gem5` continuously evolves. In contrast, `dist-gem5` is part of the official release of `gem5`, and thus it does not pose any integration challenge as SST. Furthermore, SST is not validated as a multi-component simulator and it relies on validation of components in isolation.

COTson is a full-system simulator for multi-core processors and scale-out clusters [238]. COTson combines individual node simulators to form a cluster simulator. It jointly uses a fast functional emulator (*i.e.,* AMD's SimNow [251]) and timing models to explore the trade-off between simulation precision and speed. COTSon supports (i) a dynamic sampling feature, which keeps track of SimNow's simulation statistics to identify the phases of simulated workload and enables/disables timing models based on these phase changes; and (ii) an adaptive quantum synchronization feature to support the trade-off between simulation precision and speed. In contrast, `dist-gem5` preserves simulation determinism by setting `sync-quantum` to the minimum link delay. Furthermore, `dist-gem5` can defer the synchronization start time to improve simulation speed. A key advantage of COTson is its high speed as COTson uses SimNow for fast forwarding. However, it is also a key disadvantage since it can only support x86 ISA. It is critical to support

ARM ISA with the growing interest in using ARM processors for servers. The key advantages of `dist-gem5` are its ability to simulate multiple ISAs, open-source code, and its active community.

MARSSx86 is a cycle-level full-system simulator, specifically for multi-core x86-64 architectures [239]. Similar to COTson, it takes a hybrid simulation methodology leveraging QEMU for emulation. MARSSx86 supports only a functional NIC model. In contrast, the `gem5`'s NIC model is an event-based one that can be more easily adapted to precisely model the performance aspects of a NIC.

Graphite is a parallel/distributed simulator for many-core processors [252]. It allows a user to distribute the execution of simulated cores across multiple nodes. A key advantage is that it uses a dynamic binary translator to directly execute the simulated code on the native machines for fast functional simulation. However, it is not a full-system simulator. Thus, it cannot simulate complex workloads that need OS support. Lastly, it is not intended to be completely cycle-accurate with a scalable synchronization mechanism (LaxP2P) based on periodic, random, point-to-point synchronization between target tiles.

ZSim is a fast, parallel microarchitectural simulator for many-core simulation [240]. Like Graphite, it uses a binary translation technique to reduce the overhead of conventional cycle-driven core models. Its key advantage over Graphite is that it implements a user-level virtualization to support complex workloads without requiring full-system simulation.

SlackSim implements bounded slack synchronization amongst cores simulated across multiple physical cores to improve simulation speed [253]. Parallel Mambo [241] is a multi-threaded implementation of an IBM's full-system simulator to accelerate the simulation of a PowerPC-based system. Unlike `dist-gem5` that supports simulation of multiple full-system nodes using multiple simulation hosts, SlackSim and Parallel Mambo can simulate a system using only one single (multi-core) simulation host.

Lastly, BigHouse is a simulation infrastructure for datacenter systems [254]. It uses a combination of queuing-theoretic and stochastic models to quickly simulate servers. Instead of application binaries, it uses empirically measured distribution of arrival and service time of tasks in the system. It is not appropriate for studies that require microarchitectural and operating system details.

## 6.6 Conclusion

In this chapter we introduced `dist-gem5`, a distributed version of `gem5`. We showed that `dist-gem5` can simulate a computer cluster with the network performance closely followed by the physical network performance. We showed that `dist-gem5` provides 83.1× speedup for simulating a 63-node server rack using 16 physical hosts over simulating all 63 nodes with one `gem5` process. Currently, `dist-gem5` is part of the official release of `gem5` and is actively maintained by the `gem5` community.

# Part IV

# Conclusion and Future Work

In this part, we conclude this thesis by discussing a promising future research line and summarizing the contributions of this thesis.

# CHAPTER 7

# CONCLUSION AND FUTURE WORK

## 7.1 Future Work

The challenge of next-generation computer systems is the "data supply" challenge. Data is generated at an unprecedented rate and stored in the datacenter storage tiers. The data is funneled through the processor pipelines within the datacenter. Ideally, we want to break the abstractions between processing elements, memory, and storage and process the data in-place without moving it to a compute node. In this section, we propose to loosens the "server" abstraction by implementing a network-attached, software-defined disaggregated memory (`NetSDM`) architecture for clusters that run a diverse set of applications, including AI/ML applications. This section, specifically focuses on AL/ML workloads as they are the predominant workloads that run in production datacenters. This proposal is built atop commodity products.

### 7.1.1 Motivation

The current data volume that ML/AI training and inference workloads are dealing with is growing at a phenomenal rate. To feed this huge data to the processing elements (*i.e.,* CPU, GPU, or accelerators), a large, high bandwidth, low latency, off-chip memory is required for each processing element. Currently, the only memory technology that satisfies such high bandwidth and low latency at a reasonable cost is DRAM. Although non-volatile memory technologies such as 3D-Xpoint [123] are attractive solutions to drive down the memory cost, they cannot replace DRAM for low-latency, ML inference workloads with tight service level objectives. For example, when running a personalized recommendation inference workload, a machine with 2400MHz DDR4 DIMMs executes `SparseLengthSum` operations up to 40% faster than a machine with 1600MHz DDR3 DIMMs [255]. This shows the importance the off-chip memory latency for personalized recommendation models. As a result, it seems inevitable to use DRAM as the primary memory technology for such servers.

With gigantic data set size and increasingly heterogeneous datacenter hardware, it is difficult, inefficient, and costly to manage the memory space of each processing element in the middleware or applications. Moreover, with diverse memory capacity requirements of different ML workloads, pre-configured memory capacity for servers can either make the processing elements to starve or leave some of the server's memory capacity stranded. Moreover, to improve the throughput, several ML models are often co-located on a single server. The workload co-location puts extra pressure on the memory capacity required for each server. However, increasing the memory capacity over the pin-limited memory channel results in signal integrity degradation and a lower DDR clock frequency. Because ML workloads are memory latency and bandwidth sensitive, such memory capacity and performance imbalance can significantly hurt the performance of ML workloads [256]. We propose to implement a disaggregated memory architecture called `NetSDM` not only to resolve the memory capacity wall for ML workloads but also to improve the scalability and TCO of production datacenters.

There is a rich literature on software and hardware support for disaggregated memory. Lim et al. [257] proposed using a memory blade for sharing memory between servers in a rack to solve the memory capacity limit and reduce memory provisioning and power cost. They use PCI-Express to connect the memory blade to the compute nodes. Using a centralized memory blade can become a bottleneck considering that all the memory accesses in a rack are forwarded to a single blade server. Furthermore, PCI-Express is not a scalable interconnect. A large body of previous works used RDMA network to implement a disaggregated memory system [258, 259, 260, 261]. All these works suffer from the PCI-Express latency bottleneck as RDMA NICs are PCIe devices. For example, on average, the hardware latency of a 4KB RDMA transfer over a 50Gbps InfiniBand link takes $\sim$4.5s, which is an order of magnitude slower than local memory access. Furthermore, RDMA's efficiency drops for small packets, and a large transfer size is required to utilize the network bandwidth. Moreover, RDMA uses a software-managed asynchronous completion notification mechanism. Handling local memory misses inside the software wastes processor cycles and is not low latency. Kwon et al. [262] used NVLINK to implement a disaggregated memory system for a GPU cluster used for ML training. Although NVLINK has much higher bandwidth and lower latency than PCI-Express, it is an expensive, near-range interconnect and is not scalable. Moreover, inference to ML models runs on CPU servers [256]. Google has taken a software approach to expand the effective memory capacity of its servers [263]. They compress cold pages within the memory and store them inside the local memory. The compressed pages can be uncompressed and used on-demand. All the previous proposals access the remote (or far) memory in page granularity, making the random accesses to the remote memory extremely inefficient and expensive. To design a disaggregate memory system that works for low-latency ML workloads, a solution should satisfy the following requirements: (R1) should be scalable to at least the size of a rack (*e.g.,* around 64 servers), (R2) have sub microsecond remote memory access latency, (R3) minimize on-demand remote memory accesses, (R4) support efficient remote memory sharing, (R5) efficiently use the shared datacenter network capacity, and (R6) minimize the cost.

## 7.2 `NetSDM` Architecture

**Processor and networking architecture**. The scope of `NetSDM` is a rack within a cluster. We assume that there are 64 servers in a rack that are interconnected using a top of rack switch. The DRAM memory capacity on each server is divided into two pools at the server's startup: "local-memory" and "lent-memory." The lent-memory is going to be aggregated across all the servers in the rack to implement a "remote-memory" pool without having a centralized memory blade. As explained earlier, a centralized memory blade creates network and memory bottlenecks. We propose to share the memory channels between local- and lent-memory space but separate the memory ranks. This is to provide memory channel parallelism for both local- and remote-memory accesses while reducing memory rank and bank conflicts. The local-memory space can be configured as a cache for the remote-memory or be part of a flat physical memory address space. It is critical to optimize the local-memory for "latency" as memory-intensive ML workloads are specifically sensitive to memory latency. Optimally, we want to have the minimum required local-memory capacity and lend the extra capacity to the remote memory space. Therefore, it is interesting to consider partitioning local- and lent-memory across different memory channels, and populate the local-memory channels with few unbuffered DIMMs (to maximize the memory latency) and fill the lent-memory channels with maximum capacity buffered DIMMs (to maximize the memory capacity). Another benefit of physically separating local- and lent-memory is that the lent-memories can be put in a low-power mode if they are not used.

For networking, we propose to use integrated Ethernet NICs. Currently, the Intel Xeon Scalable platform provides chipsets with up to four 10Gbps integrate Ethernet ports. Such integrate NIC is ideal for ultra-low latency remote memory accesses without paying PCI-Express overhead. We decide to use off-the-shelf lossy Ethernet links to minimize the networking cost and improve the scalability as lossless links have limited scalability (satisfy R2 and R6). Note that for ML workloads, it is not catastrophic to have occasional bit errors in data as they are tolerable to accuracy loss. The resiliency of ML workloads to occasional errors can significantly simplify the design, save cost, and improve the infrastructure's performance.

**Remote memory prefetcher.** The resources within a datacenter are underutilized, especially the clusters that run user-facing online services such as personalized recommendation models. The average network utilization of a datacenter is around 25% [6], and the average memory channel utilization is less than 20%. Utilizing this abundant, underutilized network and memory bandwidth, we propose to implement an aggressive, opportunistic remote memory prefetcher to boost the local-memory hit rate. The remote memory prefetcher can use conventional next-line or stride prefetching policies or use an online ML model that learns memory access patterns and sends prefetch requests based on the program's dynamic behavior. Using ML to improve architecture's performance is an active line of research.

A critical requirement of the remote memory prefetcher is not to hurt the performance of critical local and remote memory accesses. We propose a best-effort, fine-grain remote memory access protocol for remote memory prefetching. The remote memory accesses have two priority levels, high priority if it is on-demand remote memory access (*i.e.,* local memory miss), and low priority if it is a prefetch. In case of congestion, the network switch or NIC will first drop low priority packets. Small packets with different priorities also help reducing head of the line queuing delay in the network for latency-critical network requests. One challenge for fine-grain remote memory accesses is the high network packet header overhead. The minimum header size of an Ethernet frame is 14 bytes, which translates to over 20% of overhead for a 64-byte remote memory access. We propose to utilize the Reconfigurable Match-action Table (RMT) in programmable switches to implement a specialized routing protocol for remote memory accesses. For example, if there are 64 machines in a rack, each machine within a rack can be uniquely addressed with only one byte instead of six bytes for a regular MAC address. This can significantly reduce the network packet overhead. We call these customized packets "remote memory requests". The aggressive memory prefetcher can be repurposed to aggressively write-back the local dirty pages to the remote memory to prevent on-demand page evictions.

**Handling local memory misses**. To identify that a page is in remote memory or local memory, we propose to extend each page table entry with a bit that specifies if a page is remote or local, the MAC address of the remote memory machine, and the physical address of the remote memory. This is similar to the extensions needed for having unified address translation for SSDs [264]. Local-memory misses are purely handled in hardware. A remote memory request is created and sent over the integrated NIC to the remote memory machine in the case of a local-memory miss. Note that we can decide to bring the entire page to the local-memory or have a single access. This can be determined at runtime, depending on the access pattern of the application. On-demand remote memory requests cannot be best-effort and need to be sent out with the highest priority. Moreover, we propose to implement a NACK protocol between the switch and NIC ports to retry a high-priority remote memory request in case of a drop.

**Centralized controller**. We propose a centralized controller that manages the memory space of the rack, configures the programmable switch, and performs power management for un-allocated remote memories. The only time that a server needs to communicate with the centralized controller is when allocating/deallocating remote memory or need a writable page. The centralized controller is responsible for allocating the requested memory on a remote memory location and populate the server's page table with the corresponding information. `NetSDM` is built atop the existing literature and management frameworks already implemented by academia and industry.

**Memory sharing.** We propose to cache a list of sharers in the programmable switch, so page invalidations can be sent by the programmable switch as soon as it receives a request for a writable copy of a shared page. The centralized controller has all the information about sharers in the cluster. The centralized controller configures the programmable switch to cache as many sharer information as it can. In rare cases that programmable switch lacks the information, the packet is forwarded to the controller to be handled there. Note that writable shared pages are very rare in production ML workloads.

### 7.2.1 `NetSDM` Recap

As a future line of research, this section proposed building `NetSDM`, a software-defined, cacheline accessable, disaggregated memory architecture to expand the available memory capacity of datacenter servers, improve resource utilization, reduce the deployment cost, and boost the performance of AI workloads. Four key innovations distinguish `NetSDM` from the previous proposals for disaggregate memory: first, `NetSDM` distributes remote memory across all servers in the cluster to prevent a server from becoming a network or memory bottleneck. `NetSDM` also enables the software to configure the local memory size of each server. Second, `NetSDM` aggressively prefetches from remote memory to reduce the local-memory misses. This aggressive remote memory prefetching utilizes the unused capacity of the network and memory channels. Third, `NetSDM` implements a fine-grain remote memory access protocol with different priority levels and low network header overhead. The remote memory accesses are generated within the hardware and sent out using an integrated NIC. Fourth, `NetSDM` utilizes the reconfigurable match-action tables within the top-of-rack programmable switch to efficiently implement remote page sharing between servers within a rack.

## 7.3 Conclusion

Data movement is the primary performance and energy efficiency bottleneck in datacenter computing. Both within a server between CPU and DRAM and also across servers over datacenter network. Near memory processing proposals for integrating DRAM and CPU logic that has been proposed several decades ago was both premature and had technical problems. However, because of shifts in technology and market pull, these ideas become relevant to extract every bit of performance from the CMOS logic. Moreover, all the data in the datacenters are network data, where all the data that is going to be processed in a server cross the network adapter. Preprocessing of the data inside the network can be used not only for improving the application's performance but also for power and resource management. In this thesis, building atop commodity processors, DRAM products, and network devices, we architect a specialized datacenter based on in-network and near-memory computing techniques. Furthermore, this thesis introduces `dist-gem5`, a full-system even-driven parallel/distributed simulator for studying computer clusters.

# REFERENCES

[1] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2018.

[2] D. Meisner, B. T. Gold, and T. F. Wenisch, “Powernap: eliminating server idle power,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 205–216, 2009.

[3] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Improving resource efficiency at scale with Heracles,” *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 2, pp. 1–33, 2016.

[4] G. F. Pfister, “An introduction to the Infiniband architecture,” *High Performance Mass Storage and Parallel I/O*, vol. 42, pp. 617–632, 2001.

[5] B. G. Banavalikar, C. M. DeCusatis, M. Gusat, K. G. Kamble, and R. J. Recio, “Credit-based flow control in lossless Ethernet networks,” Patent WO2 014 139 368A1, 2016.

[6] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano et al., “Jupiter rising: A decade of Clos topologies and centralized control in Google’s datacenter network,” *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 183–197, 2015.

[7] “Looking back: Google’s first data center,” accessed: 06/1/2020. [Online]. Available: https://www.datacenterknowledge.com/archives/2014/02/05/looking-back-googles-first-data-center

[8] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, “Understanding PCIe performance for end host networking,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 327–341.

[9] E. Brose, “ZeroCopy: techniques, benefits and pitfalls,” *Hot topics on OS*, 2005.

[10] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: Understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 462–473.

[11] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 158–169.

[12] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[13] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *Proceedings of 27th International Symposium on Computer Architecture (ISCA)*, 2000, pp. 161–171.

[14] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 336–348.

[15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 105–117.

[16] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1999, pp. 192–201.

[17] J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system: A retrospective paper," in *IEEE 30th International Conference on Computer Design (ICCD)*, 2012, pp. 3–4.

[18] P. Ranganathan and A. Vahdat, "Plotting a course to a continued moore's law." [Online]. Available: https://www.youtube.com/watch?v=6wq6g_vi6yw&t=3373s

[19] M. Alian, A. H. Abulila, L. Jindal, D. Kim, and N. S. Kim, "NCAP: Network-driven, packet context-aware power management for client-server architecture," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 25–36.

[20] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 175–188.

[21] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim, "Application-transparent near-memory processing architecture with memory channel network," in *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 802–814.

[22] M. Alian and N. S. Kim, "NetDIMM: Low-latency near-memory network interface architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 699–711.

[23] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: Distributed simulation of computer clusters," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 153–162.

[24] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014, pp. 1–10.

[25] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, A. Greenberg, H. Kumar, C. Somesh, C. Matt, H. J. Lavier, and A. Greenberg Microsoft, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66.

[26] T. P. Morgan, "Hypercalers lead the way to the future with SmartNICs," Oct. 2019. [Online]. Available: https://www.nextplatform.com/2019/10/31/hypercalers-lead-the-way-to-the-future-with-smartnics/

[27] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice, “The TURBO diaries: Application-controlled frequency scaling explained,” in *USENIX Annual Technical Conference (USENIX ATC)*, 2014, pp. 193–204.

[28] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, p. 74, 2 2013.

[29] V. Pallipadi and A. Starikovskiy, “The ondemand governor,” in *Proceedings of the Linux Symposium*, vol. 2, no. 00216, 2006, pp. 215–230.

[30] V. Pallipadi, S. Li, and A. Belay, “cpuidle: Do nothing, efficiently,” in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 119–125.

[31] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, “Power management of online data-intensive services,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 319–330, 6 2011.

[32] Y. Liu, S. C. Draper, and N. S. Kim, “SleepScale: Runtime joint speed scaling and sleep states management for power efficient data centers,” in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 313–324.

[33] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 301–312, 10 2014.

[34] Wikipedia, “Advanced configuration and power interface.” [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface

[35] D. Brodowski and N. Golde, “CPU frequency and voltage scaling code in the Linux kernel,” 2015.

[36] “Voltage regulator module (VRM) and enterprise voltage regulator-down,” Tech. Rep., 2009. [Online]. Available: http://www.intel.com/technology/security/.

[37] “Interrupt moderation using Intel GbE controllers,” Tech. Rep., 2007. [Online]. Available: http://www.intel.com/products/processor_number

[38] M. Flajslik and M. Rosenblum, “Network interface design for low latency request-response protocols,” in *USENIX Annual Technical Conference (ATC)*, 2013, pp. 333–346.

[39] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[40] P. Mochel, "The sysfs filesystem," in *Linux Symposium*, 2005, p. 313.

[41] V. Spiliopoulos, A. Bagdia, A. Hansson, P. Aldworth, and S. Kaxiras, "Introducing DVFS-management in a full-system simulator," in *IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, 2013, pp. 535–545.

[42] D. Brodowski, "Linux CPUFreq - CPU frequency and voltage scaling code in the Linux kernel — the Linux kernel documentation," Tech. Rep. [Online]. Available: https://www.kernel.org/doc/html/latest/cpu-freq/index.html

[43] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 37–48, 2012.

[44] "ab - Apache HTTP server benchmarking tool." [Online]. Available: https://httpd.apache.org/docs/2.2/programs/ab.html

[45] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *43rd Proceedings of International Symposium on Computer Architecture (ISCA)*, 8 2016, pp. 456–468.

[46] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. ACM, 2010, pp. 267–280.

[47] S. Li, J. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, Apr 2013.

[48] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 271–282.

[49] S. Dawson-Haggerty, A. Krioukov, and D. E. Culler, "Power optimization-a reality check," EECS Department, University of California, Berkeley, Tech. Rep., 2009.

[50] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "TimeTrader: Exploiting latency tail to save datacenter energy for online search," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, vol. 05-09-December-2015, 12 2015, pp. 585–597.

[51] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Waikiki: ACM, 12 2015. [Online]. Available: http://dx.doi.org/10.1145/2830772.2830797

[52] F. D. Rossi, M. Da, S. Conterato, T. Ferreto, and C. A. F. De Rose, "Evaluating the trade-off between DVFS energy-savings and virtual networks performance," *Proceedings of the International Conference on Networking (ICN)*, 2014.

[53] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 1223–1231.

[54] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 1223–1231.

[55] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 571–582.

[56] 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)*, 2014, pp. 583–598.

[57] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training deep networks in Spark," *arXiv preprint arXiv:1511.06051*, 2015.

[58] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: Near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.

[59] 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," in *arXiv:1706.02677 [cs.CV]*, 2017.

[60] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.

[61] Y. You, Z. Zhang, C. Hsieh, and J. Demmel, "100-epoch ImageNet training with AlexNet in 24 minutes," in *arXiv:1709.05011v10 [cs.CV]*, 2018.

[62] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[63] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 367–381.

[64] D. S. Banerjee, K. Hamidouche, and D. K. Panda, "Re-designing CNTK deep learning framework on modern GPU enabled clusters," in *CloudCom*, 2016.

[65] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters," in *PPoPP*, 2017.

[66] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[67] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[68] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[69] I. Kokkinos, "UberNet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory," in *CVPR*, 2017.

[70] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.

[71] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[72] S. Sardashti, A. Arelakis, and P. Stenstrm, *A Primer on Compression in the Memory Hierarchy.* Morgan & Claypool Publishers, 2015.

[73] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[74] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary gradients to reduce communication in distributed deep learning," *CoRR*, vol. abs/1705.07878, 2017.

[75] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient sgd via gradient quantization and encoding," in *NIPS*, 2017.

[76] N. Dryden, T. Moon, S. A. Jacobs, and B. Van Essen, "Communication quantization for data-parallel training of deep neural networks," in *2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, 2016, pp. 1–8.

[77] N. Strom, "Scalable distributed DNN training using commodity GPU cloud computing," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[78] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *45th IEEE International Conference on Parallel Processing (ICPP)*, 2016, pp. 242–247.

[79] "Snappy compression." [Online]. Available: https://github.com/google/snappy

[80] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 730–739.

[81] K. Ramakrishnan, "The addition of explicit congestion notification (ECN) to IP," Tech. Rep., 2007. [Online]. Available: https://tools.ietf.org/html/rfc3168

[82] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein et al., "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[83] Apache Incubator, "Handwritten digit recognition, https://mxnet.incubator.apache.org/tutorials/python/mnist.html," 2017.

[84] A. Damien, "Tensorflow examples," 2017. [Online]. Available: https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/multilayer_perceptron.py

[85] Google INC, "Keras examples," 2017. [Online]. Available: https://github.com/keras-team/keras/blob/master/examples/mnist_mlp.py

[86] K. Sopyla, "Tensorflow MNIST convolutional network tutorial," 2017. [Online]. Available: https://github.com/ksopyla/tensorflow-mnist-convnets

[87] Google INC, "Tensorflow model zoo," 2017. [Online]. Available: https://github.com/tensorflow/models

[88] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[89] Nvidia Corporation, "Nvidia CUDA C programming guide," 2010.

[90] Intel Corporation, "Intel math kernel library," 2018. [Online]. Available: https://software.intel.com/en-us/mkl

[91] OpenMPI Community, "OpenMPI: A high performance message passing library," 2017. [Online]. Available: https://www.open-mpi.org/

[92] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[93] Samsung Corporation, "Samsung DDR4," 2017. [Online]. Available: http://www.samsung.com/semiconductor/global/file/product/DDR4-Product-guide-May15.pdf

[94] Netgear Corporation, "ProSafe xs712t switch," 2017. [Online]. Available: https://www.netgear.com/support/product/xs712t.aspx

[95] Intel Corporation, "Intel x540," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x540-t2-brief.html

[96] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[97] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.

[98] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *ISCA*, 2017.

[99] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.

[100] P. Judd, A. Delmas, S. Sharify, and A. Moshovos, "Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing," in *ISCA*, 2016.

[101] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.

[102] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.

[103] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[104] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer." in *MICRO*, 2014.

[105] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Misra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *MICRO*, Oct. 2016.

[106] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *HPCA*, 2017.

[107] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerator," in *MICRO*, 2016.

[108] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO*, 2017.

[109] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.

[110] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating persistent neural networks at datacenter scale," in *HotChips*, 2017.

[111] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen, "Machine learning on FPGAs to face the IoT revolution," in *Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 894–901.

[112] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[113] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA*, 2017.

[114] Q. Wang, Y. Li, and P. Li, "Liquid state machine based pattern recognition on FPGA with firing-activity dependent power gating and approximate computing," in *ISCAS*, 2016.

[115] Q. Wang, Y. Li, B. Shao, S. Dey, and P. Li, "Energy efficient parallel neuromorphic architectures with approximate arithmetic on FPGA," *Neurocomputing*, vol. 221, pp. 146–158, 2017.

[116] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016.

[117] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.

[118] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv:1606.06160 [cs]*, 2016.

[119] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings. 25th Annual International Symposium on Computer Architecture (ISCA)*, 1998, pp. 192–203.

[120] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava et al., "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC)*, 1999, pp. 57–57.

[121] F. Devaux, "The true processing in memory accelerator," in *Hot Chips*, 2019. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8875680

[122] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing SoC accelerators for memcached," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013, pp. 36–47.

[123] Micron, "3D XPoint$^{\text{TM}}$ technology." [Online]. Available: https://www.micron.com/products/advanced-solutions/3d-xpoint-technology

[124] 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 *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[125] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow, J. H. Ahn, and N. S. Kim, "Near-DRAM acceleration with single-ISA heterogeneous processing in standard memory modules," *IEEE Micro*, vol. 36, no. 1, pp. 24–34, 2016.

[126] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 113–124.

[127] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, 2014, pp. 85–98.

[128] S. Lloyd and M. Gokhale, "Near memory key/value lookup acceleration," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 26–33.

[129] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, "Integrated 3D-stacked server designs for increasing physical density of key-value stores," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[130] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 273–287.

[131] Y. K. Rupesh, P. Behnam, G. R. Pandla, M. Miryala, and M. N. Bojnordi, "Accelerating k-medians clustering using a novel 4t-4r RRAM cell," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[132] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos et al., "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17–1, 2015.

[133] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, "Intelligent RAM (IRAM): The industrial setting, applications, and architectures," in *ICCD*, Oct 1997.

[134] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *26th IEEE symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[135] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010, pp. 10–10.

[136] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, "User's guide to MPICH, a portable implementation of MPI," *Argonne National Laboratory*, vol. 9700, pp. 60 439–4801, 1995.

[137] Qualcomm, "Snapdragon 835 mobile platform," 2016. [Online]. Available: https://www.qualcomm.com/products/snapdragon/processors/835

[138] JEDEC Standard, "DDR4 SDRAM registered DIMM design specification," *JESD21-C*, 2016.

[139] JEDEC Standard, "DDR4 SDRAM load reduced DIMM design specification," *JESD21-C*, 2016.

[140] P. J. Meaney, L. D. Curley, G. D. Gilda, M. R. Hodges, D. J. Buerkle, R. D. Siegl, and R. K. Dong, "The IBM z13 memory subsystem for big data," *IBM Journal of Research and Development*, vol. 59, no. 4/5, pp. 4:1–4:11, July 2015.

[141] B. Jacob and T. Mudge, "Virtual memory in contemporary microprocessors," *IEEE Micro*, vol. 18, no. 4, pp. 60–75, 1998.

[142] E. Dumazet, "Busy polling: Past, present, future," *NetDev 2.1*, 2017.

[143] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Comparing implementations of near-data computing with in-memory MapReduce workloads," *Micro, IEEE*, vol. 34, no. 4, Jul. 2014.

[144] T. R. Halfhill, "Power8 hits the merchant market: Memory bandwidth helps IBM server processor ace big benchmarks." [Online]. Available: https://www-03.ibm.com/systems/power/advantages/smartpaper/memory-bandwidth.html

[145] "The Exynos 5433 SoC," Tech. Rep. [Online]. Available: http://techreport.com/review/27539/samsung-galaxy-note-4-with-the-exynos-5433-processor/2

[146] A. Wei, "Qualcomm Snapdragon 835 first to 10 nm," 2017. [Online]. Available: http://www.techinsights.com/about-techinsights/overview/blog/qualcomm-snapdragon-835-first-to-10-nm/

[147] Nvidia, “Nvidia Tegra x1,” 2015. [Online]. Available: https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf

[148] Micron, “NVDIMM,” 2016. [Online]. Available: https://www.micron.com/products/dram-modules/nvdimm/

[149] J. S. Turner and D. E. Taylor, “Diversifying the Internet,” in *Global Telecommunications Conference, 2005. GLOBECOM’05. IEEE*, vol. 2. IEEE, 2005, pp. 6–pp.

[150] T. Anderson, L. Peterson, S. Shenker, and J. Turner, “Overcoming the Internet impasse through virtualization,” *Computer*, vol. 38, no. 4, pp. 34–41, 2005.

[151] “IPv4 standard,” accessed: 2018-03-25. [Online]. Available: https://en.wikipedia.org/wiki/IPv4

[152] “Ethernet frame,” accessed: 2018-03-25. [Online]. Available: https://en.wikipedia.org/wiki/Ethernet_frame

[153] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder, “Offload of TCP segmentation to a smart adapter,” Patent US5 937 169A, 1999.

[154] “TCP frame,” accessed: 2018-03-25. [Online]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[155] “JEDEC Standard: DDR4 SDRAM,” 2012.

[156] B. Sukhwani, T. Roewer, C. L. Haymes, K.-H. Kim, A. J. McPadden, D. M. Dreps, D. Sanner, J. Van Lunteren, and S. Asaad, “Contutto: A novel FPGA-based prototyping platform enabling innovation in the memory subsystem of a server class processor,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 15–26.

[157] Intel, “Nios II processor,” 2017. [Online]. Available: https://www.altera.com/products/processors/overview.html

[158] Intel, “Avalon interface specifications,” 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf

[159] M. Alian, D. Kim, and N. S. Kim, “pd-gem5: Simulation infrastructure for parallel/distributed computer systems,” *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 41–44, 2016.

[160] 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," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[161] "iperf: The ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: https://iperf.fr/

[162] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber et al., "The NAS parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[163] "Coral benchmark codes." [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[164] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from Internet services," 2 2014, pp. 488–499.

[165] R. Azimi, T. Fox, and S. Reda, "Understanding the role of GPGPU-accelerated SoC-based ARM clusters," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 333–343.

[166] L. Grossman, "Large receive offload implementation in Neterion 10GbE Ethernet driver," in *Linux Symposium*, 2005, p. 195.

[167] C. B. Melzer, J. Rosen, R. O'Gorman, P. A. Wood, M. C. Drummond, and D. Hiller, "IP checksum offload," Patent US5 898 713A, 1999.

[168] "DPDK: Data plane development kit." [Online]. Available: http://dpdk.org/

[169] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 489–502.

[170] M. Chan and D. R. Cheriton, "Improving server application performance via pure TCP ACK receive optimization," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC)*, 2013, pp. 359–364.

[171] A. Andreyev, "Introducing data center fabric, the next-generation Facebook data center network," 2014. [Online]. Available: {https://code.facebook.com/posts/360346274145943/}

[172] M. F. Deering, S. A. Schlapp, and M. G. Lavelle, "FBRAM: A new form of memory optimized for 3D graphics," in *SIGGRAPH*, Jul. 1994.

[173] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the DIVA processing-in-memory chip," in *ICS*, Jun. 2002.

[174] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational RAM: A memory-SIMD hybrid and its application to DSP," in *CICC*, May 1992.

[175] M. L. Chu, N. Jayasena, D. P. Jang, and M. Ignatowski, "High-level programming model abstractions for processing in memory," in *Workshop on Near-Data Processing*, Dec. 2013.

[176] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, Jun. 2008.

[177] J. T. Pawlowski, "Hybrid memory cube," in *Hot Chips*, Aug 2011.

[178] A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA*, Feb. 2015.

[179] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3DIC*, Oct. 2013.

[180] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468.

[181] 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 *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 190–200.

[182] A. J. Awan, M. Ohara, E. Ayguadé, K. Ishizaki, M. Brorsson, and V. Vlassov, "Identifying the potential of near data processing for Apache Spark," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017, pp. 60–67.

[183] C. D. Kersey, H. Kim, and S. Yalamanchili, "Lightweight SIMT core designs for intelligent 3D stacked DRAM," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017, pp. 49–59.

[184] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, Jan 2015.

[185] Intel, "An introduction to the Intel QuickPath interconnect." [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf

[186] Intel, "IvyTown Xeon + FPGA: The HARP program, CPU+FPGA - OpenCL based high level synthesis for CPU+FPGA coherent systems," 2016. [Online]. Available: https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf

[187] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, 1988, pp. 314–329.

[188] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese, "Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 255–266.

[189] "Wall street's quest to process data at the speed of light," accessed: 2018-12-3. [Online]. Available: https://www.informationweek.com/wall-streets-quest-to-process-data-at-the-speed-of-light/d/d-id/1054287

[190] W.-F. Wang, J.-Y. Wang, and J.-J. Li, "Study on enhanced strategies for TCP/IP offload engines," in *Proceedings of 11th International Conference on Parallel and Distributed Systems*, vol. 1, 2005, pp. 398–404.

[191] W.-c. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda, "Performance characterization of a 10-gigabit Ethernet TOE," in *Proceedings of the 13th Symposium on High Performance Interconnects*, 2005, pp. 58–63.

[192] H. Jang, S.-H. Chung, D. K. Kim, and Y.-S. Lee, "An efficient architecture for a TCP offload engine based on hardware/software co-design," *Journal of Information Science and Engineering*, vol. 27, no. 2, pp. 493–509, 2011.

[193] "Large send offload," accessed: 2018-11-28. [Online]. Available: https://en.wikipedia.org/wiki/Large_send_offload

[194] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 63–74, 2011.

[195] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* USENIX Association, 2012, pp. 19–19.

[196] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM, 2017, pp. 29–42.

[197] "Low-latency Ethernet device polling," accessed: 2018-11-28. [Online]. Available: https://lwn.net/Articles/551284/

[198] "Open fast path," accessed: 2018-11-28. [Online]. Available: https://openfastpath.org/

[199] M. Beck and M. Kagan, "Performance evaluation of the RDMA over Ethernet (RoCE) standard in enterprise data centers infrastructure," in *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching.* International Teletraffic Congress, 2011, pp. 9–15.

[200] "ConnectX-2 EN with RDMA over Ethernet (RoCE)," accessed: 2018-11-28. [Online]. Available: http://www.mellanox.com/related-docs/prod_software/ConnectX-2_RDMA_RoCE.pdf

[201] R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture.* Addison-Wesley Professional, 2004.

[202] G. Liao, X. Znu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 255–265.

[203] S. Larsen and B. Lee, "Platform IO DMA transaction acceleration," in *International Conference on Supercomputing (ICS) Workshop on Characterizing Applications for Heterogeneous Exascale Systems (CACHES)*, 2011.

[204] A. K. M. Kaminsky and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference*, 2016, p. 437.

[205] W. de Bruijn and E. Dumazet, "sendmsg copy avoidance with msg_-zerocopy."

[206] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated network interfaces for high-bandwidth TCP/IP," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 315–324, 2006.

[207] H.-y. Kim, V. S. Pai, and S. Rixner, "Increasing web server throughput with network interface data caching," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 239–250, 2002.

[208] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura, "Caching memcached at reconfigurable network interface," in *24th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6.

[209] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with FlexNIC," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, 2016, pp. 67–81.

[210] J. Song and J. Alves-Foss, "Performance review of zero copy techniques," *International Journal of Computer Science and Security (IJCSS)*, vol. 6, no. 4, p. 256, 2012.

[211] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 50–59.

[212] "Intel data direct I/O technology (Intel DDIO): A primer," Tech. Rep. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html

[213] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling slos in network function virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[214] D. Tang, Y. Bao, W. Hu, and M. Chen, "DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[215] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, "Architectural breakdown of end-to-end latency in a TCP/IP network," *International Journal of Parallel Programming*, vol. 37, no. 6, pp. 556–571, 2009.

[216] "A prelude to nonvolatile DIMM technology. Future of NVDIMM-P," accessed: 06/30/2019. [Online]. Available: https://gigglehd.com/gg/hard/1893698

[217] “Single- and Multi-channel memory modes,” accessed: 06/30/2019. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000005657/boards-and-kits.html

[218] Wikipedia, “High memory.” [Online]. Available: https://en.wikipedia.org/wiki/High_memory

[219] S. Seth and M. A. Venkatesulu, *TCP/IP Architecture, Design, and Implementation in Linux*. John Wiley & Sons, 2009, vol. 68.

[220] D. Firestone, “VFP: A virtual switch platform for host SDN in the public cloud,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 315–328.

[221] Intel, *Intel memory latency checker v3.5*, Nov. 2018. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[222] “Overcoming system memory challenges with persistent memory and NVDIMM-P,” accessed: 2018-12-5. [Online]. Available: https://www.jedec.org/sites/default/files/Bill_Gervasi.pdf

[223] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization,” in *MICRO*, 2013.

[224] “Add 4GB DMA32 zone,” accessed: 2018-11-20. [Online]. Available: https://lwn.net/Articles/152337/

[225] Micron, *Micron DDR4 SDRAM datasheet*. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[226] “Diablo conjures up hell of a DIMM: 128GB NAND pretend-RAM summoned,” accessed: 06/30/2019. [Online]. Available: https://www.theregister.co.uk/2016/07/22/diablos_devilishly_clever_nandbased_pretend_dram_dimms_now_shipping/

[227] Intel, “Intel Ethernet controller X710/XXV710/XL710 datasheet,” Feb 2018. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf

[228] M. Alian, K. P. Srinivasan, and N. S. Kim, “Simulating PCI-Express interconnect for future system exploration,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 168–178.

[229] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, “Simulating DRAM controllers for future system architecture exploration,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 201–210.

[230] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15, 2015, pp. 123–137.

[231] James Hongyi Zeng, “Data sharing on traffic pattern inside Facebook datacenter network,” accessed: 03/30/2019. [Online]. Available: https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/

[232] “Arista 7130 connect series ultra-low latency switches,” accessed: 03/30/2019. [Online]. Available: https://www.arista.com/en/products/7130-series

[233] H.-y. Kim, S. Rixner, and V. S. Pai, “Network interface data caching,” *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1394–1408, 2005.

[234] R. Minnich, D. Burns, and F. Hady, “The memory-integrated network interface,” *IEEE Micro*, vol. 15, no. 1, pp. 11–19, 1995.

[235] N. Tanabe, J. Yamamoto, H. Nishi, T. Kudoh, Y. Hamada, H. Nakajo, and H. Amano, “MEMOnet: Network interface plugged into a memory slot,” in *Proceedings IEEE International Conference on Cluster Computing (CLUSTER)*, 2000, pp. 17–26.

[236] N. Tanabe, H. Nakajyo, H. Amano, M. Yoshimi, A. Kitamura, and T. Miyashiro, “DIMMnet-2: A reconfigurable board connected into a memory slot,” in *2006 International Conference on Field Programmable Logic and Applications*. IEEE, 2006, pp. 1–4.

[237] “GPU models - gem5.” [Online]. Available: http://www.m5sim.org/GPU_Models

[238] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, “Cotson: infrastructure for full system simulation,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

[239] A. Patel, F. Afram, and K. Ghose, “Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors,” in *1st International Qemu Users’ Forum*, 2011, pp. 29–30.

[240] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 475–486, 2013.

[241] K. Wang, Y. Zhang, H. Wang, and X. Shen, "Parallelization of IBM mambo system simulator in functional modes," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 71–76, 2008.

[242] Y. Kanada, "Ethernet switch/terminal simulators for novices to learn computer networks," in *2015 International Conference on Information Networking (ICOIN)*. IEEE, 2015, pp. 487–492.

[243] S. L. Gardner, J. S. Hiscock, and M. Yuen, "Advanced Ethernet auto negotiation," Patent US6 580 697B1, 2003.

[244] "memcached," accessed: 2018-11-12. [Online]. Available: https://memcached.org/

[245] D. Mosberger and T. Jin, "Httperf: A tool for measuring web server performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[246] B. Huang, M. Bauer, and M. Katchabaw, "Hpcbench: A Linux-based network benchmark for high performance networks," in *19th IEEE International Symposium on High performance Computing Systems and applications (HPCS)*, 2005, pp. 65–71.

[247] R. Jones et al., "Netperf: a network performance benchmark," *Information Networks Division, Hewlett-Packard Company*, 1996.

[248] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 4 2014, pp. 1–14.

[249] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: Virtual prototyping of parallel computers," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993, pp. 48–60.

[250] M. Hsieh, K. Pedretti, J. Meng, A. Coskun, M. Levenhagen, and A. Rodrigues, "SST + gem5 = a scalable simulation infrastructure for high performance computing," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, 2012, pp. 196–201.

[251] R. Bedichek, “SimNow: Fast platform simulation purely in software,” in *Hot Chips*, vol. 16, 2004, pp. 48–60.

[252] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, “Graphite: A distributed parallel simulator for multicores,” in *The Sixteenth IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[253] J. Chen, M. Annavaram, and M. Dubois, “SlackSim: a platform for parallel simulations of CMPs on CMPs,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 20–29, 2009.

[254] D. Meisner, J. Wu, and T. F. Wenisch, “BigHouse: A simulation infrastructure for data center systems,” in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2012, pp. 35–45.

[255] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia et al., “The architectural implications of Facebook’s DNN-based personalized recommendation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 488–501.

[256] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro et al., “Applied machine learning at Facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[257] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, “Disaggregated memory for expansion and sharing in blade servers,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 267–278, 2009.

[258] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, “FaRM: Fast remote memory,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414.

[259] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, “Efficient memory disaggregation with Infiniswap,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 649–667.

[260] H. A. Maruf and M. Chowdhury, “Effectively prefetching remote memory with leap,” *arXiv preprint arXiv:1911.09829*, 2019.

[261] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, “LegoOS: A disseminated, distributed OS for hardware resource disaggregation,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 69–87.

[262] Y. Kwon and M. Rhu, “A disaggregated memory system for deep learning,” *IEEE Micro*, vol. 39, no. 5, pp. 82–90, 2019.

[263] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid et al., “Software-defined far memory in warehouse-scale computers,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 317–330.

[264] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, “Unified address translation for memory-mapped SSDs with FlashMap,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 580–591.