

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Towards Heterogeneous Systems: A Full-Stack Approach

Permalink

<https://escholarship.org/uc/item/7v21p5fx>

Author

Wang, Zixuan

Publication Date

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards Heterogeneous Systems: A Full-Stack Approach

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

in

Computer Science

by

Zixuan Wang

Committee in charge:

Professor Jishen Zhao, Chair
Professor Farinaz Koushanfar
Professor Steven Swanson
Professor Dean Tullsen
Professor Geoffrey Voelker

2024

Copyright
Zixuan Wang, 2024
All rights reserved.

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

University of California San Diego

2024

DEDICATION

To my parents, whose wholehearted support has been my strength.

To my love, who has shared life's greatest joys with me.

To my cat, who was by my side during my loneliest times.

And to my younger self, who bravely chose to go through this challenging
journey.

EPIGRAPH

十年磨一剑
一剑惊江湖

*A decade to hone a sword,
one stroke to stun the world.*

The first sentence originates from a poem by the Tang Dynasty poet Jia Dao, and the second is from myself.

This has been my life motto since high school, and it happened to take me ten years—four in undergraduate school and six in Ph.D.—to polish my skills in computer science.

TABLE OF CONTENTS

Dissertation Approval Page	iii
Dedication	iv
Epigraph	v
Table of Contents	vi
List of Figures	ix
List of Tables	xii
Acknowledgements	xiii
Vita	xviii
Abstract of the Dissertation	xix
Chapter 1	
Introduction	1
1.1 Background and Motivation	1
1.2 Contribution	3
1.2.1 Characterizing and Modeling Heterogeneous Memory Archi- tecture	3
1.2.2 Building Scalable Systems on Heterogeneous Architecture .	4
1.2.3 Attacking and Securing Heterogeneous Systems	5
1.2.4 Programming for the Future Heterogeneous Systems	5
Chapter 2	
Characterizing and Modeling Heterogeneous Memory Architecture	7
2.1 Introduction	7
2.2 Background and Motivation	10
2.2.1 Optane DIMM-based Server Systems	10
2.2.2 Performance Discrepancy	12
2.2.3 Inefficient NVRAM Simulation Models	12
2.2.4 Insufficient Profiling Tools	13
2.3 LENS: Low-level NVRAM Profiler	14
2.3.1 LENS Framework	14
2.3.2 Optane DIMM System Characterization	19
2.3.3 On-DIMM Buffer Analysis	22
2.3.4 Policy Analysis	24
2.4 VANS: Validated NVRAM Simulator	25
2.4.1 Simulation Model Design	26
2.4.2 DRAM Model Verification	27

	2.4.3	Validation with Microbenchmarks	28
	2.4.4	Validation with SPEC CPU Benchmarks	31
	2.4.5	Discussion	33
2.5		Cloud Workload Optimizations	34
	2.5.1	Performance Inefficiency	34
	2.5.2	In Memory Pre-Translation	35
	2.5.3	Lazy cache	38
	2.5.4	Evaluation	39
2.6		Related Work	40
2.7		Conclusion	41
2.8		Acknowledgements	42
Chapter 3		Building Scalable Systems on Heterogeneous Architecture	43
	3.1	Introduction	43
	3.2	Background and Motivation	46
	3.2.1	DL Training Parallelism Schemes	46
	3.2.2	Communication Issues in DL Training	48
	3.2.3	Opportunities With Disaggregated Memory Systems	51
	3.3	COARSE Design	53
	3.3.1	System Deployment	53
	3.3.2	A Naïve Design	54
	3.3.3	A Disaggregated Design	55
	3.3.4	Decentralized Parameter Synchronization	55
	3.3.5	Tensor Routing and Partitioning	57
	3.3.6	Dual Parameter Synchronization	61
	3.4	Implementation	63
	3.4.1	Sync Cores	63
	3.4.2	COARSE	65
	3.4.3	Disaggregated Memory Prototype	67
	3.5	Evaluation	68
	3.5.1	Experiment Setup	68
	3.5.2	CCI Prototype Performance	69
	3.5.3	Tensor Routing	70
	3.5.4	Training Speedup	71
	3.5.5	Communication Time	74
	3.6	Related Work	74
	3.7	Conclusion	76
	3.8	Acknowledgements	77
Chapter 4		Attacking and Securing Heterogeneous Systems	78
	4.1	Introduction	78
	4.2	Background	81
	4.2.1	NVRAM	81

4.2.2	Microarchitectural Side Channels	84
4.3	NVLeak Reverse-Engineering	85
4.3.1	NVRAM System Configuration	85
4.3.2	Recovering Details of NVCache and WPQ	86
4.3.3	Recovering Wear-Leveling Policy	91
4.3.4	Summary of Findings	96
4.4	NVLeak Covert Channels	96
4.4.1	Cross-VM NVCache Channel	96
4.4.2	Data-Migration-Based Channel	99
4.5	NVLeak Side-Channel Attacks	101
4.5.1	Database Operation Leakage	102
4.5.2	PMDK Key-Value Store Leakage	105
4.5.3	Code Execution Path Leakage	106
4.6	Mitigations	112
4.7	Discussion	115
4.8	Related Work	115
4.9	Conclusion	117
4.10	Acknowledgements	117
4.A	Additional Reverse Engineering Results	117
4.A.1	Additional Results on Server A	117
4.A.2	Additional Results on Server B	118
Chapter 5	Programming for the Future Heterogeneous Systems	120
5.1	Introduction	120
5.2	Background and Motivation	123
5.2.1	Multi-Threading Programming	123
5.2.2	Heterogeneous Systems	124
5.2.3	Compute Express Link	125
5.2.4	WebAssembly System Interface	126
5.3	Design	126
5.3.1	Compile High-Level Code	126
5.3.2	Runtime System	127
5.3.3	CXL Interconnection	128
5.4	Evaluation	129
5.4.1	CXL Performance	129
5.4.2	WASI Performance	130
5.5	Future Directions	131
5.6	Conclusion	131
5.7	Acknowledgements	132
Chapter 6	Conclusion	133
Bibliography		135

LIST OF FIGURES

Figure 2.1:	Comparison between PMEP [19] emulation and Optane DIMM-based real machine measurement. <i>Store-clwb</i> are writes followed by <i>clwb</i> cache write-back instructions [20]; <i>store-nt</i> are non-temporal writes [20].	8
Figure 2.2:	Optane DIMM-based system configurations.	11
Figure 2.3:	Comparison between memory simulators and Optane DIMM system profiling.	13
Figure 2.4:	LENS probes and Optane DIMM parameters. Red numbers are obtained from Intel documents; blue numbers are characterized by LENS.	21
Figure 2.5:	LENS buffer prober tests on Optane DIMM. The curves show average results of 500 runs; shaded areas represent error envelopes. In (c) <i>RaW</i> is read-after-write, <i>R+W</i> is the sum of load and store latency in (a).	21
Figure 2.6:	Read(a) and write(b) amplification scores in our buffer prober test.	22
Figure 2.7:	LENS policy prober tests on Optane DIMM.	24
Figure 2.8:	VANS overview.	24
Figure 2.9:	VANS performance validation with microbenchmarks: (a-c) pointer chasing and (d) overwrite. (e) VANS overall accuracy on load/store latency and bandwidth.	28
Figure 2.10:	Sensitivity study of memory configurations.	29
Figure 2.11:	VANS performance validation with SPEC CPU 2006 and SPEC CPU 2017.	30
Figure 2.12:	Redis and YCSB profiling. Each metric value is normalized to the corresponding <i>Rest</i> case. “Top10” stands for the ten most frequently written cache lines.	35
Figure 2.13:	Customized Lazy cache (a) for write intensive workloads and Pre-translation (a–c) for pointer chasing read intensive workloads. Baseline in (d–e) is measured without any optimization mechanism.	36
Figure 3.1:	Communication schemes in distributed DL training.	47
Figure 3.2:	Communication overhead in ML training.	49
Figure 3.3:	CCI bandwidth with CPU load/store instruction (<i>CCI</i>), GPU indirect copy through CPU (<i>GPU Indirect</i>), and GPU direct peer-to-peer copy (<i>GPU Direct</i>). Read and write bandwidth are normalized to the <i>CCI</i> bandwidth, respectively.	50
Figure 3.4:	System deployment of a single server node.	52
Figure 3.5:	DENSE architecture.	54
Figure 3.6:	COARSE architecture.	56
Figure 3.7:	Decentralized parameter synchronization.	58
Figure 3.8:	PCIe device-to-device bidirectional bandwidth on (a) V100 from AWS and (b) P100 from SDSC.	59
Figure 3.9:	Tensor partitioning and Dual synchronization.	60
Figure 3.10:	Deadlock in first-come-first-serve synchronization. $T_{i,j}$ represents tensor i from client j	63
Figure 3.11:	Parameter synchronization with sync cores (denoted as <i>SC</i> in figure b and c).	63

Figure 3.12:	Disaggregated memory prototype with two FPGAs.	66
Figure 3.13:	CCI bandwidth with CPU load/store instructions (<i>CCI</i>), GPU indirect copy through CPU (<i>GPU Indirect</i>), and GPU direct peer-to-peer copy (<i>GPU Direct</i>). Numbers are normalized to the CCI bandwidth with 512MB access size.	70
Figure 3.14:	PCIe DMA bandwidth from host CPU to FPGA.	70
Figure 3.15:	Communication performance from one client to its local and remote proxies. The local proxy shares the PCIe switch with the client. Numbers are normalized to local proxy performance.	71
Figure 3.16:	DL training on various machines.	72
Figure 3.17:	Blocked communication time in DL training. In (a-d) numbers are normalized to the CCI case; in (e-f) numbers are normalized to the AllReduce case. The CCI bar is ignored for better comparison between AllReduce and COARSE.	73
Figure 4.1:	Memory hierarchy equipped with Optane DIMMs.	82
Figure 4.2:	Pointer chasing microbenchmark results	87
Figure 4.3:	Detecting NVCache set structures with a strided pointer chasing microbenchmark.	88
Figure 4.4:	Pointer chasing microbenchmark results.	89
Figure 4.5:	Overwrite microbenchmark results. (b) shows a sample of results from Figure 4.6.	91
Figure 4.6:	Overwrite 99.99 percentile (P99.99) latencies under various delay duration and delay injection interval settings.	93
Figure 4.7:	Data migration robustness when using various memory regions. The y-axis shows the number of 256 B writes it requires to trigger one data migration. Each bar shows the distribution of write iterations.	94
Figure 4.8:	Overview of the NVRAM findings. <i>Reproduced</i> results are from LENS, <i>corrected</i> results are from NVLeak and correct observations from LENS, and <i>new</i> results are from NVLeak and not discovered by LENS.	95
Figure 4.9:	Cross-VM covert channel. (c) is a receiver signal from (b) that achieves 11 kbps bandwidth with 3.5% error rate. This channel uses 14 blocks, 1 MiB stride size and 16 repeat rounds. Y-axis shows the average per-block latency.	97
Figure 4.10:	Filesystem inode-based covert channel on NVRAM and DRAM.	99
Figure 4.11:	Evaluate SQL operations.	102
Figure 4.12:	Access patterns of database operations. (a)-(h) each shows the access pattern for a database operation: the bottom figure shows the latencies in a concentrated range of iterations and sets, and the top figure shows the latency distributions.	109
Figure 4.13:	Memory access pattern of SQLite ranged query.	110
Figure 4.14:	PMDK key-value store's memory access pattern when updating different key-value pairs.	111
Figure 4.15:	Code execution leakage and wolfSSL analysis.	111

Figure 4.16:	NVLeak mitigations applied to (a-b) PMDK key-value store and (c) a PMDK benchmark suite.	113
Figure 4.17:	Pointer chasing latency under various stride sizes.	118
Figure 4.18:	Reverse engineering results on Server B (Table 4.1).	119
Figure 5.1:	Heterogeneous programming is generally more complex than multi-threading programming due to the need of using external libraries, system drivers, and compiler toolchains.	121
Figure 5.2:	Heterogeneous system stack with conventional programming techniques, where workloads have to use multiple libraries and compilers to utilize underlying heterogeneous architectures. Such programming complexity limits the system’s scalability of adding new architectures.	124
Figure 5.3:	Heterogeneous system programming flow with CodeFlow.	125
Figure 5.4:	CodeFlow runtime performance compared to running native x86 code, using pointer-chasing read.	130

LIST OF TABLES

Table 2.1:	Comparison of profiling tools.	11
Table 2.2:	LENS overview.	15
Table 2.3:	Server hardware configuration.	19
Table 2.4:	Evaluated SPEC CPU benchmarks.	31
Table 2.5:	Simulated system configuration.	32
Table 3.1:	Machines for DL training	68
Table 4.1:	NVRAM-equipped server system configuration.	86
Table 4.2:	PMDK Benchmarks in Figure 4.16c	114
Table 5.1:	System for Evaluation	128
Table 5.2:	CXL Performance	129

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my Ph.D. advisor, Professor Jishen Zhao, for her constant support and guidance throughout my Ph.D. journey. When I was an undergraduate student applying for Ph.D. programs, Jishen was the only advisor brave enough to offer a Ph.D. to someone with no research background like me. Before I started my Ph.D. journey, I had no idea how fortunate I was to work with such a great advisor. She has offered me her sincere opinions on everything, not just the research but also how to deal with challenges outside the research. Jishen has supported my crazy research ideas, provided me with all the resources needed, and provoked my thoughts on every detail of my projects. She also showed me how to be a great researcher and a great person by being kind to everyone and respecting the thoughts of others. While searching for academic positions, she supported me in every way, from funding my travel to conferences to introducing me to many well-known researchers and having countless discussions. She is the best advisor and a great friend of mine. I am very lucky to have worked with and learned from her over the past few years.

I would like to thank Professor Steven Swanson. He always encouraged me to explore all my ideas and gave me his sincere guidance on many, if not all, of my research projects. I still remember my chat with Steve about my academic job application. He said I was ready to be a professor. This incentivized me to try that path.

Professor Yuan Xie has provided me with a lot of help in my academic job search. He invited me to his home during my visit to Santa Barbara and had a great discussion with me on my academic journey. He has connected me with his students and colleagues, written a reference letter for me, and introduced me to his connections during the conferences. I would never achieve so far without his help and encouragement.

I would like to thank Professor Dean Tullsen. I remember that while we were working on the NVLeak paper, Dean spent many nights polishing the paper with me, discussing our ideas, and giving me advice. It was nice to have someone else working on the paper with me at 2 AM.

I would like to thank Professor Geoffrey Voelker, Professor Farinaz Koushanfar, Professor Steven Swanson, and Professor Dean Tullsen for being on my doctoral committee. Their discussions and constructive feedback have greatly improved this dissertation.

I would like to thank my labmates and friends in the STABLE group. Xiao Liu was one of my first few friends when I came to the US and has been one of my best friends ever since. He helped me write my first research paper and gave me his sincere opinion on how to do research. He has also been a great friend in my life. We have traveled to many places, played video games, visited restaurants, and taken care of each other during the COVID-19 pandemic. Hui Shi is another best friend of mine. She had many funny ideas in her life, shared all kinds of exciting stories with me, and had nice chats with me. I would like to thank Hanxiang Huang for the nice research collaboration and great chats on all aspects of our Ph.D. life. Haolan Liu is such a nice buddy that I would miss my time without him being around. I enjoyed the great lab chats and, later, the camping experience with Hengyu Zhao. I would like to thank Cheng Fu, Yun Joon Soh, Theodoros Michailidis, and Jinpyo Kim for all the chats and fun in our lab.

I also would like to thank my friends in the NVSL group. Suyash Mahar is one of my best friends. His provoking jokes and occasional funny words (you know what I mean if you are also one of his friends) really provoked my days. We have been to many places, shared our life stories, and been through many things together. I still remember the time we spent at the Meta internship. It was the first time we really got to know each other, and we have shared all the fun times ever since. I would like to thank Yi Xu for being a great friend who went to many lab events together and chatted with me all the time. Jian Yang helped a lot during my initial Ph.D. research project and took me on a nice trip to Yellowstone National Park. I enjoyed the time spent with the other members of the NVSL, including Ziheng Liu, Mingyao Shen, Juno Kim, Nara Batsoyol, Lu Zhang, Yanbo Zhou, Ehsan Hajyasini, and many others.

I'm so fortunate to meet with some other friends during my journey. Xiaofan Yu, Sophia Sun, and Hui Shi have been a great group of friends with me. We shared a WeChat group, chatted

virtually about anything, made jokes about each other, and helped anytime anyone needed it. My time with them has been one of my best treasures during the past few years. Sihang Liu and Yi Ding have been great friends with me and have supported me throughout my academic job hunt. I really enjoyed our lunches, dinners, and all the other times during our journey at many conferences. Zesen Zhang and Haolan Liu have been great friends. They have listened to my stories, shared great times with me, and had lunches on campus. I have worked with Luyi Li, Jangseon Park, and Yue Pan on one of my final PhD research projects, and it's great to work with them. I have been fortunate to know Kazem Taram as a great researcher, a fantastic collaborator, and an excellent friend. I would like to thank Daniel Moghimi for supporting our research collaborations and his sincere help in my industry job hunting; I would never have gotten a Google offer without his support. Bingyu Shen is also a nice friend who helped me a lot, including with my industry job hunting. I'm deeply in debt to Marc Orr for being an excellent intern mentor during my Google internship and as a great friend. Marc has shared his knowledge and passion throughout over a hundred emails between us during my industry job hunting, with all his detailed thoughts in my shoes. I look forward to working with him at Apple after my graduation.

I would like to thank my travel gang and the great friends group, including Reinhard Wang, Kehan Wang, Ya He, Irene Sun, and Ruochen Wang. We have been on so many adventures and shared so many shiny memories. This journey has lit my life. Thank you for being such a great group of friends.

I would like to thank my cat, Inu, for being the best cat ever in the world. I brought Inu home during the COVID pandemic, and he has been such a lovely friend to me, even during this loneliest time. Inu is so bright and kind that he comes to me wherever I go, sitting on my sofa or sleeping with me at night. I remember all the nights he came to my bed and slept beside my pillow. He was so thoughtful to leave me alone once I fell asleep because I was so sensitive to noises, and he'd come to me in the morning. I appreciate his accompaniment and could not

imagine my life without him.

I want to thank my love, Ningzi Zhan, for sharing the most joyful life with me. She entered my life all of a sudden and made me realize how happy my life could be. I still remember the first time I met her when we called to discuss a travel plan. Even without meeting her in person, I fell in love with her thoughtful mind and kindness to everyone. It is such a gift that she happened to love me when I took my courage and expressed my love to her for the first time in my life. She has been my princess ever since; we have had so many adventures, visited so many great restaurants, and endured so many hard times. I remember every city we have been to, every shiny moment in our lives, every ups and downs in our journey, and all the fantastic moments. I remember we went to the fish market for crawfish only to find they were selling lobsters; I remember being her “the friend upstairs” and taking pictures for her Chinese dancing performance; I remember the jet ski experience and horse riding courses as part of my birthday gifts from her; I remember the first time we traveled together, just two of us, to Calgary for snowboarding. There are countless joyful moments every day and many unique experiences that I could never imagine without her. She is the best person in the world and the one I love the most.

Last, I could not have completed this journey without the wholehearted support and unconditional love of my parents, Pingping Yan and Wenying Wang. They have supported every choice in my life, educated me on how to be nice, and tried to put themselves in my shoes in all cases. I remember all those nights I took video calls with them, sharing ups and downs in my life, laughing with them, and getting mental care from their words. Although they had no formal training in the scientific field, they were always interested in what I was doing and what challenges I was facing. Most importantly, they know how to express their love to me, and I have never, for one day, felt in need of love because they have given me so much. I can never accomplish so much without any of their support and love. They are the best parents in the world, and I love them.

Chapter 2, in full, is a reprint of the material as it appears in IEEE/ACM International Sym-

posium on Microarchitecture, 2020. Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, Jishen Zhao. “Characterizing and Modeling Non-Volatile Memory Systems”. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in IEEE International Symposium on High-Performance Computer Architecture, 2022. Zixuan Wang, Joonseop Sim, Euicheol Lim, Jishen Zhao. “Enabling Efficient Large-Scale Deep Learning Training with Cache Coherent Disaggregated Memory Systems”. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in USENIX Security Symposium, 2023. Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, Jishen Zhao. “NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems”. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is currently being prepared for submission for publication of the material. Zixuan Wang, Jishen Zhao. “Fork is All You Need in Heterogeneous Systems”. The dissertation author was the primary investigator and author of this paper.

VITA

2018 B. S. in Computer Science, Zhejiang University
2024 Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Zixuan Wang, Jishen Zhao. “Fork is All You Need in Heterogeneous Systems”, Preparing for Submission.

Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, Jishen Zhao. “NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems”, *USENIX Security Symposium*, August 2023.

Zixuan Wang, Joonseop Sim, Euicheol Lim, Jishen Zhao. “Enabling Efficient Large-Scale Deep Learning Training with Cache Coherent Disaggregated Memory Systems”, *IEEE International Symposium on High-Performance Computer Architecture*, April 2022.

Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, Jishen Zhao. “Characterizing and Modeling Non-Volatile Memory Systems”, *IEEE Micro TopPicks*, March 2021.

Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, Jishen Zhao. “Characterizing and Modeling Non-Volatile Memory Systems”, *IEEE/ACM International Symposium on Microarchitecture*, October 2020.

ABSTRACT OF THE DISSERTATION

Towards Heterogeneous Systems: A Full-Stack Approach

by

Zixuan Wang

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Jishen Zhao, Chair

Heterogeneous systems emerge to enhance application performance by distributing tasks across various domain-specific accelerators and memory devices. However, adopting such systems often results in unanticipated performance characteristics in certain workloads, unforeseen vulnerabilities introduced by new architecture designs, and programming challenges utilizing novel architectures.

This dissertation explores a cross-stack approach to building heterogeneous systems. We start from the architecture level, studying the performance characteristics of emerging heterogeneous memory architectures. This led to our system-level study integrating diverse types of memory devices at a large scale, which facilitates efficient training of large machine learning

models. These explorations revealed vulnerabilities in emerging memory architectures and led to our development of architectural side-channel attacks and cross-stack defenses to secure heterogeneous systems. Finally, to facilitate the adoption of heterogeneous systems, we proposed a novel programming model that provides a unified interface for heterogeneous computing.

Chapter 1

Introduction

1.1 Background and Motivation

The ever-growing performance demands from modern applications drive the development of specialized memory and computing architectures: the non-volatile main memory brings persistence to the CPU main memory while delivering higher memory capacity; high-bandwidth memory (HBM) is integrated into modern GPUs to accelerate large machine-learning training and inference jobs; new types of memory interconnection protocols such as Compute Express Link (CXL) emerged to provide a standard interface between various types of memory devices and accelerator processors. Integrating such emerging architectures forms a heterogeneous system that distributes data and computing to domain-specific architectures to enhance application performance. Although widely deployed in the industry, such heterogeneous systems pose challenges across the entire computing stack, from architecture and systems to security and programming.

At *architecture* level, specialized memory and accelerator architectures have been proposed to fulfill modern workloads' requirements. These domain-specific architecture designs result in distinct software performance behaviors [1], making it essential to understand their

architecture properties to better optimize application performance. However, the proprietary nature of these designs presents a challenge, as industry vendors often keep the details of their memory architecture confidential. This lack of transparency motivates our research to focus on characterizing and modeling memory architectures to fill this knowledge gap.

We further observed that the memory architecture is moving toward a disaggregated and heterogeneous *system* design, where discrete memory and processor units are interconnected to work together. This approach is the key to sustaining performance growth for modern workloads. Traditional interconnection links like PCIe, however, are struggling to meet the increasing demands for higher bandwidth and lower latency in memory access required in modern applications. The Compute Express Link (CXL) emerges to address these challenges. By providing a cache-coherent interconnect standard, CXL allows processors and memory devices to dynamically share memory resources with considerably reduced latency. This motivates us to compose heterogeneous systems and leverage their cache coherence to provide high performance while maintaining programming compatibility.

Heterogeneous systems integrate various accelerators and memory architectures, thus posing unique *security* challenges. Each type of architecture has distinct vulnerabilities, and when combined, their interactions introduce new attack surfaces ranging from information leakage to data disruption. Studying security in these systems is crucial for protecting sensitive data processed through different architectures and ensuring the integrity of computations. This research is increasingly important across various fields, such as AI, big data analysis, and cloud computing, where the use of heterogeneous systems is growing.

Furthermore, current heterogeneous computing systems encounter *programming* challenges when incorporating new accelerators due to complexities at both system and programming levels. Historically, accelerators have established unique protocols to exchange data and manage tasks, relying on specialized operating system drivers and programming libraries. This leads to considerable complexity when adopting diverse accelerators in a single application: Applications

must employ specialized code for accelerators while ensuring compatibility between them, and the operating system must manage different drivers. Such programming and system requirements significantly limit the broader adoption of new accelerators.

1.2 Contribution

This dissertation presents a full-stack approach to building systems across the computing stack to support heterogeneous systems. We, at the architecture level, studied performance characteristics of emerging memory architectures. This led to our system-level study integrating diverse types of memory architectures on a large scale, facilitating training large AI models more efficiently. These explorations revealed vulnerabilities in emerging memory architectures and led to our development of architectural side-channel attacks and cross-stack defenses to secure heterogeneous systems. To facilitate the adoption of such large-scale systems, we developed a novel programming framework that unifies the programming interfaces for heterogeneous systems. This research journey allowed us to rethink how to design new architectures, build systems and programming techniques at a large scale, and secure the heterogeneous system stack.

1.2.1 Characterizing and Modeling Heterogeneous Memory Architecture

Chapter 2 presents a novel software framework [2] comprising a memory performance profiler, LENS, and a heterogeneous memory architecture simulator, VANS. The profiler LENS employs carefully crafted microbenchmarks to trigger specific hardware behaviors, resulting in different latency and bandwidth patterns with different memory architecture properties. Through the analysis of these performance patterns, we successfully reverse-engineered memory architecture designs across the hierarchy, from CPU cache to main memory. Subsequently, we developed a simulator VANS to model the architecture behaviors and performance across the memory hierarchy. This framework enabled us to be the first to reverse engineer the architecture of emerging

non-volatile memory systems. This framework is generic, allowing for reverse-engineering and simulation of multiple memory architectures with minimal adjustments, thereby facilitating future memory architecture research and development. This research is recognized by IEEE MICRO Top Picks [3] as one of the year’s 12 most influential architecture research papers.

Throughout this research, we observed that existing CPU-centric memory systems encounter scalability issues. New memories often have to sacrifice performance due to limitations such as the CPU’s power supply capacity. This reduction in performance degrades the adoption of new memory architectures. Looking ahead, we envision that emerging memory architectures will increasingly be integrated into heterogeneous systems. In these systems, new memory devices will be decoupled from the CPU memory bus, surpassing the current limitations and opening new possibilities in memory architecture development. As such, we investigated the system support for such heterogeneous designs.

1.2.2 Building Scalable Systems on Heterogeneous Architecture

Chapter 3 presents a heterogeneous system for machine learning framework to utilize CXL [4]. Specifically, we were the first to build a CXL-based disaggregated memory architecture, integrating it with GPUs to serve as an external memory pool. This initiative, supported by our development in operating systems and programming frameworks, allowed for the efficient training of large AI models with significantly fewer GPUs than conventional systems. This research outlines the critical role of CXL in modern computing systems. It enables faster and more efficient memory access and meets the diverse and growing requirements of modern applications such as AI and big data analytics, where rapid and efficient memory access is crucial. Considering the machine learning community is dealing with ever-growing AI model sizes that outpace GPU memory expansion speed, deploying CXL-based heterogeneous architectures at scale would significantly improve the performance and efficiency of machine learning workloads.

1.2.3 Attacking and Securing Heterogeneous Systems

Chapter 4 presents a software framework [5] to reverse engineer the design of memory architecture across the entire memory stack. Our prior research observed that the memory system organizes multiple memory architectures as a hierarchy and adds emerging memory as a new level to fill the performance gap while maintaining compatibility. These added layers are prone to architectural side-channel attacks that reveal sensitive information. To uncover and address such vulnerabilities, we developed a memory security analysis framework, NVLeak [5], to reverse engineer the architecture design of emerging memory to discover undocumented yet vulnerable architecture components. By leveraging this framework, we were the first to demonstrate practical attacks in an emerging memory system that break system security isolation boundaries, leaking private encryption keys and full database storage.

1.2.4 Programming for the Future Heterogeneous Systems

Chapter 5 presents our effort on simplifying programming for heterogeneous systems by rethinking the programming interface to abstract architectural details. We have observed that programming multi-accelerator systems is akin to multi-threading programming of multi-CPU systems. Multi-threading allows threads to handle specific aspects of an application’s functionality and relies on CPU cache coherence to access shared code and system resources. CXL and similar technologies enable cache-coherent interconnection across diverse accelerator architectures, including CPUs, making a multi-threading programming model for heterogeneous systems feasible. In this model, each accelerator runs an application thread, communicating with others using standard multi-threading techniques like shared memory and locks. The code for each accelerator thread is defined similarly to a CPU thread task, with transparent access to system functionalities like filesystems without custom code. We explored a software-based language runtime system based on WebAssembly that allocates application threads across different CXL-

connected accelerators to implement this. This further enables conventional multi-threading programs, written initially for multi-CPU systems, to use the multi-accelerator heterogeneous system without modifications. Taking this idea further, we envision the future of heterogeneous programming unified in a standard, well-established programming model such as multi-threading programming, thus alleviating the programming complexities in current heterogeneous systems.

Chapter 2

Characterizing and Modeling

Heterogeneous Memory Architecture

2.1 Introduction

Non-volatile RAMs (NVRAMs) [6–8] have been envisioned as a new tier of memory in server systems, offering comparable performance to DRAM with the durability property of storage devices [1, 9–12]. Seeing the great value, a large body of prior studies investigated how to exploit NVRAM to benefit future computer systems. Yet only recently has the first server-grade NVRAM DIMM product come to market, namely Intel Optane DC Persistent Memory (aka. Optane DIMM) [13].

The latest characterization studies [1, 14–18] exposed substantial discrepant performance characteristics compared to what we thought before the real products were released. For instance, Figure 2.1 compares results of the widely-used Persistent Memory Emulation Platform (PMEP) [19] and our Optane DIMM server measurement (denoted as *Optane*) on single-thread read and write bandwidth (Figure 2.1a) and latency per cache line (CL) access (Figure 2.1b) with a pointer chasing microbenchmark. The microbenchmark randomly accesses a contiguous

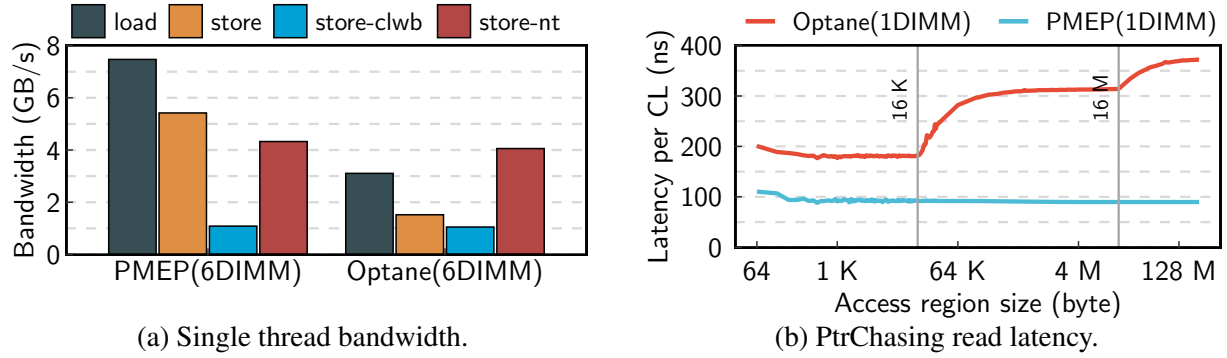


Figure 2.1: Comparison between PMEP [19] emulation and Optane DIMM-based real machine measurement. *Store-clwb* are writes followed by `clwb` cache write-back instructions [20]; *store-nt* are non-temporal writes [20].

data region with fixed 64B objects. By varying the data region size from 64B to 256MB, we observe a clear inconsistency between the emulation and real-system results (discussed in detail in Section 2.2.2). Other previous NVRAM emulators and simulators also generate different performance characteristics compared to real-machine results, as discussed in Section 2.2 and Section 2.3. As such, previous NVRAM emulation and simulation tools are insufficient to model modern real NVRAM systems.

Moreover, Intel’s announcement [13] mentions that the Optane DIMM adopts an on-DIMM buffer structure, although without further details. The *Optane* curve in Figure 2.1b also shows buffer effects at the sizes of 16KB and 16MB, respectively. Both indicate that Optane DIMM adopts a much more complex microarchitecture design than conventional DRAM DIMMs. However, most of the existing Optane DIMM studies focused on systems-level performance profiling and system software design [14, 16, 17, 21, 22], without detailed investigation on microarchitecture. Furthermore, the existing architecture- and system-level performance profiling tools are designed for DRAM-based memories, inefficient in investigating the architectural structure details of real NVRAM systems as compared in Table 2.1.

Our goal in this paper is to facilitate research on architecture and systems design of NVRAM-based memory systems by (1) investigating detailed performance and microarchitecture

characteristics of Optane DIMM, (2) designing a generic architecture-level performance profiling and reverse engineering framework, which identifies the key architectural and performance characteristics of NVRAM systems, and (3) developing a memory simulator that models the microarchitecture designs of modern NVRAM DIMMs. To achieve our goal, we make the following technical contributions:

- **Profiling and reverse engineering framework.** We design a Low-level profiler for Non-volatile memory Systems (LENS), which consists of a set of profiling tools and microbenchmarks to analyze the architectural and performance characteristics of NVRAM-based memory systems. LENS is designed as a Linux kernel module. It exposes detailed architectural characteristics of memory systems, such as on-DIMM buffer size and hierarchy, queuing scheme, and the access granularity of each component.
- **Real-system characterization.** We employ LENS to perform a comprehensive architecture and performance analysis on Optane DIMM-based servers. Using LENS, our profiling reveals a detailed picture of the Optane DIMM microarchitecture.
- **NVRAM simulator.** We develop a Validated cycle-Accurate NVRAM Simulator (VANS), which models NVRAM-based memory system architecture designs based on our characterization. We validate VANS against extensive Optane DIMM-based real-machine profiling. VANS adopts a flexible modular design, which allows users to explore a design space with various architecture designs and extend the simulator to model other NVRAM microarchitecture designs.
- **Case studies.** We employ VANS to evaluate two architecture optimizations on Optane DIMM, Lazy cache and Pre-translation, which address Optane DIMM-based memory system performance inefficiencies in cloud workloads identified by LENS.

2.2 Background and Motivation

Overview of Our Motivation. Recent Optane DIMM profiling [1, 14–18] shows different performance characteristics compared to many previous NVRAM studies [23–28]. However, most of the Optane DIMM studies focus on system-level performance analysis and system software design. Industrial documents indicate that the memory controllers and Optane DIMMs adopt more complex microarchitecture designs than conventional DRAM DIMM-based memory systems, but without many of the details. These challenges motivate us to explore microarchitecture design and memory system modeling based on real NVRAM products.

2.2.1 Optane DIMM-based Server Systems

Server System Organization. Figure 2.2 illustrates an example of Optane DIMM-based server systems. Optane DIMMs – denoted as NVRAMs – sit on the memory bus along with DRAM DIMMs. They are controlled by the processor’s integrated memory controllers (iMCs). Intel’s Cascade Lake is the first microarchitecture to support Optane DIMM [29]. Specifically, our server systems adopt one or two processor dies in each CPU socket. Each processor die has two iMCs, and each iMC supports three memory channels. Each channel is configurable to have zero or one Optane DIMM.

Operation Modes. Optane DIMMs operate in one of two modes: Memory Mode or App Direct Mode. The Memory Mode does not support data persistence. Each memory channel incorporates an Optane DIMM and a DRAM DIMM; the DRAM serves as a data cache for the Optane DIMM. In App Direct Mode, the Optane DIMMs are used as persistent memory, which unifies the fast access interface of memory with the persistence property of storage [27, 30]. For example, programmers can directly issue load/store instructions to access in-memory data structures while NVRAM system software and hardware ensure that the data structures are recoverable in the face of system crashes and power loss [23, 31].

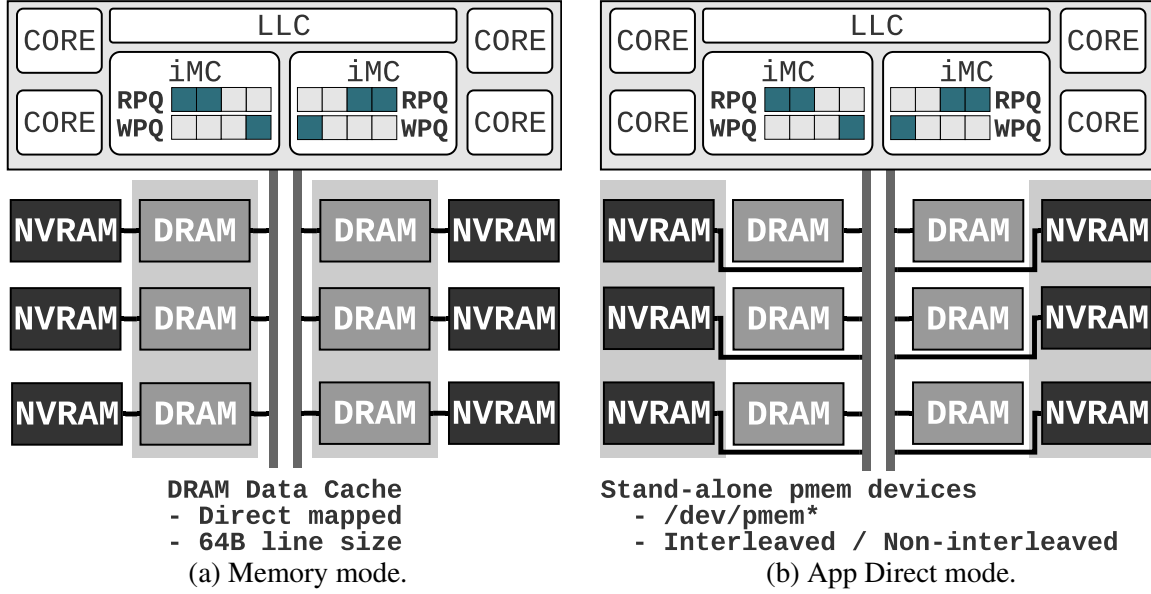


Figure 2.2: Optane DIMM-based system configurations.

Table 2.1: Comparison of profiling tools.

Tools	Basic			On-DIMM buffer			Long tail-latency*	
	Latency	Bandwidth	Addr mapping	Size	Granularity	Hierarchy	Frequency	Granularity
MLC [34]	✓	✓	✗	✗	✗	✗	✗	✗
perf [35]	✓	✓	✗	✗	✗	✗	✗	✗
DRAMA [36]	✓	✗	Partial	✗	✗	✗	✗	✗
LENS (our design)	✓	✓	Partial	✓	✓	✓	✓	✓

* Long write latency during repeated writes to the same memory area.

Known Architecture Components. Industrial documents disclose several architectural design components related to Optane DIMM [13, 32, 33]; but most of the descriptions do not provide the details as identified by our study (Figure 2.4). To ensure data persistence, each iMC maintains a write pending queue (WPQ) and a read pending queue (RPQ) for Optane DIMMs; the WPQs belong to the asynchronous DRAM refresh (ADR) domain [32]. CPU ensures that the data reaches the ADR domain is persisted during power outage. Optane DIMMs support CPU cache line granularity access. The current implementation adopts 3D-Xpoint chips as NVRAM media with a 256-byte access granularity [33]. In addition, each Optane DIMM maintains buffers inside the DIMM controller; sub-256-byte write accesses will trigger a read-modify-write (RMW) procedure in the buffers [13].

2.2.2 Performance Discrepancy

Many previous NVRAM studies perform experiments using NVRAM emulators and simulators, which assume that NVRAM performs like a slower DRAM. For instance, widely-used NVRAM emulators PMEP [19] and Quartz [37] model NVRAM systems by stalling the CPU for additional cycles (estimated as the cycles that the CPU would have to wait if DRAM is replaced by a slower NVRAM) and throttling bandwidth. Other studies simply inject delays in software codes to emulate NVRAM’s slower writes relative to DRAM [31, 38]. However, Optane DIMM characterization [1, 14–18] demonstrates that real NVRAM systems have much more complex performance behaviors than conventional DRAM systems. As an example, Figure 2.1 shows our experiments that illustrate two observations on the performance discrepancy. First, Figure 2.1a shows that PMEP models both load and store bandwidth higher than the bandwidth of non-temporal stores (*store-nt*). However, with Optane DIMM, using non-temporal stores lead to a higher bandwidth than the others. Second, PMEP maintains a stable latency per cache line read, as shown in Figure 2.1b. If Optane DIMM were a slower DRAM, the curve would appear stable with a similar shape as the PMEP curve. However, Optane DIMM read latency increases with the growth of the pointer chasing region size, showing three clear segments in the curve. Both observations show that the real NVRAM system is not simply a slower DRAM DIMM-based system.

2.2.3 Inefficient NVRAM Simulation Models

Previous memory simulators, such as DRAMSim2 [39] and Ramulator [40], model memory architecture based on conventional DRAM systems. We find that these simulation models fail to accurately match the performance behaviors of Optane DIMM, as demonstrated in Figure 2.3. In Figure 2.3a, we compare the load/store bandwidth and latency of various memory simulators with Optane DIMM. The average accuracy is evaluated as the arithmetic

mean of accuracies under experiments with different access sizes. Figure 2.3a shows a large mismatch of bandwidth and latency characteristics between simulation and our real-system profiling. Figure 2.3b shows the same experiment as in Figure 2.1b, performed by Ramulator simulation. Similar to PMEP results, the simulated read latency appears to be stable, whereas Optane DIMM system profiling shows an increasing latency with larger pointer chasing regions.

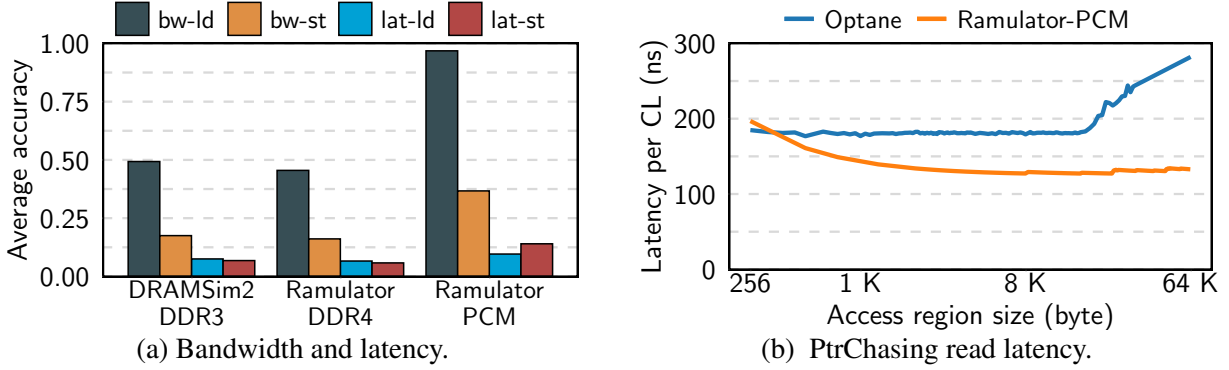


Figure 2.3: Comparison between memory simulators and Optane DIMM system profiling. (a) Simulator average accuracy wrt. Optane DIMM load/store bandwidth (*bw-ls* and *bw-st*) and latency (*lat-ls* and *lat-st*). (b) Comparison between Ramulator and Optane DIMM on read latency per cache line with the pointer chasing test.

2.2.4 Insufficient Profiling Tools

In order to develop a memory simulator that models the sophisticated performance behavior and microarchitecture of real NVRAM systems, we need to collect sufficient information about detailed performance characteristics. This requires a comprehensive architecture-level memory system profiling. Most previous performance profiling tools focus on investigating basic memory system performance characteristics, such as latency, bandwidth, and access counts. Beyond these characteristics, DRAMA [36] examines certain address mapping schemes. However, as illustrated in Table 2.1, none of the widely-used performance profiling tools allows us to analyze detailed on-DIMM buffering and management schemes of Optane DIMM.

2.3 LENS: Low-level NVRAM Profiler

To address the challenge of insufficient profiling tools, we propose a *Low-level profiler for Non-volatile memory System* (LENS). LENS consists of a set of NVRAM profiling tools and low-level microbenchmarks. With detailed performance profiling, LENS also allows us to reverse engineer the microarchitectural design of NVRAM systems with on-DIMM buffers and various control schemes. In this paper, we use LENS to profile the detailed architecture design of Optane DIMM, discussed in Section 2.3.2.

2.3.1 LENS Framework

LENS adopts three key components – *probers* – to examine the following three aspects of NVRAM architecture design, respectively.

- **Buffer prober** – Analyzes on-DIMM buffers’ architecture and their properties, including buffer (or queue) hierarchy, capacity, and access granularity.
- **Policy prober** – Analyzes NVRAM control policies on data migration and multi-DIMM interleaving.
- **Performance prober** – Facilitates the above two probers to analyze performance characteristics, including memory bandwidth and the access latency of various buffers and queues in NVRAM DIMMs and iMCs.

Each prober employs customized microbenchmarks to trigger specific hardware behaviors, which leads to different latency and bandwidth patterns with different memory architecture properties (Table 2.2). By analyzing the performance patterns, we identify the corresponding microarchitecture properties and parameters.

Table 2.2: LENS overview.

Prober	Microbenchmark	Hardware Behavior	Microarchitecture
Buffer	PtrChasing (64B block)	Buffer overflow	Buffer size
	PtrChasing (various block)	R/W amplification	Buffer entry size
	Read-after-write	Data fast-forwarding	Buffer hierarchy
Policy	Sequential/Strided write	Interleaving speedup	Interleaving scheme
	Overwrite (256B region)	Data migration	Migration latency
	Overwrite (various region)	Data migration	Migration block size
Perf.	Strided write	Stable amplification	Internal bandwidth
	N/A	N/A	Internal latency

Microbenchmarks. LENS provides three microbenchmarks: *pointer chasing*, *overwrite*, and *stride*.

Pointer chasing is a random memory access benchmark: it divides a contiguous memory region – referred to as a pointer chasing region (PC-Region) – into equal-sized blocks (PC-Blocks); it reads/writes all PC-Blocks in a PC-Region in random order, and sequentially accesses data within each PC-Block. In order to bypass CPU caches, while still generating cacheline-sized memory accesses, all pointer chasing tests are implemented using non-temporal AVX512 load/store instructions. Pointer chasing has three variants to detect various buffer architecture characteristics: (1) collecting average latency per cache line with a fixed PC-Block size across various PC-Region sizes, (2) quantifying read and write amplification by using a fixed PC-Region size, while varying the size of PC-Block, (3) issuing read-after-write requests, which issue writes in a pointer chasing order, followed by reads in the same order.

Overwrite repeatedly generates sequential writes to a fixed memory region, and then measures the execution time of each iteration. It has two variants: (1) collecting the execution time of each write iteration with a fixed memory region size, (2) measuring the frequency of long tail-latency by changing the memory region size.

Stride sequentially reads or writes to a set of stride cache lines with a fixed striding distance. It has two variants: (1) measuring bandwidth by using a fixed striding distance and increasing the access size, (2) characterizing multi-DIMM interleaving by a fixed total access

size and a variable striding distance.

Buffer Prober detects the on-DIMM buffer capacity, entry size, and organization, by measuring the latency change caused by buffer overflow and read/write amplification.

To identify the buffer capacity, the buffer prober runs the pointer chasing microbenchmark with fix-sized objects (PC-Block). The prober measures the average read/write latency by scanning through various pointer chasing region (PC-Region) sizes. Once a PC-region is sufficiently large to overflow a buffer, the prober will detect a rapid increase in read/write latency, e.g., the inflection point of the Optane latency curve at the 16KB region size in Figure 2.3b. As such, we estimate that the buffer capacity equals to the PC-Region size corresponding to the inflection point in the latency curve.

The prober also examines buffer organization by answering two questions: (i) How many levels in the buffers? (ii) If there are multiple levels of buffers, are they organized as a multi-level inclusive hierarchy or multiple independent buffers? We answer question (i) by analyzing the aforementioned buffer overflow results: the number of the inflection points in a latency curve indicates the number of buffers with different capacities. In fact, we identify that Optane DIMMs adopt multiple levels of on-DIMM buffers by inducing multiple overflows under different PC-Region sizes (Section 2.3.3). To answer question (ii), the prober measures the data fast-forward effect – the effect of reading the existing dirty data in a buffer before it is written back to the underlying memory or the next level of buffer – by a pointer chasing read-after-write (RaW) test. Independent buffers can fast-forward data from each buffer in parallel. Therefore, the RaW latency is shorter than the sum of independently collected read latency and write latency. A multi-level inclusive buffer hierarchy does not have such a parallel data fast-forward behavior.

To identify the buffer entry size, the prober evaluates the read/write amplification – measured as a ratio of actually read/written data size to the requested data size. For example, let's consider a buffer with 256B entries, where a read request of 64B will read 256B into

the buffer, leading to a read amplification of 4. As one memory bus transfer has a fixed size with a fixed latency, a read/write amplification will result in more bus transfers detected as a latency increase. In practice, the buffer prober identifies the entry size of various buffers by a series of pointer chasing tests with different PC-Block sizes. Once the read/write amplification drops to one, that PC-Block size is considered as the buffer entry size. We observe that Optane DIMMs adopt different entry sizes in different levels of buffers (Section 2.3.3). Due to the lack of hardware counters we can access on Optane DIMM machines, we indirectly evaluate the read/write amplification by deriving an “amplification score”. It is calculated as the ratio of latency of the buffer overflow case to the non-overflow case. As such, the amplification score drops to one if and only if the actual amplification drops to one.

Policy Prober investigates NVRAM control policies, including data migration (e.g., for wear-leveling) and multi-DIMM interleaving policy. Typical NVRAM wear-leveling schemes migrate data from one NVRAM media location to another to maintain evenly distributed wear out. The prober detects the frequency, granularity, and latency overhead of such data migration procedures.

To detect the migration frequency and latency, the prober employs the overwrite microbenchmark to constantly write 256B regions and measures the latency of each 256B write. Once a migration occurs in this region, e.g., triggered by a wear-leveling algorithm, the subsequent writes cannot be issued until the migration completes. As a result, the latency of a write delayed by a migration is over a magnitude higher than a normal write, showing as a tail latency. The prober estimates the migration latency to be equal to the elevated tail latency. In addition, the prober also collects the time intervals between two consecutive migrations to calculate the migration frequency.

To detect the migration granularity, the prober performs overwrite tests on regions of various sizes and collects the frequency of migration. Once the region size is sufficiently large to cross over two migration blocks, the frequency will drop. Therefore, we identify the migration

granularity as the overwrite region size that leads to a frequency drop.

The policy prober also examines the multi-DIMM interleaving behavior, which is an address mapping mechanism to achieve higher bandwidth by spreading data among different DIMMs. To detect such an interleaving scheme in an NVRAM design, the prober measures the speedup of sequential and strided writes on NVRAM DIMMs. Compared to non-interleaved DIMMs, interleaved DIMMs result in lower execution time with same-sized sequential writes (Figure 2.7a). The prober also identifies the interleaving granularity (the size of the maximum continuous memory block assigned to each DIMM) by detecting the repeated pattern in sequential and strided writes execution time on interleaved NVRAMs.

Performance Prober facilitates the analysis of the other two probers to measure the device bandwidth and latency for on-DIMM architecture components. The performance prober measures the read bandwidth of a buffer by a stride read experiment, with a stride size equals to the buffer entry size. Each buffer entry is read exactly once to prevent the impact of an upper-level buffer on the bandwidth. To measure the latency of accessing a buffer, the prober takes pointer chasing latency results from the buffer prober and then estimates the buffer miss rate based on the buffer size and access granularity. Then, the prober calculates the buffer latency as a function of pointer chasing latency and buffer miss rate.

LENS Implementation. We build LENS as a Linux kernel module to avoid the overhead of switching between user and kernel spaces. We implement all our microbenchmarks in assembly language to enforce a well-controlled memory access behavior. We also disable process preemption and hardware prefetchers to avoid noises on probing results.

Table 2.3: Server hardware configuration.

CPU	Intel Cascade Lake engineering sample 24 Cores per socket, 2.2 GHz 2 sockets, HyperThreading off
L1 Cache	32KB 8-way I\$, 32KB 8-way D\$, private
L2 Cache	1MB, 16-way, private
L3 Cache	33MB, 11-way, shared
TLB	L1D 4-way 64 entries, L1I 8-way 128 entries STLB 12-way 1536 entries
DRAM	DDR4, 32GB, 2666MHz, 2 sockets, 6 channels per socket
NVRAM	Intel Optane DIMM, 256 GB, 2666 MHz 2 sockets, 6 channels per socket

2.3.2 Optane DIMM System Characterization

We employ LENS to characterize the architecture details of a real server memory system with Optane DIMMs.

Real System Configuration. Table 2.3 lists our server hardware configuration. We set the Optane DIMMs to App Direct mode and use a customized Linux v4.13 kernel that supports App Direct mode. We use the `—ipmctl—` tool [41] to configure these Optane DIMMs in non-interleaved mode; we also employ the `—ndctl—` tool [42] to create a Linux `—pmem—` device on a single Optane DIMM from the local NUMA domain. LENS creates a dummy filesystem on the `—pmem—` device and performs experiments in the kernel space. We set the Model Specific Register (MSR) 0x1a4 to 0xf to disable the following CPU cache prefetchers to avoid noises in profiling [43]: L2 hardware prefetcher, L2 adjacent cache line prefetcher, DCU prefetcher, and DCU IP prefetcher. We run our experiments for over 500 times and show average profiling results with error envelopes/bars (some error envelopes may be too narrow to be visible in the figures).

Overview of Our Characterization and Observations. Figure 2.4 shows an overview of our real-machine characterization. Each prober identifies certain memory system architecture and

performance characteristics, as illustrated by arrows in the graph. Figure 2.8 shows details of the Optane DIMM microarchitecture identified by our characterization. We identified the write-pending-queue (WPQ) size and multi-DIMM interleaving scheme in iMC. We also identified two on-DIMM buffers, a 16KB SRAM-based read-modify-write (RMW) buffer and a 16MB DRAM-based address indirection translation (AIT) [44] buffer, with 256B and 4KB access granularity, respectively. We find that these two buffers are organized as a two-level inclusive buffer hierarchy rather than independent buffers. We also identified the size of a load-store-queue (LSQ) [45], which reorders the incoming requests to perform write combining. Finally, we identified a long tail-latency effect, which may be caused by wear-leveling data migration. We present details of our characterization results and observations in Section 2.3.3 and Section 2.3.4.

The Existence of the Architecture Components. Industrial articles revealed the existence of several aforementioned components, as described in Section 2.2.1. In addition, Intel’s announcement [13, 45] declared that transactions could be re-ordered in Optane DIMM, indicating an on-DIMM reorder queue, which we refer to as LSQ. Optane DIMM is also believed to have on-DIMM SRAM and DDR4 DRAM modules [46]. However, these articles do not provide detailed information about these components, such as size, access granularity, management policy, and latency. We employ LENS to profile these parameters and identify other architecture components.

Profiling Results Confirmation. We confirmed as much of our profiling results as possible with the vendor, including our bandwidth profiling, idle latency of sequential and random accesses, and load latency across various access granularities.

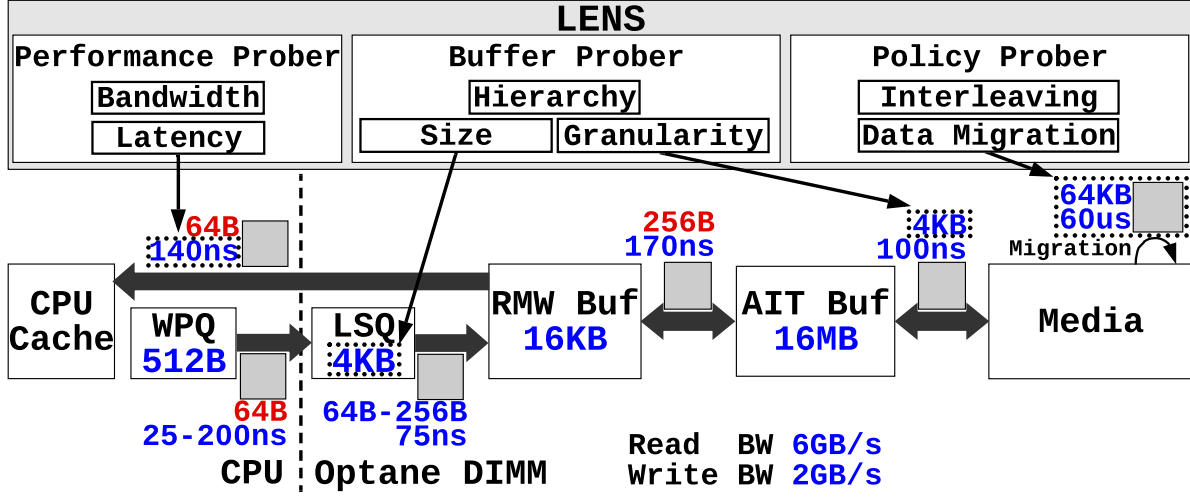


Figure 2.4: LENS probes and Optane DIMM parameters. Red numbers are obtained from Intel documents; blue numbers are characterized by LENS.

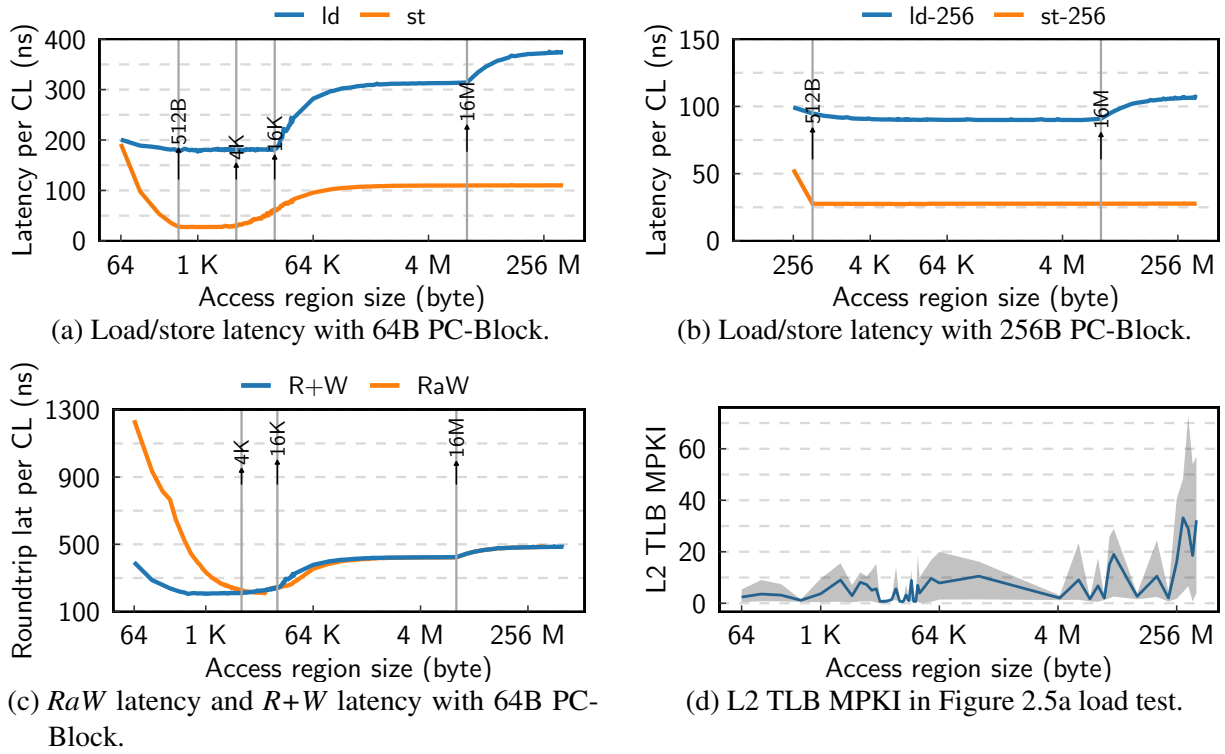


Figure 2.5: LENS buffer prober tests on Optane DIMM. The curves show average results of 500 runs; shaded areas represent error envelopes. In (c) *RaW* is read-after-write, *R+W* is the sum of load and store latency in (a).

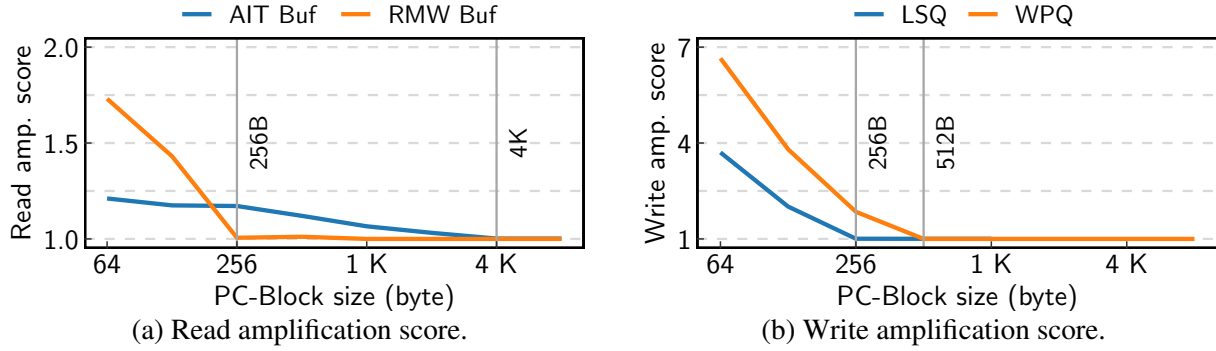


Figure 2.6: Read(a) and write(b) amplification scores in our buffer prober test.

2.3.3 On-DIMM Buffer Analysis

We run the LENS buffer and performance probes to profile the on-DIMM buffer architecture design.

Buffer Capacity. Figure 2.5 shows LENS buffer prober results. Figure 2.5a and Figure 2.5b show a pointer chasing read/write test with various PC-Block and PC-Region sizes. The experiments measure latency per cache line (CL) access. The shape of the curves exposes the read requests overflow points at 16KB and 16MB, where the read latency drastically changes. This indicates that the Optane DIMM has two levels of read buffers, one 16KB and the other 16MB. Figure 2.5a also shows that the write curve (denoted by *st*) has two overflow points at 512B and 4KB, respectively. This indicates that the memory has two write buffers, namely WPQ and LSQ, of two distinct capacities. Note that the L2 TLB of our server machine has 1536 entries, which can store the address translation of up to $4\text{KB} \times 1536$, i.e., 6MB, of data. However, Figure 2.5a shows that the read latency significantly increases at 16MB instead of 6MB, ruling out that this happens due to L2 TLB misses. Furthermore, we track L2 TLB misses for the load test. As shown in Figure 2.5d, the L2 TLB miss rate remains stable without significant change at 16KB and 16MB access regions. Therefore, we conclude that L2 TLB misses are not the main reason for the sudden latency increases in Figure 2.5a.

We refer to the 16MB buffer as AIT Buffer. Because its read latency is approximately 100ns as shown in Figure 2.5a, it is likely to locate inside the on-DIMM DRAM, where AIT resides [13, 44]. We believe that the 512B buffer is WPQ [32], given its small size; the 4KB buffer is an on-DIMM LSQ that reorders the read/write requests [45].

Access Granularity. Figure 2.6a shows our read amplification tests. Our results indicate that the RMW Buffer and AIT Buffer adopt 256B and 4KB access granularities, respectively. Figure 2.6b shows our buffer write amplification tests. It demonstrates that the two write queues, WPQ and LSQ, adopt 512B and 256B granularities, respectively. As a result, a *mfence* will cause the WPQ to flush in total 512B data; the LSQ combines 64B writes into 256B in order to reduce RMW operations.

Buffer Hierarchy Organization. Figure 2.5c shows our read-after-write (RaW) experiment to characterize the buffer hierarchy organization. Figure 2.5c indicates that RMW Buffer and AIT Buffer form a two-level inclusive hierarchy – if they were independent from each other, they would have demonstrated a RaW latency speedup at 16MB by fast-forwarding data in parallel. They do not appear to be exclusive either, because they adopt different access granularities and entry sizes.

Figure 2.5c also demonstrates that the read-after-write latency (denoted as *RaW*) is significantly higher than the total latency of performing individual reads and writes (denoted as *R+W*) for small PC-Regions. The key reasons are small-sized RaW requests (i) trigger frequent memory bus redirection [47] and (ii) under-utilize the WPQ and LSQ capacity. Therefore, due to the identified architectural design, small-sized requests in Optane DIMMs tend to perform better with pure reads and writes than with mixed read and write access patterns. In addition, the high RaW latency with small PC-Region sizes also indicates that *mfence* flushes the LSQ, because: (i) RaW latency reduces when the PC-Region size increases; (ii) RaW latency equals the *R+W* latency when the PC-Region size reaches 4KB, the size of the LSQ.

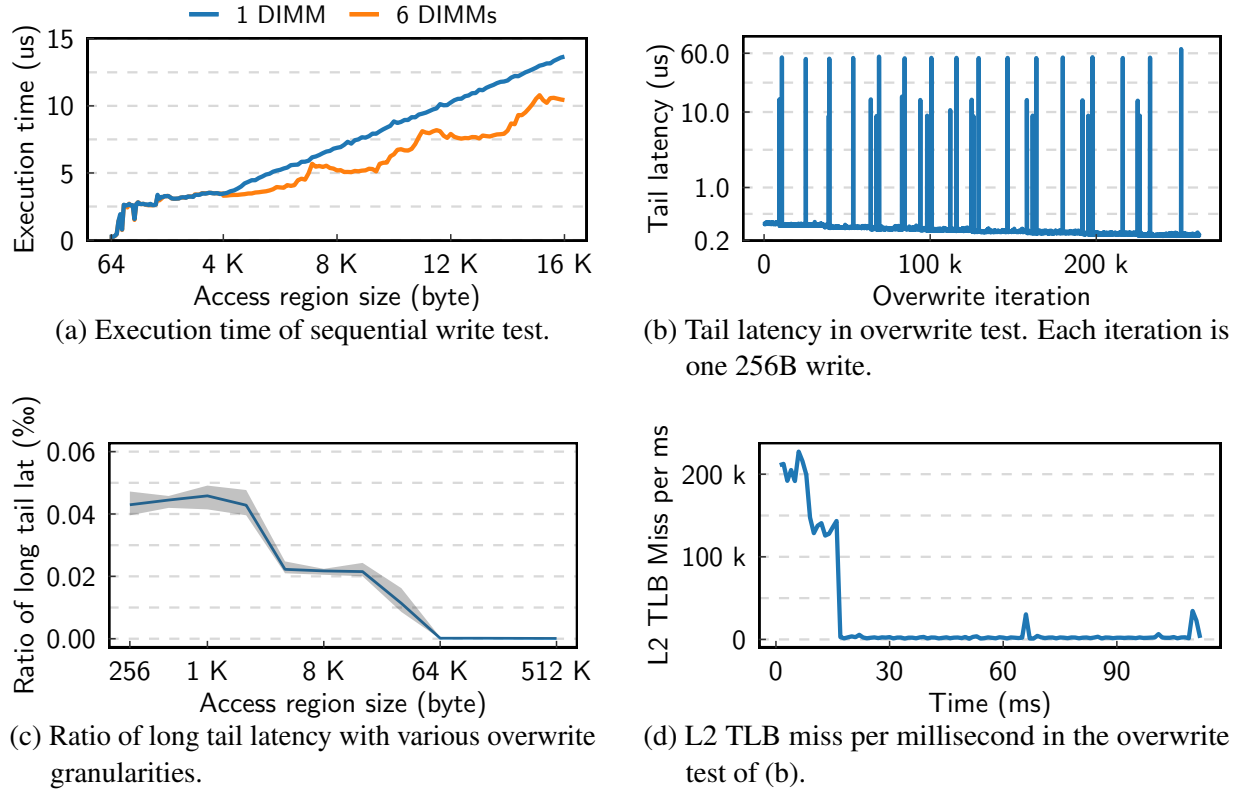


Figure 2.7: LENS policy prober tests on Optane DIMM.

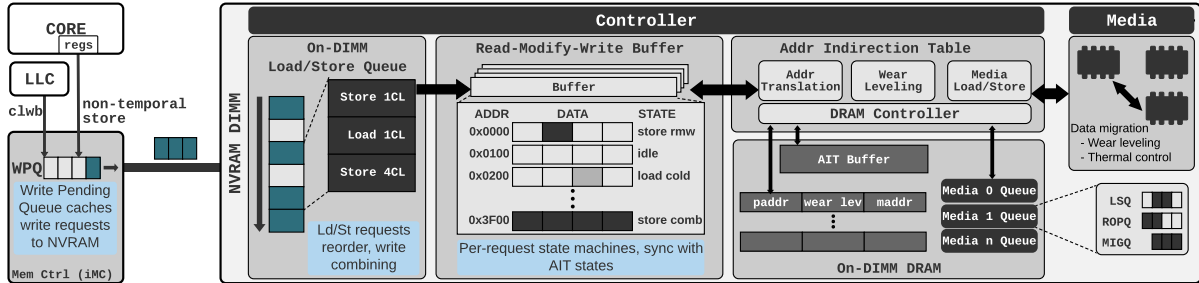


Figure 2.8: VANS overview.

2.3.4 Policy Analysis

We run LENS policy and performance probers to profile Optane DIMM management schemes.

Multi-DIMM Interleaving Analysis. Figure 2.7a shows a multi-DIMM interleaving policy analysis. LENS measures the execution time of various sized sequential writes on interleaved and

non-interleaved DIMMs, respectively. In the interleaved experiment, the first 4KB has a similar execution time as in the non-interleaved case, indicating that the first 4KB is written to a single DIMM. We also observe a repeated execution time pattern every 4KB, indicating that different 4KB data writes are directed to different DIMMs. These observations indicate a 4KB granularity for multi-DIMM interleaving. We consider the reason for the interleaving granularity is to fully utilize the 4KB sized LSQ and 4KB entries in AIT Buffer.

Tail Latency Analysis. Figure 2.7b shows the overwrite experiment for tail latency analysis. The policy prober constantly writes data to the same 256B memory region, and measures the latency of each 256B write. We observe a long tail latency every $\sim 14,000$ write iterations (i.e., every 3.4MB for 256B overwrite test), which incurs over $100\times$ latency penalty on average. We do not observe similar trends on DRAM. Moreover, L2 TLB miss rate remains stable in the overwrite experiment as shown in Figure 2.7d. Therefore, we consider wear-leveling as a major contributor to the tail latency.

To study the size of the blocks that wear-leveling tracks, we increase the size of the overwrite test region and count the frequency of the long tail latency. Each test writes the same amount of data to NVRAM. As shown in Figure 2.7c, the frequency of long tail latency dramatically drops, once we overwrite on 64KB or larger memory regions. This indicates that a possible block size for wear-leveling is 64KB.

2.4 VANS: Validated NVRAM Simulator

Based on our profiling observations, we build *Validated cycle-Accurate NVRAM Simulator* (VANS) that models Optane DIMM’s microarchitectural design and its parameters. Figure 2.8 depicts an overview of VANS (CPU core and LLC are not part of VANS). We adopt a modular implementation by decoupling the architecture’s modules as much as possible, in order to allow users to modify or extend the simulator with various architectural designs. VANS also offers an

interface to be attached to full-system simulators, such as gem5 [48].

Because no publicly available formal verification tool exists for NVRAM DIMMs, we validate VANS’s accuracy by comparing the simulated bandwidth, load and store latency, and various SPEC CPU benchmarks performance results with Optane DIMM real-system profiling. In addition, the on-DIMM DRAM model is verified by Micron’s verification model [49] and Cadence toolchain [50].

2.4.1 Simulation Model Design

VANS models both released Optane DIMM hardware components (e.g., WPQ in iMC) and undocumented microarchitectural components (e.g., LSQ, multi-level buffers). The iMC in VANS supports multi-DIMM control, which controls multiple DRAM and NVRAM DIMMs to provide both Memory and App Direct modes. We implement the WPQ in iMC with Asynchronized DRAM Refresh (ADR). The NVRAM DIMM model consists of an LSQ, an RMW Buffer, and an AIT. The LSQ serves as the highest-level storage in the DIMM, directly queuing load/store requests from the iMC. During each scheduling epoch, the LSQ performs write combining to reduce the number of read-modify-write operations.

The RMW Buffer receives read and write requests from the LSQ, and accesses the AIT Buffer at certain granularity (by default 256B based on our Optane DIMM analysis). The RMW Buffer performs read-modify-write operations if write requests are smaller than 256B. We place this buffer in SRAM, based on our Optane DIMM profiling. The AIT consists of a translation table and a data buffer (AIT Buffer). The translation table stores the records of CPU address to media address translation. It also stores the media wear-leveling records in each table entry. If one media block is wearing out, AIT stalls the inflight CPU writes to this block, migrates the data into another media block, updates the translation record, and then resumes the CPU write. The AIT Buffer stores data from the media to accommodate read and write requests from RMW Buffer. We place the AIT table and buffer in the on-DIMM DRAM, based on our Optane DIMM

profiling.

VANS adopts a first-come-first-serve scheduling policy by default inside the NVRAM, based on our Optane DIMM profiling. In addition, on the DIMMs, both RMW and AIT buffers maintain a state machine for each buffer entry running in parallel. The RMW Buffer issues FIFO requests to the AIT Buffer. The AIT Buffer issues FIFO requests to the NVRAM media. The scheduling policy can be modified by users to adapt to other NVRAM DIMM devices (Section 2.4.5).

The iMC and the NVRAM DIMM communicate by a request/grant scheme [45]: for example, when the iMC needs to read a cache line from NVRAM DIMM, it sends a cache line read request and waits for NVRAM DIMM’s response. When the NVRAM DIMM finishes fetching this cache line into the RMW Buffer, it notifies the iMC by sending a data-is-ready signal and requests an entry in the iMC’s read pending queue (RPQ). It then sends data to the RPQ, after the iMC allocates the space for it.

Optane DIMM adopts an unpublished DDR-T protocol, which extends DDR4 with persistence-related commands. Therefore, we model the on-DIMM DRAM timing based on the DDR4 protocol.

2.4.2 DRAM Model Verification

To verify our DRAM model, we first catch memory traces from SPEC2006 and SPEC2017 benchmarks listed in Table 2.4; these traces are fed into VANS to obtain internal DRAM command traces. Then, we employ Micron’s DDR4 verification model [49] with Cadence toolchain [50] to test these command traces. The results demonstrate that our model does not generate any illegal DDR4 command, showing that our on-DIMM DRAM model complies with DDR4 specification.

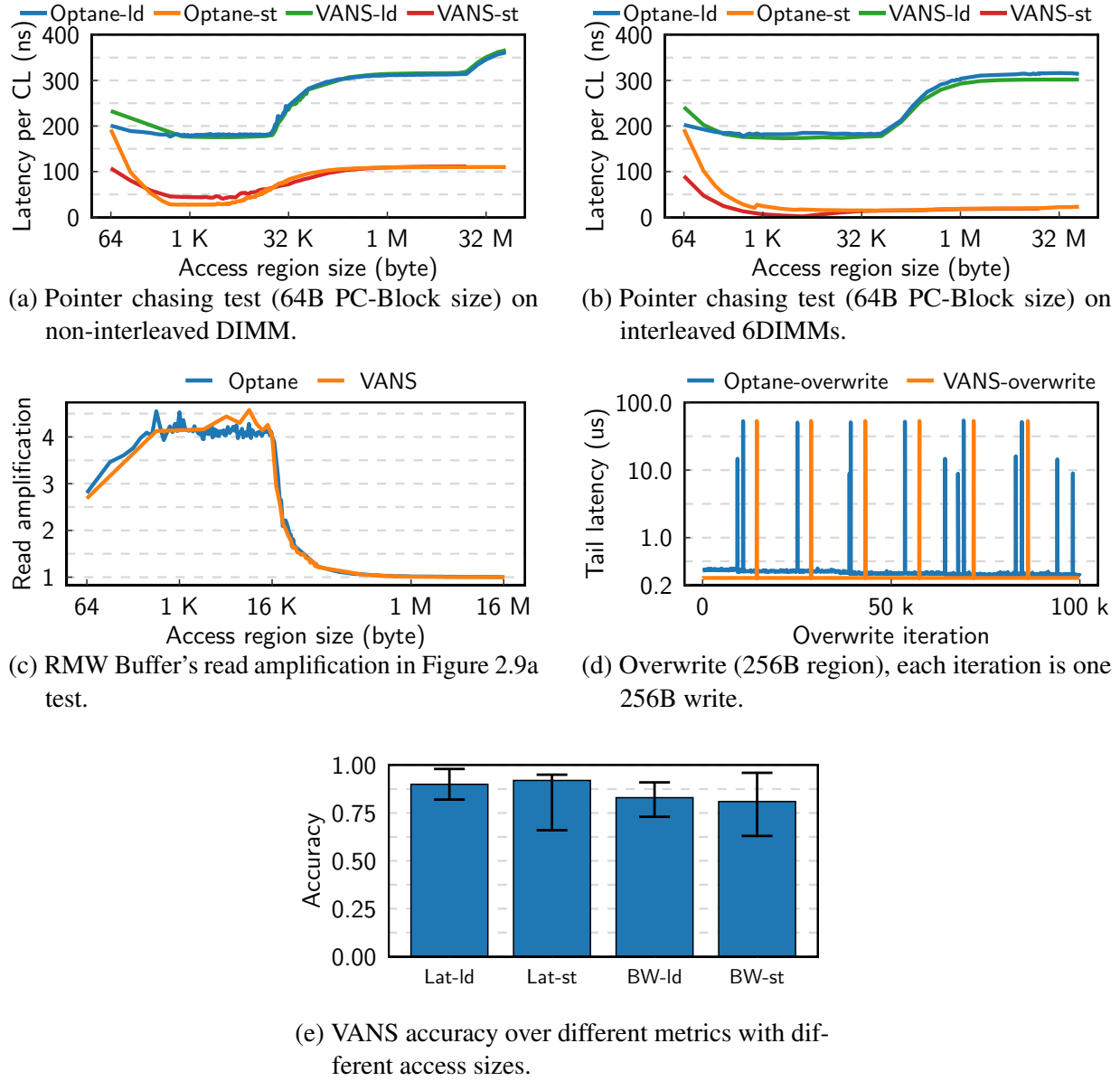


Figure 2.9: VANS performance validation with microbenchmarks: (a-c) pointer chasing and (d) overwrite. (e) VANS overall accuracy on load/store latency and bandwidth.

2.4.3 Validation with Microbenchmarks

We use microbenchmarks to validate the accuracy of VANS's architecture model. We catch memory traces of LENS microbenchmarks and feed them into VANS. We run VANS alone

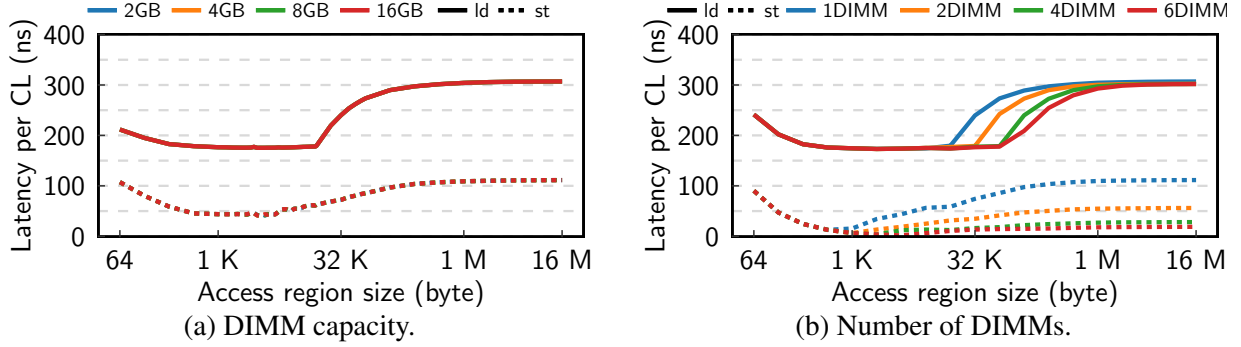


Figure 2.10: Sensitivity study of memory configurations.

in trace mode¹ and compare the simulated load/store latency, bandwidth, and read amplification with Optane DIMM-based real-machine results. Overall, our evaluation shows that VANS achieves an average 86.5% accuracy across four metrics (Figure 2.9e). Figure 2.9 (a-d) shows various performance curves generated by simulation and real-system profiling.

Figure 2.9a shows the load and store latency of the pointer chasing microbenchmark across different access region sizes with single-DIMM configuration. The store latency curves of VANS match the curves of real-system profiling well with a difference lower than 10% across all the access region sizes. The reason for the deviation of store latency at access region sizes smaller than 4KB (31.54%) is that small-sized write tests do not overflow the NVRAM buffers or queues on the real machine. Therefore, the store latency collected from real machine experiments is dominated by CPU on-core latency (e.g., the latency of executing `mfence`), which is not evaluated by this set of simulations without attaching a CPU model.

Figure 2.9b shows the pointer chasing latency of six DIMMs with interleaved configuration. We apply the 4KB-based interleaving scheme in VANS. Figure 2.9b shows an average 12% difference between simulation and real machine profiling.

Figure 2.9c validates the RMW Buffer read amplification in pointer chasing. We employ Intel’s in-house performance profile tool to investigate Optane DIMM’s read amplification in

¹We run VANS alone without `gem5`, because `gem5` does not support the non-temporal AVX512 instructions in the microbenchmarks.

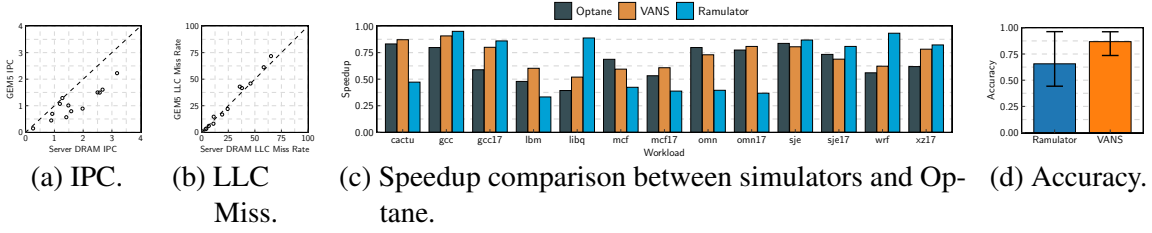


Figure 2.11: VANS performance validation with SPEC CPU 2006 and SPEC CPU 2017. (a,b) IPC and LLC miss ratio comparison between DRAM-based simulation and the server with DRAM main memory. (c) Speedup comparison between NVRAM-based simulation and the server with Optane DIMM main memory. (d) Accuracy (geometric mean).

RMW Buffer. We compare the results with the corresponding statistics generated by VANS. The difference across the curves is within 9%. Due to the limitation of Intel’s profiling tool, we cannot compare other hardware counters.

Figure 2.9d shows the tail latency for a 256B overwrite test. Each “iteration” performs a 256B write to a single NVRAM DIMM. The simulation matches the real-machine profiling in terms of tail latency length and interval.

We validate VANS accuracy with 4GB DIMM capacity (i.e., 4GB NVRAM media size), with a 256GB Optane DIMM media size. To study the impact of media size, we investigate pointer chasing benchmark on VANS with different media sizes (buffer and queue sizes remain unchanged). Figure 2.10a compares VANS-ld and VANS-st curves in Figure 2.9a (noted as ld and st in Figure 2.10a) with curves generated by various NVRAM media capacities. Media capacity does not affect the latency curves, because the media latency is hidden by on-DIMM buffers and queues. Figure 2.10b shows a sensitivity study on the number of DIMMs. With more DIMMs, the buffering effect on load latency is postponed, and the store latency reduces once memory access overflows the WPQ.

Table 2.4: Evaluated SPEC CPU benchmarks.

SPEC	Workload	LLC MPKI	Footprint
2006	gcc	2.9	1.2 GB
	mcf	27.1	9.1 GB
	sjeng	2.7	0.63 GB
	libquantum	3.4	2.3 GB
	omnetpp	2.1	1.4 GB
	cactusADM	2.0	2.2 GB
	lbm	7.7	2.9 GB
	wrf	2.4	1.0 GB
2017	gcc	21.5	1.1 GB
	mcf	26.3	8.7 GB
	omnetpp	2.1	0.96 GB
	deepsjeng	2.5	0.58 GB
	xz	2.7	1.8 GB

2.4.4 Validation with SPEC CPU Benchmarks

To further validate VANS, we compare VANS+gem5 full system simulation with real-machine profiling on SPEC CPU benchmarks [51, 52].

Benchmarks. We employ SPEC CPU 2006 [51] and 2017 [52] benchmark suites in our experiments. We first run all SPEC CPU benchmarks in speed mode on the Optane DIMM-based server machine. We use Linux perf [35] and Intel’s Emon [53] tools to profile each benchmark. Based on the profiling, we select the memory-intensive workloads that have at least two last-level cache (LLC) misses per thousand instructions (MPKI) for the rest of our experiments. Table 2.4 shows the selected benchmarks, as well as their LLC MPKI statistics and main memory footprints.

Simulator and Real Machine Configurations. We attach VANS to gem5 to conduct full system simulations. We modify gem5 to model the Cascade Lake microarchitecture [54] that we adopt in our server machine. Table 2.5 shows the details of our server machine configurations. We perform two sets of experiments on the server with (1) pure DDR4 DRAM DIMM main memory and (2) Optane DIMM-based NVRAM main memory, respectively. The Optane DIMMs

Table 2.5: Simulated system configuration.

CPU	
Core	4 cores, out-of-order, 2.2GHz
ROB-SQ-LQ	224-56-72 entries
L1 Cache	I\$ 32KB 8-way, D\$ 32KB 8-way, private
L2 Cache	1MB, 16-way, private
L3 Cache	32MB, 16-way, shared
TLB	L1I 128 entries 8-way, L1D 64 entries 4-way
WPQ	L2TLB 1536 entries (1024 d-entries, 512 i-entries)
	512B
DRAM Main Memory	
Configuration	DDR4 2666MHz 4-channel, 4GB/channel
Timing	tCAS(19) tRCD(19) tRP(19) tRAS(43)
NVRAM Main Memory	
Configuration	2666MHz 6-channel, 4GB/channel
Interleaving	4KB interleaving
LSQ	64 entries, 64B line
RMW Buffer	64 entries, 256B line
AIT Buffer	4096 entries, 4KB line
Internal DRAM	512MB DDR4 2666MHz
Operation Mode	AppDirect
Software	
Linux Kernel	v5.1.15
GNU gcc	v9.1

operate in *Memory mode* and are exposed as NUMA nodes [55]. Each workload is binded to a local Optane NUMA node. Our full system simulation adopts the same system configurations. The experiments on the DRAM-based server are used to evaluate the accuracy of our gem5 modification. To this end, we ensure the accuracy of the DRAM timing simulation attached to gem5 by modifying VANS to simulate DDR4 DIMMs with a formally verified DDR4 protocol timing model (Section 2.4.2). In experiments on NVRAM DIMMs, we configure VANS to have the same microarchitecture and parameters of Optane DIMM, based on industrial documents and LENS characterization observations (Section 2.3). To compare VANS with other memory simulators, we attach Ramulator [40] PCM model to gem5.

Results. Each simulation has two stages: (1) a warm-up stage that employs the gem5 simple CPU model to achieve a stable LLC MPKI; (2) an execution stage that executes at least two billion instructions on gem5 detailed CPU model. We first evaluate the accuracy of our gem5 modification

by DDR4 DRAM-based system experiments. We compare the execution stage instruction per cycle (IPC) and LLC miss rate of each benchmark between DRAM-based simulation and server profiling. Figure 2.11a shows that the accuracy of IPC is 61.2% on average (geometric mean). As we ensure the accuracy of our DRAM timing simulation by formal verification, the main reason for the accuracy loss is that we cannot fully implement Cascade Lake microarchitecture, due to the limited architecture details that can be modeled in gem5. Figure 2.11b shows a comparison of LLC miss rate (LLC miss / LLC total references). The average accuracy is 85.5%. Note that we turn off the cache prefetchers on both our server machine and simulation to avoid noises. Figure 2.11c shows a comparison of our NVRAM-based simulation with Optane DIMM server profiling. We calculate speedup as

$$Speedup(workload) = \frac{ExecTime_{DRAM}(workload)}{ExecTime_{NVRAM}(workload)}$$

VANS+gem5 achieves an accuracy of 87.1% on average (geometric mean) across various benchmarks, despite the less accurate CPU simulation model used. Ramulator (PCM)+gem5 only achieves an average accuracy of 65.6%.

2.4.5 Discussion

Simulator validation and verification methodologies. We adopt the same verification and validation methodologies used by various existing architecture simulators. DRAM simulators, such as DRAMSim2 [39] and Ramulator [40], also used the Micron’s verification model to verify their DRAM models. Our performance-based validation methodology is also adopted by several widely-used architecture simulators [56, 57]: Amber [56], an SSD simulator, is validated by comparing simulation results with SSD products on (i) microbenchmark latency and bandwidth and (ii) bandwidth, power, and system performance of real workloads. MGPUSim [57], a multi-

GPU simulator, employs (i) microbenchmarks to stress test each microarchitecture component and (ii) benchmarks, selected from two benchmark suites, to validate the performance accuracy of the simulator.

Modeling Other NVRAM DIMMs. VANS is initially developed to model Optane DIMM. But it can be flexibly adapted to other NVRAM DIMM architectures. To do so, users first need to run LENS on the target NVRAM system to obtain insights on its microarchitecture design, e.g., internal latency and buffer structures. Then, the users can reconfigure VANS based on the new parameters and microarchitecture design. Our modular software design allows users to add additional architecture components.

2.5 Cloud Workload Optimizations

The advantages of capacity and persistence make NVRAM a natural fit for memory-hungry cloud applications [58]. In this section, we use VANS to study the use of NVRAM for cloud applications. We identify the performance overhead due to applications' memory access patterns and alleviate the overhead with two architecture optimizations.

2.5.1 Performance Inefficiency

To understand the NVRAM architectural behavior when running cloud applications, we run two widely used cloud workloads, Redis [59] and YCSB [60], in VANS+gem5 full system simulation with the configuration in Table 2.5. The results show that the pointer chasing access patterns and the wear-leveling triggered write amplification are the two primary sources of inefficiency.

Notable Read Misses. Figure 2.12a shows that read operations dominate the execution overhead in Redis. Cycles Per Instruction (CPI) of read operations is $8.8\times$ higher than CPI of

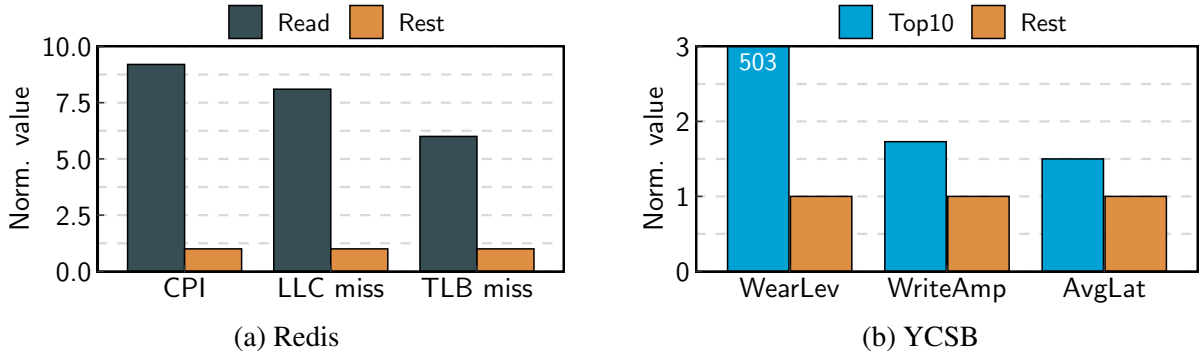


Figure 2.12: Redis and YCSB profiling. Each metric value is normalized to the corresponding *Rest* case. “Top10” stands for the ten most frequently written cache lines.

other workload activities. The major overhead originates from reads leading to misses in LLC and TLB, due to the pointer chasing memory access pattern that repeatedly shifts between random memory regions. Such a pattern is commonly observed in data structures that are often used in cloud workloads, such as B-Trees and hash tables [61–63].

Write Amplification Overhead. Figure 2.12b shows that most YCSB writes are concentrated in ten cache lines, which are written over $100\times$ more than the total of other cache lines. Compared with accesses to other cache lines, the writes to the Top10 cache lines, which form only 15% of total memory traffic, trigger $503\times$ more wear-leveling operations. These operations lead to higher write amplification than others and elevate average memory access latency.

2.5.2 In Memory Pre-Translation

To address the read miss induced overhead, we propose *Pre-translation* to optimize the address translation in the read operations. We observe that the NVRAM performs a “page translation” within the NVRAM DIMM that translates the physical address to the media address². To allow the CPU to concurrently receive data for the current memory access and the TLB entry for the next memory access, Pre-translation combines the NVRAM translation with the CPU

²“Physical address” refers to the address translated by MMU. The “media address” refers to the real physical address used in the NVRAM media, which is transparent to CPU memory controllers.

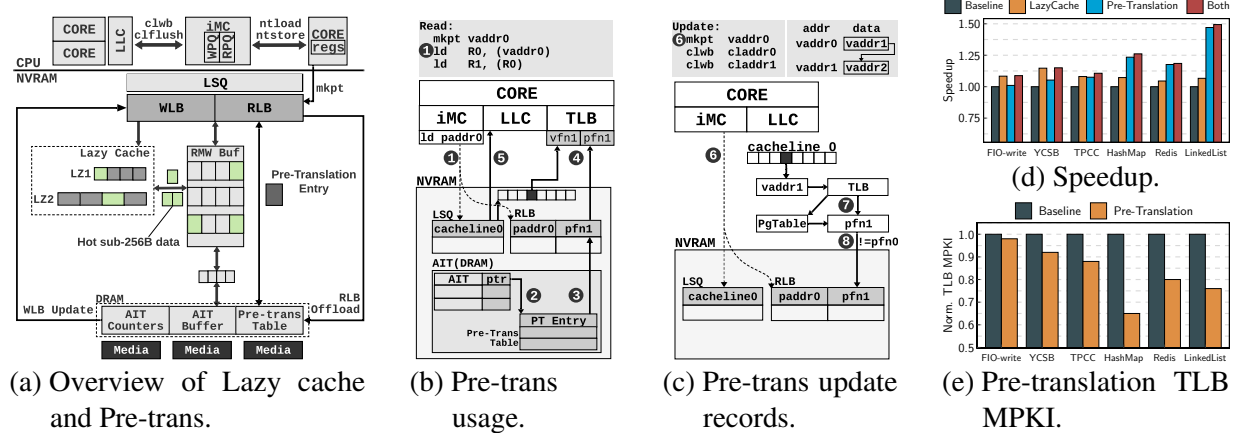


Figure 2.13: Customized Lazy cache (a) for write intensive workloads and Pre-translation (a–c) for pointer chasing read intensive workloads. Baseline in (d–e) is measured without any optimization mechanism.

translation. As shown in Figure 2.13a, Pre-translation consists of two hardware structures – the Pre-translation table and the Read Lookaside Buffer (RLB) – and a new instruction `mkpt`.

The Pre-translation table is stored in the on-DIMM DRAM as a part of an AIT entry [44]. This table maps a physical address (`paddr`) to a page frame number (`pfn`) pointed to by `paddr`. In practice, it stores only the `pfn` and employs `paddr` as an index; this allows the integrated DRAM to store more Pre-translation entries. To take advantage of the existing translations in AIT, we add a pointer to each AIT entry, so that it takes only one more DRAM access to find the corresponding Pre-translation table entry (Figure 2.13b).

Similar to the TLB, RLB buffers the Pre-translation table and is stored in SRAM. Each RLB entry consists of a page frame number from a Pre-translation table entry and a physical address that is used as the index to the table entry.

The new instruction `mkpt` is used to update Pre-translation table. This instruction takes one virtual address as an argument and hints the memory controller to read or update the Pre-translation entry. To use this instruction, programmers or compiler need to insert it before load and store instructions that lead to a pointer chasing access.

A Pre-translation Involved Example. We use an example of a linked list traversal to demonstrate the detailed hardware operations in the Pre-translation, as shown in Figure 2.13b and Figure 2.13c. In this example, the traversal currently visits the node stored in address `vaddr0` (top right Figure 2.13c), and generates a pointer chasing access.

To use Pre-translation (Figure 2.13b), `mkpt` is invoked to “mark” a read access to `vaddr0` (❶). When NVRAM receives the read request with this mark, it checks the RLB or Pre-translation table for an entry, using `paddr0` as an index (❷). If the Pre-translation entry is found (❸), NVRAM uses the data at `paddr0` and page frame number `pfm1` from the Ptr-translation entry to create a TLB entry (❹); this entry is sent to the CPU’s TLB when returning the requested data (❺). In this case, the CPU receives the data at `vaddr0` and the TLB entry for the next access to `vaddr1` simultaneously.

The Pre-translation table is updated by `mkpt` (Figure 2.13c): each time when `mkpt` is invoked with an address `vaddr0` (❻), the CPU checks if the corresponding Pre-translation entry is missing or out-of-date. If so, the CPU updates the Pre-translation entry: it first translates `vaddr0` to `paddr0` and fetches the data at this address, then it uses this data as a virtual address (`vaddr1`) to find out the page frame number `pfm1` (❼). Finally, the CPU writes the entry `<paddr0, pfm1>` to RLB, if `pfm1` is not equal to `pfm0` (❽).

Validating Pre-translation Entry. Pre-translation entries may become stale, when the CPU updates the page table. To reverse the effect of stale TLB entries, fetched from NVRAM, we propose a “check-before-read” mechanism, which relies on the existing NVRAM memory request queue in the CPU’s memory controller³. We extend each entry in the NVRAM request queue with a new “uncertain” bit. When CPU gets a TLB entry from the NVRAM, the memory management unit (MMU) uses this entry to translate the virtual address and issues a read request to NVRAM. This read request enters the NVRAM request queue with the uncertain bit set. In the meantime,

³According to Intel’s announcement [45], CPU and NVRAM work in a request/grant scheme. This indicates that the CPU maintains a memory request queue in iMC for NVRAM accesses.

the MMU starts an asynchronized page table walk to determine whether this TLB entry from NVRAM matches the latest record in the page table. If this TLB entry is stale, the memory controller updates the corresponding NVRAM read request with the latest translation info, issues the updated read request to NVRAM, and sets the uncertain bit to 0. If the TLB is up-to-date, then the memory controller just resets the uncertain bit to 0. The data is not fetched from NVRAM if the corresponding uncertain bit is 1, so the CPU never gets a wrong cache line caused by a stale TLB entry.

2.5.3 Lazy cache

To reduce the write amplification in cloud workloads, we add a Lazy cache to NVRAM (Figure 2.13a) to cache the frequently written data. It adopts a 2-level inclusive cache, LZ1 and LZ2, with 64B and 128B granularity, respectively. It employs a Write Lookaside Buffer (WLB) to store the addresses of Lazy cache entries.

We reuse AIT’s wear-out record to update Lazy cache: once a write triggers wear-leveling, AIT calculates the Lazy cache priority during the data migration; it requires Lazy cache to cache the incoming write to this location, if the priority is higher than a certain threshold. This Lazy cache update does not exacerbate the performance and storage overhead, as AIT migration is significantly slower than the Lazy cache access and we reuse the existing AIT records.

Lazy cache relies on the existing ADR [32] to ensure data persistence in the face of power outages. In Section 2.5.4, we show that a small Lazy cache (3KB) is efficient in improving performance and reducing memory traffic. The 3KB Lazy cache is much smaller than other on-DIMM buffers (16KB and 16MB). Thus ADR is sufficient to ensure its persistence.

2.5.4 Evaluation

To evaluate the performance of Pre-translation and Lazy cache, we use fio [64], two PMDK [65] microbenchmarks (HashMap and LinkedList), and three cloud workloads from WHISPER benchmarking suite [66] (Redis, TPCC and YCSB). Table 2.5 lists our simulation setup. We run unmodified workloads without any architectural optimization to obtain the baseline performance.

To compare with baseline, we configure (1) Pre-translation with a 1KB RLB and a 16MB Pre-translation table and (2) Lazy cache with a 1KB L1 cache and a 2KB L2 cache. We modify the workloads' source code to use Pre-translation, and use unmodified workloads on Lazy cache.

Figure 2.13d shows the evaluation results of Pre-translation, Lazy cache, and a combination of both. Pre-translation achieves 1% to 48% speedup across all six workloads. It efficiently optimizes TLB misses to achieve this speedup: as shown in Figure 2.13e, Pre-translation reduces the TLB misses per thousand instruction (MPKI) by 17% on average. Lazy cache achieves an average of 10% speedup across all six workloads. It achieves this by detecting memory patterns that trigger wear leveling on a small region, and caching these writes accordingly, thereby reducing the write amplification overhead. Using both Pre-translation and Lazy cache achieves 8% to 49% speedup across all workloads. It does not achieve the ideal speedup, because enabling both optimizations leads to higher memory bus contention.

Overall, Pre-translation optimizes pointer chasing, as it reduces the read operation overhead as well as the TLB miss rate. Lazy cache is able to reduce the write operation overhead and write traffic. The combination of these two mechanisms benefits cloud workloads running on real NVRAM-based system under various settings.

2.6 Related Work

To our knowledge, this is the first paper to unveil the architecture details of real NVRAM DIMM products in server systems and develop an architecture-level simulation model for the device. This section discusses related works.

NVRAM Emulation and Simulation. Besides PMEP [19] and Quartz [37], previous works also explored using FPGA to emulate NVRAM, e.g., Lloyd et al. [67] proposed an FPGA emulator to model the latency projected for NVRAM. However, most of the emulators model NVRAM by injecting delays to mimic the NVRAM latency. Previous memory architecture simulators used to model NVRAM, such as NVMain [68], DRAMSim2 [39], and Ramulator [40], model NVRAM based on conventional DRAM DIMM architecture and timing. However, as discussed in Section 2.2, these emulators and simulators do not fully model the performance and microarchitecture characteristics of Optane DIMM.

Optane DIMM System Studies. Most of the existing Optane DIMM system studies focus on system-level performance profiling and system software design. Recent works observed inefficient performance on Optane DIMM by running the existing NVRAM-aware system software with certain access patterns. For example, several studies [1, 69, 70] observed that repeated writes to a concentrated Optane DIMM memory area lead to high write latency. FlatStore [69] addresses this issue by grouping small-sized writes and evenly spreading writes to multiple Optane DIMMs. To reduce the overhead caused by the long write latency, MOD [71] proposes a framework to build persistent data structures with minimal ordering instructions. Other studies observed that concurrent or mixed reads and writes to the Optane DIMM in a remote NUMA node lead to performance degradation with high-performance computing and storage workloads [17, 18, 70]. In addition, multi-threaded memory accesses do not scale well on Optane DIMM systems [1, 14, 16, 69]. A recent study [70] shows that one reason behind this issue is the contention in

the WPQ and RMW buffers. Our study shows that the contention in the AIT Buffer and the LSQ exacerbates this scaling issue. Recent work [25] investigated two data structure design strategies for RocksDB: Memtable that generates small random accesses and FLEX that generates sequential accesses. DRAM-emulated NVRAM experiments show that Memtable is 19% faster than FLEX; but experiments on Optane DIMM show that FLEX is 10% faster than Memtable [25]. Our study indicates that one main reason is that the small random accesses of Memtable do not fully utilize the buffer capacity and access granularity. Existing NVRAM-aware system software design may also lead to high read and write amplification on Optane DIMM [1, 70]. However, these studies only show the inefficiency from the system software design perspective. Our study indicates that one main reason for the amplification is the large on-DIMM buffer access granularity.

Pointer Chasing Optimizations. Jump pointer [72] proposes a data prefetching scheme for linked data structures. It requires a new data field in each data structure. Moreover, it does not reduce the TLB miss overhead. Our Pre-translation reduces TLB miss overhead without requiring modification on data structures. ASAP [73] proposes a prefetcher for page table walks to reduce the execution time for each TLB miss. Pre-translation also reduces the TLB miss rate.

2.7 Conclusion

We design LENS and VANS, a low-level performance profiler for NVRAM systems and an architecture-level memory simulator that models the recently released Optane DIMM memory system. Using LENS, we perform a detailed characterization of the Optane DIMM microarchitecture design. LENS allows users to characterize the performance and reverse engineer the architecture design of real NVRAM DIMM systems. VANS allows researchers who do not have access to Optane DIMM physical devices to explore and evaluate new design ideas on architecture and systems. Furthermore, both LENS and VANS are flexible to be used with other

NVRAM systems beyond the current Optane DIMM design.

2.8 Acknowledgements

Chapter 2, in full, is a reprint of the material as it appears in IEEE/ACM International Symposium on Microarchitecture, 2020. Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, Jishen Zhao. “Characterizing and Modeling Non-Volatile Memory Systems”. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Building Scalable Systems on Heterogeneous Architecture

3.1 Introduction

Deep learning (DL) is a key enabler of tremendous breakthroughs in tasks ranging from object detection [74] to speech recognition [75, 76] and language translation [77]. As DL models evolve, the model size has been continuously increasing to adapt to complex tasks and improve accuracy [78, 79]. The size of training data also increases to achieve higher accuracy [80–82].

A common practice to train a large DL model with a large amount of input data is distributed DL training on multiple workers (computational devices such as GPUs or TPUs [83, 84]). These workers are optimized for high-performance matrix multiplication, which is heavily used in DL training. However, distributed DL training is increasingly constrained by (1) single device parallelism and on-device memory capacity and, (2) cross-device communication overhead [85–87].

On the one hand, the state-of-the-art GPUs and TPUs use tens of Gigabytes of on-device high-performance memory [88, 89], which is an order of magnitude smaller than the system

memory. The on-device memory capacity does not evolve as fast as DL model sizes, leading to the popularity of data parallelism [90], model parallelism [91], and pipeline parallelism [85, 87, 92]. Although parallelism schemes enable the training of large DL models, they inevitably introduce additional code complexity and trade-offs without always leading to training speedup [93]. Another solution is to build extended parameter storage with system memory and storage [94]. But this only benefits a subset of DL models with certain parameter characteristics (e.g., parameter data density) [94].

In addition, the bandwidth of commodity cross-device communication, including PCIe [95], NVLink [96], and the network, is an order of magnitude lower than on-device memory bandwidth. This imposes a major performance bottleneck when adding more worker devices to a distributed training system. As discussed in Section 3.2.2, the overhead due to cross-device communication is up to 76% of total DL training time, significantly degrading the device utilization and training performance [85, 86]. Recent efforts are devoted to decentralized training with the message passing interface (MPI) to mitigate the communication bottleneck [97, 98]. However, MPI creates a synchronous point that forces the faster workers to wait for the slower ones, hence degrading the computation utilization of worker devices [99].

The emerging cache-coherent interconnection (CCI) [100, 101] promises new opportunities in addressing the memory capacity and communication challenges. CCI allows the memory to be attached to serial buses (e.g., PCIe) instead of only the memory bus (DDR channels). As such, system memory capacity is no longer constrained by the memory channels supported by the processor sockets. CCI protocols typically provide hardware-level cache coherence support and customized protocols to reduce memory access latency. With CCI, host processors can directly issue memory load/store instructions to memory devices on a serial bus without using a software driver. CCI also enables disaggregated memory systems, where each memory device incorporates an on-device processor with on-device memory. The on-device processor can access the host CPU’s memory through CCI [100, 101].

However, a naïve design, which simply offloads parameter synchronization to the CCI memory devices, falls short to enable efficient large-scale DL training due to critical design challenges. First, parameter data transfer is bounded by a single memory device’s serial bus bandwidth; coherence traffic also increases with the number of computation devices sharing the same memory region, reducing the bandwidth available to accommodate parameter data transfer. Second, both parameter size and local/remote communication bandwidth characteristics are not uniformly distributed in different distributed systems; in addition, there is a non-uniform bandwidth and latency demand with different parameter sizes in DL training. As a result, no single partitioning and configuration of DL operations fit all.

Our goal in this paper is to enable efficient large-scale DL training by designing a parameter synchronization scheme that addresses the previously described challenges. To this end, we propose a Cache cOherent interconnected pARameter SErver (COARSE), which is built on top of collective communication for synchronization similar to that of MPI-based decentralized training. In particular, COARSE consists of three key design principles. First, we propose a decentralized parameter communication scheme to decentralize parameter synchronization and localize parameter storage. Second, we propose a tensor routing and partitioning scheme, which exploits parameter granularity and non-uniform interconnection bandwidth to fully utilize serial bus bandwidth and improve tensor locality in the GPUs. The tensor partitioning enables a pipelined tensor synchronization and takes advantage of serial bus bi-directional bandwidth; The bandwidth-aware tensor routing routes a GPU’s tensor to a bandwidth-friendly memory device, even if they are not under the same serial bus switch. Finally, we develop a dual parameter synchronization scheme to reduce parameter synchronization traffic and enable high GPU computation utilization.

Internally, COARSE handles parameter synchronization with a set of collective communication based on CCI protocol – similar to that of MPI-based decentralized training – to reduce the communication latency and improve the synchronization bandwidth. Yet to allow easy integration

with the commodity DL training frameworks, we provide a parameter push/pull interface similar to that of conventional parameter servers.

This paper makes the following key contributions:

- We propose COARSE, an efficient parameter synchronization scheme that accelerates distributed DL training by leveraging disaggregated memory systems.
- We identify non-uniform local and remote bandwidth distribution in different distributed systems.
- We build a COARSE prototype integrated with TensorFlow, offering significant improvements on training latency, computation utilization, and interconnection bandwidth utilization in various distributed training workloads running in different cloud systems.

3.2 Background and Motivation

DL training has witnessed a wave of rapidly increasing scales of data sets and models. This has led to the popularity of distributed training to scale-out training workloads on distributed server nodes (*workers*), each with an increasing number of interconnected GPUs (Figure 3.1). As a result, the data communication overhead is becoming one of the key obstacles for the continuous scaling of distributed DL training [85, 86]. In this section, we motivate our design by discussing the challenges and opportunities of enabling efficient parameter communication with cache-coherent disaggregated memory systems.

3.2.1 DL Training Parallelism Schemes

Modern large DL models and data sets are too large to fit in a single worker while achieving the target accuracy within a reasonable period of time [102]. This leads to the exploration of various classes of DL parallelism schemes to train a DL workload on multiple server nodes – data, model, pipeline, and hybrid parallelism [103] – without an obvious performance winner.

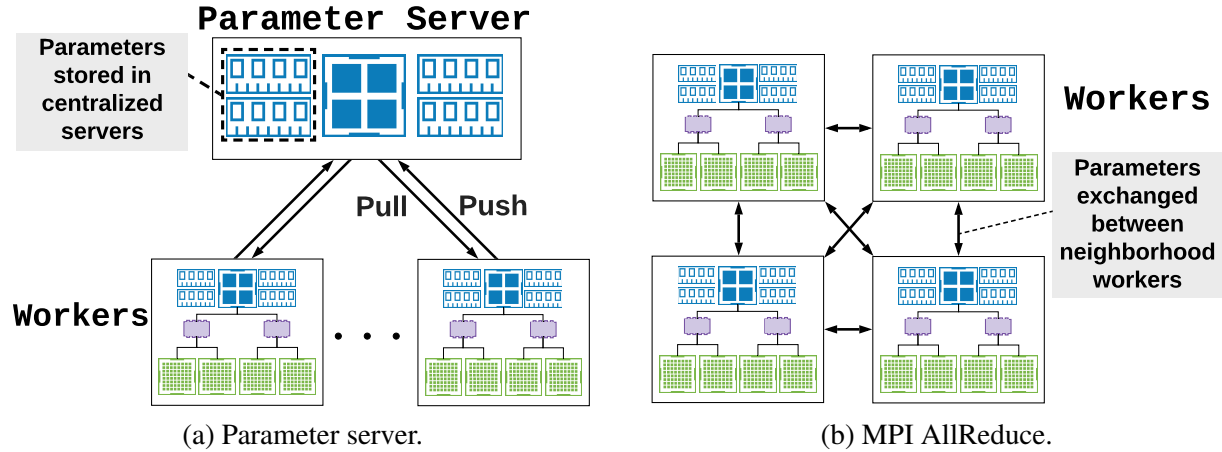


Figure 3.1: Communication schemes in distributed DL training.

In this paper, we focus on data parallelism due to its popularity in commodity systems and the opportunity introduced by the emerging cache-coherent disaggregated memory systems (Section 3.2.3). The major advantage of data parallelism is that it is applicable to any DL model without further domain knowledge of the model.

Data Parallelism. Data parallelism is widely used in state-of-the-art commodity DL training systems [90]. Rather than training a model with all the available input data on a single server, the system replicates the model across many workers and feeds each replica with a subset of the input data. Because the model replicas are trained using different input data, their parameters will typically diverge.

Parameter Server. To reconcile these parameters and ensure that all model replicas eventually converge, each replica periodically pushes its local parameter values to a centralized server, i.e., a parameter server. Parameter server is typically implemented as a distributed key-value store with consistent shared metadata, including server status and/or a hash table with distributed key-value mapping. The parameter server aggregates all the received updates for each parameter – e.g., by averaging them – and then sends back to all replicas a newly computed set of values, which will be used at the beginning of the next training iteration. However, not only the data but also the

model size keeps skyrocketing. For instance, DLs used in recommendation systems or natural language processing (NLP) tasks commonly have large network parameters, e.g., BERT [104] model has ~ 300 million parameters. To accommodate the huge number of parameters in large DL models, systems employ multiple parameter servers, with each one being responsible for a subset of the parameters.

Model Parallelism. Model parallelism partitions the operators in a training model across multiple workers; each worker evaluates and updates a subset of the model parameters for all inputs. However, model-parallel training suffers from computing resource under-utilization due to data dependency and the large communication overhead of intermediate results, such as activation and gradient [85].

Pipeline Parallelism. Pipeline parallelism partitions DL operations and splits a mini-batch into multiple micro-batches. Each worker computes the output of a model partition with a set of micro-batches, then propagates the output to the subsequent workers. By executing multiple micro-batches in parallel and making communication traffic point-to-point (stage to next stage), pipeline parallelism improves worker utilization compared to model parallelism [85, 87, 92]. However, such parallelism schemes lead to either low computing resource utilization [87] or accuracy loss [85], depending on their implementations.

3.2.2 Communication Issues in DL Training

Data parallelism imposes severe communication and synchronization overhead due to back-propagation updates. Such communication typically relies on two types of training system architectures (i) *centralized* communication (Figure 3.1a) – commonly used for inter-node communication between the parameter server and workers and (ii) *decentralized* communication (Figure 3.1b) – commonly used for intra-node communication between worker GPUs [105, 106].

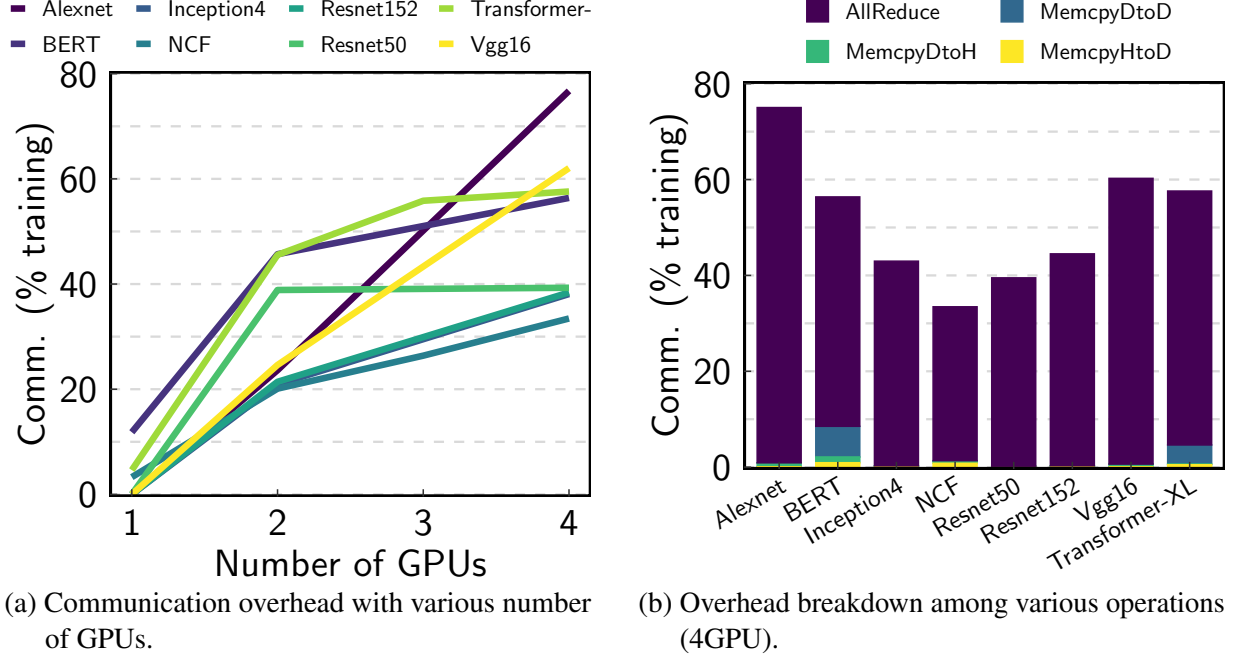


Figure 3.2: Communication overhead in ML training.

In centralized communication, the computed gradients in each worker need to be transferred back to the parameter server for model updates, before continuing to compute the next mini-batch of training data. A parameter server provides a push/pull interface for each worker. As a result, a critical performance bottleneck is the parameter server communication bandwidth. In decentralized communication, workers directly communicate with each other via an MPI interface, such as `AllReduce`.

Recent studies show that data communication among workers is a major bottleneck in distributed DL training [85, 86]. We further evaluate the communication overhead in decentralized communication with eight DL models implemented by TensorFlow [107] and NVIDIA [108], as illustrated in Figure 3.2. We use NCCL [97], which is an optimized decentralized communication scheme using the MPI `AllReduce` interface. We train the models on an AWS EC2 `g4dn.12xlarge` instance [109] with various numbers of GPUs, and employ `nvprof` [110] to measure the communication overhead in terms of the time spent on communication operations, including CUDA memory copy operations and NCCL `AllReduce`. Figure 3.2a shows that

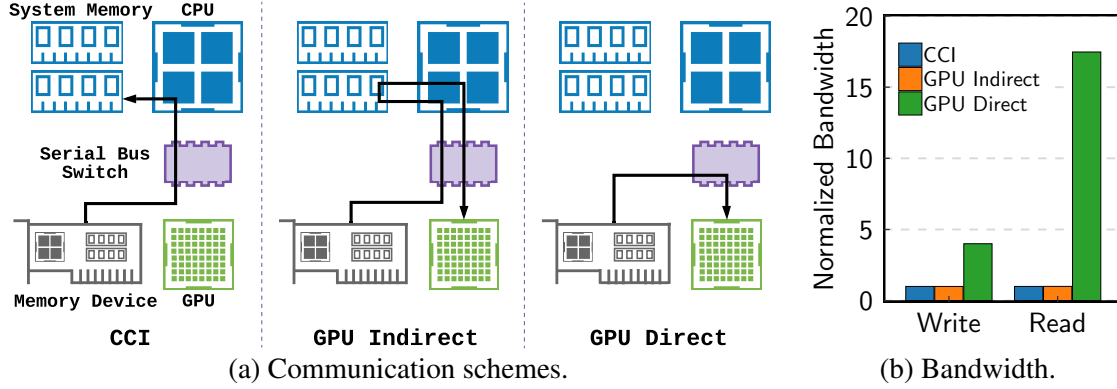


Figure 3.3: CCI bandwidth with CPU load/store instruction (*CCI*), GPU indirect copy through CPU (*GPU Indirect*), and GPU direct peer-to-peer copy (*GPU Direct*). Read and write bandwidth are normalized to the *CCI* bandwidth, respectively.

an increasing number of GPUs leads to a higher percentage of communication overhead, with up to 76% of training time spent on data communication using four GPUs. Figure 3.2b shows that `AllReduce` dominates the communication latency overhead among all the operations that perform communication.

The key bottleneck with centralized communication in a single server node is the limited serial bus lanes in the CPU, which constrains the service throughput of GPU requests on parameters: In a multi-GPU server node, the number of serial bus lanes on CPUs is typically smaller than the total serial lanes required by all the GPUs. As a result, the concurrent access from GPUs to the parameter server is constrained by the CPU serial bus lanes.

The key issue with decentralized communication is that GPU computation is blocked during parameter synchronization: First, a parameter synchronization operation blocks all GPUs involved for synchronization, forcing the fast GPUs to wait for the slower ones [99, 111]. Second, the `MPI AllReduce` approach generates more communication requests than centralized training, depending on the topology adopted. Third, using a commonly adopted ring topology, the `AllReduce` performance is bounded by the lowest device-to-device bandwidth, leading to bandwidth utilization, e.g., as low as 34% on NVIDIA DGX-1 systems [86].

3.2.3 Opportunities With Disaggregated Memory Systems

Cache-coherent interconnection (CCI) protocols, such as CCIX [112], Gen-Z [100], and Intel CXL [101], enable cache-coherent disaggregated memory systems. By offering a CPU-transparent hardware coherence support, CCI allows the CPU to directly issue load and store instructions to access multiple memory devices attached to the existing or customized serial buses. These memory devices form a disaggregated memory pool: each memory device adopts an on-device processor and a large capacity on-device memory; the memory devices map their local memory into CCI-unified memory address space and share the address space with the host CPU and other memory devices [100, 112]. CCI also allows the host CPU to distribute memory-intensive computation jobs to the processors on memory devices, schedule the jobs, and collect computed results.

CCI Performance Benefits. CCI reuses existing serial bus physical layer protocol and builds a customized higher-level protocol stack [101]. This customized protocol focuses on improving fine-grained memory access latency while providing sufficient peak bandwidth (e.g., 90% of underlying serial bus peak bandwidth [101]). Thus, compared to existing serial bus protocols, e.g., PCIe [95, 113, 114], CCI can achieve higher bandwidth when accessing small memory blocks. This unique feature allows GPUs to seamlessly access external large-capacity disaggregated memory devices through CCI.

Promising CCI-based Peer-to-peer (P2P) Communication. We identify that peer-to-peer accesses to CCI-based disaggregated memory achieve significantly higher bandwidth than indirect access through the host CPU.

We evaluate the bandwidth of an FPGA-based disaggregated memory prototype implemented on industrial CCI protocol (more details in Section 3.4.3) under three use cases: (i) copying data from a disaggregated memory device to CPU memory (*CCI*). (ii) copying data from

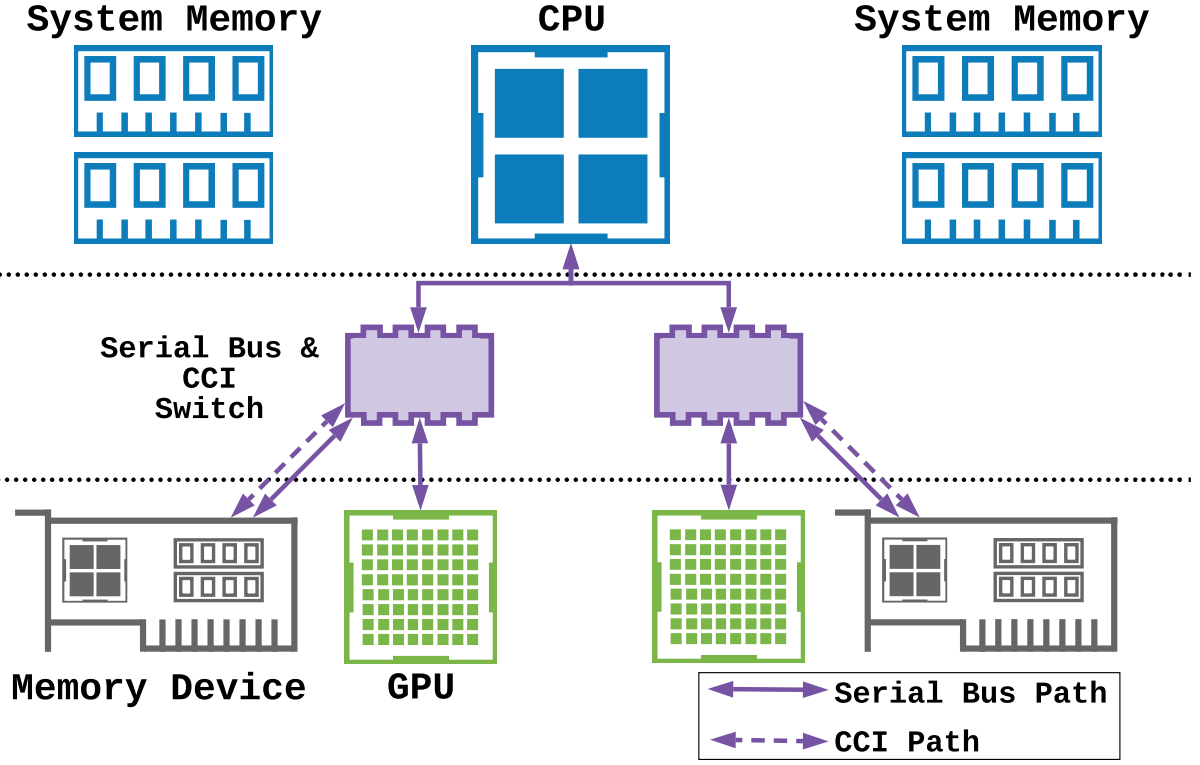


Figure 3.4: System deployment of a single server node.

disaggregated memory to CPU then to GPU memory (*GPU Indirect*). (iii) directly copying data from disaggregated memory to GPU memory (*GPU Direct*). Figure 3.3 shows that GPU direct peer-to-peer copy provides $17\times$ read bandwidth and $4\times$ write bandwidth speedup. This motivates us to enable GPU direct access to CCI memory for better performance.

Benefits to Data Parallel DL Training. CCI is beneficial to data parallel training in two aspects: (i) in data parallel training, cross-device communication is a major overhead; this communication can take advantage of CCI low-latency memory access to improve the parameter synchronization performance. (ii) the parameter synchronization operations block the training procedure and take up GPU computing resources; while using CCI, GPU can work with memory device processors coherently to offload these synchronization operations to memory device processors, thus improving the GPU utilization and reducing the communication overhead.

3.3 COARSE Design

Overview. Our goal in this paper is to enable efficient large-scale DL training by designing a parameter synchronization scheme with (1) light communication traffic, (2) high interconnection bandwidth utilization, and (3) high GPU computation utilization. To achieve our goal, we propose Cache cOherent interconnected pARameter SErver (COARSE), which is a decentralized parameter synchronization design offloaded to CCI-based disaggregated memory systems. COARSE provides a parameter server push/pull interface for easy integration, while internally, it consists of the following novel parameter synchronization mechanisms:

- **A decentralized parameter communication scheme** to decentralize and localize parameter communication.
- **A tensor routing and partitioning scheme** which exploits non-uniform interconnection bandwidth characteristics to fully utilize serial bus bandwidth and improve tensor locality in GPUs.
- **A dual parameter synchronization scheme** to reduce parameter synchronization traffic and enable high GPU computation utilization.

This section presents the key ideas of COARSE design. Implementation details will be described in Section 3.4.

3.3.1 System Deployment

We assume a single server node deployment illustrated in Figure 3.4: GPUs are connected to serial bus switches and communicate with other serial bus devices as in commodity systems [83]. Each GPU is paired with a CCI-based disaggregated memory device (referred to as *memory device* in the rest of the paper) sharing the same serial bus switch, such that communication between a pair of GPU and memory device achieves full bandwidth. A similar approach in existing systems is to put a network card under each PCIe switch to improve GPU RDMA bandwidth [83]. Each

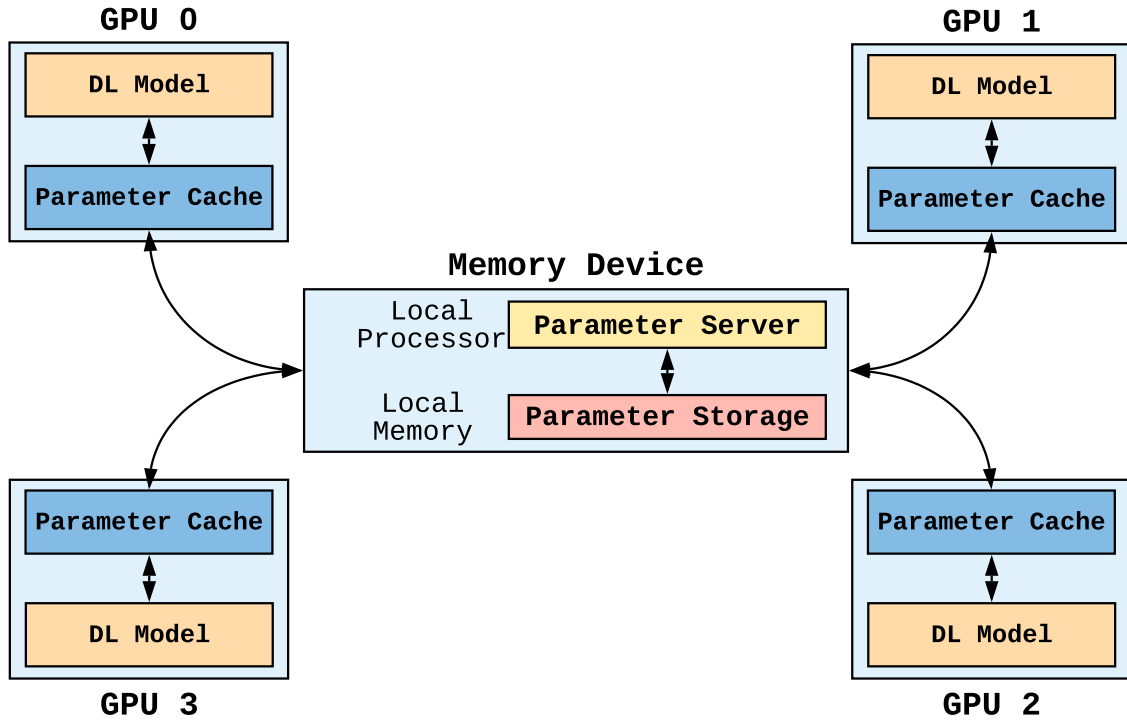


Figure 3.5: DENSE architecture.

memory device incorporates an on-device processor (e.g., with ARM cores) and a large-capacity memory. Intra-node memory devices are interconnected by a CCI. CCI protocols allow the memory devices to support (i) full-bandwidth data copy through a serial bus interface (the solid lines) and (ii) cache-coherence traffic through a CCI interface (the dashed lines) [100, 101]. As such, serial bus devices can copy data peer-to-peer through a serial bus switch or host bridge [115], bypassing data copy in system memory.

3.3.2 A Naïve Design

We assume a baseline architecture – *DENSE* – as shown in Figure 3.5. In DENSE, each GPU maintains a parameter data cache supported by CCI. To update parameters, the GPU first updates the local parameter cache and then employs CCI to update the global parameter coherently. The global parameters are stored in a memory device, which runs a parameter server

on the on-device processor and stores the parameters in on-device memory. The memory device maps the on-device memory to CCI address space and shares it with other GPUs through CCI. This architecture bypasses the host CPU data copy, and we use it as a baseline in the following sections.

In DENSE, parameter synchronization jobs are offloaded to the processors in memory devices. Such offloading introduces promising benefits for DL training: Compared to conventional centralized training, the offloading allows GPUs to directly communicate with memory devices at full serial bus bandwidth, without being limited by CPU serial bus lanes; Compared to decentralized training, the offloading allows GPUs to continue with computation during parameter synchronization, substantially improving GPU utilization.

3.3.3 A Disaggregated Design

Simply offloading parameter synchronization to the memory devices in DENSE falls short to enable efficient large-scale DL training. In the following sections, we identify critical design challenges and insights obtained from our experiments in two different distributed system infrastructures, from Amazon AWS [109] and San Diego Supercomputing Center (SDSC) [116], respectively. Based on our observations, we propose a disaggregated parameter synchronization architecture, *COARSE*, as illustrated in Figure 3.6.

In COARSE, the parameter synchronization service is further disaggregated to multiple memory devices, providing higher communication bandwidth compared to DENSE’s centralized architecture. Each memory device communicates with one GPU sharing the same serial bus switch for parameter updates, and communicates with other memory devices for parameter synchronization. The following sections introduce the design details of COARSE.

3.3.4 Decentralized Parameter Synchronization

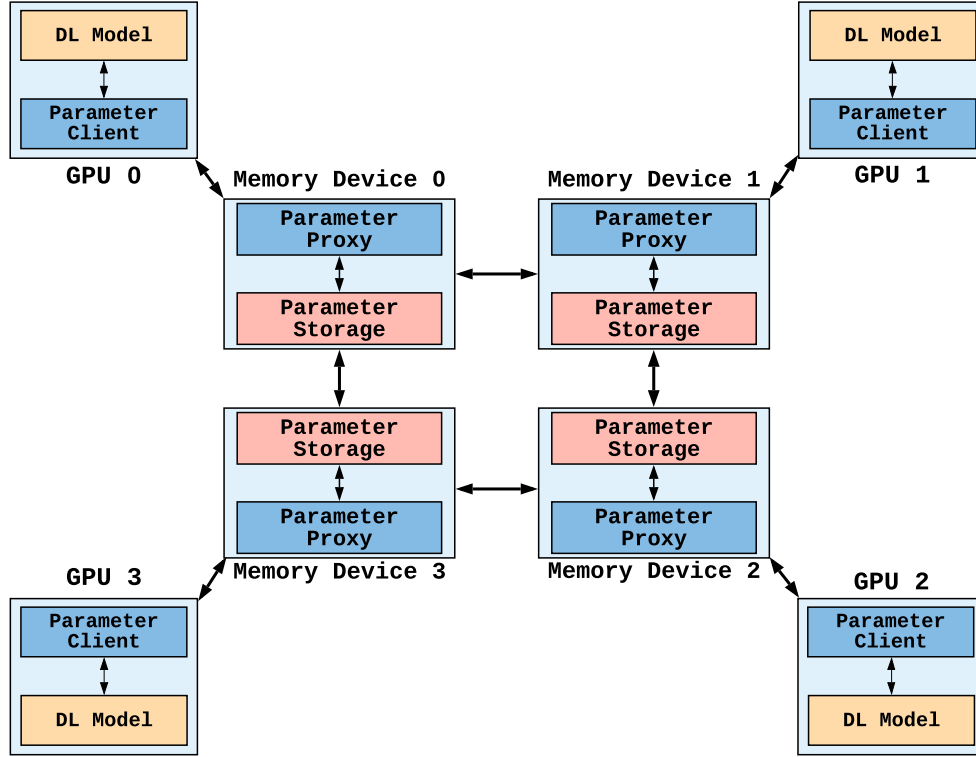


Figure 3.6: COARSE architecture.

Interconnection Bandwidth and Scalability Issues. Although disaggregated memory system allows multiple computation devices (e.g., GPUs) to share memory at low latency, parameter data transfer is bounded by serial bus bandwidth in the DENSE architecture. Such bandwidth limitation will constrain the number of computation devices sharing the disaggregated memory system. Furthermore, coherence traffic also increases with more computation devices sharing the same memory region [100, 101], reducing the bandwidth available to accommodate parameter data transfer on the CCI path.

Decentralized Parameter Communication. Our key idea to address the above challenges is to localize parameter storage and decentralize parameter synchronization. To this end, we propose a decentralized parameter synchronization scheme, which splits the original synchronization functionality between parameter proxy and storage.

Figure 3.7 shows an overview of our design, which communicates parameters across three

levels in a hierarchy: *parameter client*, *parameter proxy*, and *parameter storage*. Each worker GPU runs a parameter client that maintains local parameters and communicates with a dedicated proxy to perform parameter synchronization. Each client provides a *push* and a *pull* interface to the local DL training operations – a parameter server functionality in conventional designs. Both parameter proxy and parameter storage run on the memory devices. A proxy is a communication service, acting as a bridge between a client and the parameter storage located in the same memory devices. The proxy also maintains a cache of the parameters stored with the parameter storage in the same memory device. As such, the multi-level parameter communication scheme reduces the data traffic by localizing parameter communication between client-proxy and local proxy-storage pairs. This allows the CCI path to accommodate proxy-proxy parameter synchronization at low latency, as illustrated in Figure 3.7.

3.3.5 Tensor Routing and Partitioning

In COARSE clients, we propose (1) a *tensor routing* mechanism to exploit non-uniform parameter size and bandwidth distribution; and (2) a *tensor partitioning* mechanism to exploit the bidirectional bandwidth offered by the serial bus interface in disaggregated memory systems.

Non-uniform Parameter Size and Local/Remote Bandwidth Distribution. We observe that both parameter size and local/remote communication bandwidth characteristics are not uniformly distributed in our experiments in AWS and supercomputing distributed systems. To investigate communication bandwidth characteristics, we measure PCIe peer-to-peer communication in two systems: (1) an AWS EC2 p3 instance with eight NVIDIA V100 GPUs, interconnected by NVLink and PCIe; this is similar to an NVIDIA DGX-1 system [83]; (2) an instance in the supercomputing center with four NVIDIA P100 GPUs, interconnected by PCIe. Figure 3.8 shows the bidirectional bandwidth between pairs of GPUs. On the supercomputing instance, local PCIe bandwidth (under the same PCIe switch) is higher than remote bandwidth, which is

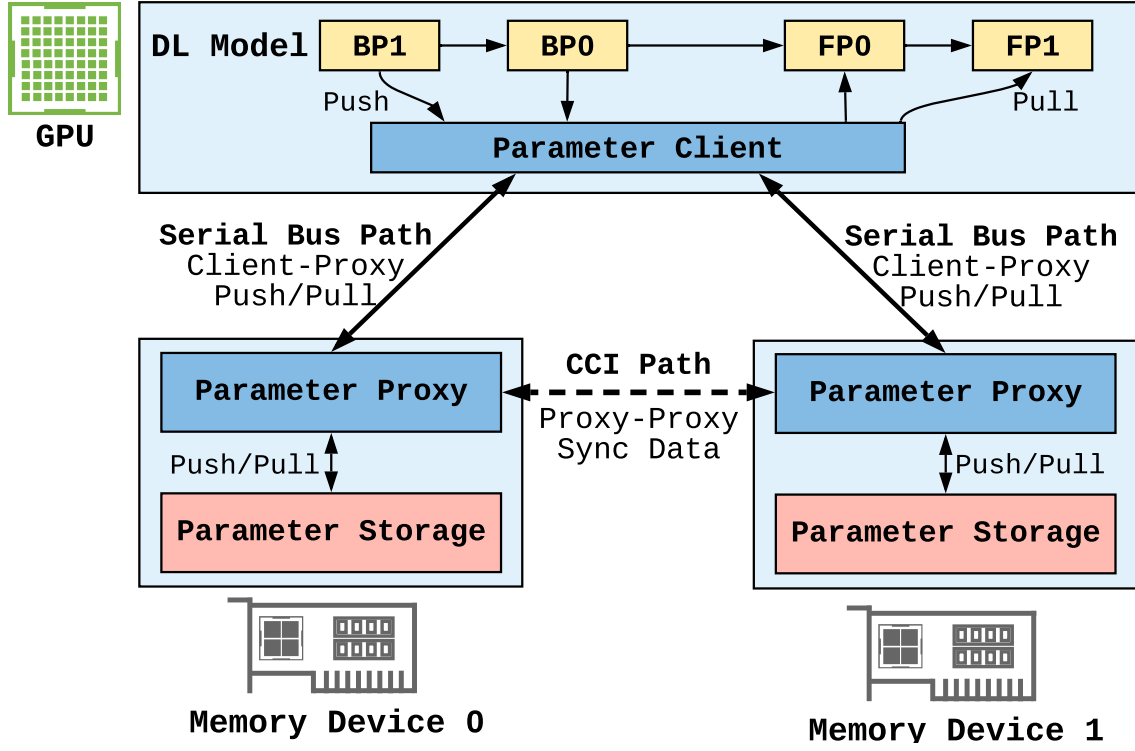
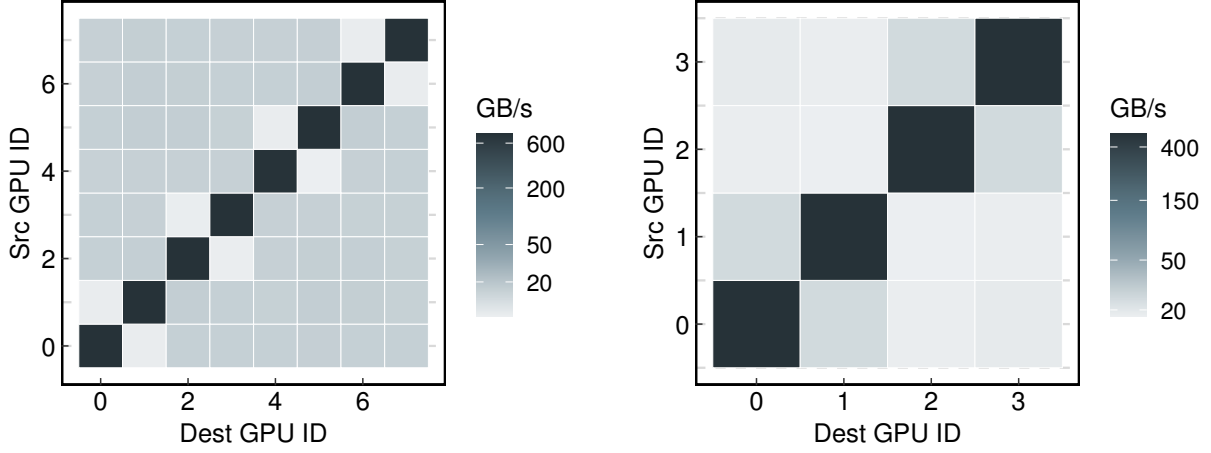


Figure 3.7: Decentralized parameter synchronization.

straightforward to imagine. However, the AWS instance appears to have a reverse bandwidth characteristic – higher remote than local bandwidth¹; such bandwidth “anti-locality” may be caused by unbalanced physical signal paths in the PCIe switch chipsets [117, 118]. In addition, we observe a non-uniform bandwidth and latency demand with different parameter sizes in our experiments with DL training in both systems: small-size parameter communication (less than 2MB) is latency-critical because the communication does not saturate the serial bus bandwidth; instead, transfer of large-size parameters is bandwidth critical. These observations indicate that no single routing and partitioning of DL operations fits all.

Tensor Routing. Ahead of DL training, COARSE builds a routing table for each client, which describes the destination proxies for various sized tensor push/pull requests. This tensor routing mechanism is based on two observations: (i) using serial buses, small-sized data transfer

¹This reverse bandwidth distribution is also observed by a recent study [117] and referred to as “anti-locality”.



(a) V100: Local PCIe bandwidth (9GB/s) is lower than remote (16GB/s).

(b) P100: Local PCIe bandwidth (25GB/s) is higher than remote (18GB/s).

Figure 3.8: PCIe device-to-device bidirectional bandwidth on (a) V100 from AWS and (b) P100 from SDSC.

is latency-sensitive while large data transfer is bandwidth-sensitive (Figure 3.14); (ii) non-uniform bandwidth: local serial bus communication bandwidth is not always higher than remote communication, although local latency is always better.

The routing table contains three entries, a data size threshold, a proxy id (*BwProxy*) for large tensor requests, and another proxy id (*LatProxy*) for small ones. The client issues tensor push/pull request to the *BwProxy* if the tensor size is larger than the threshold, otherwise to the *LatProxy*.

Benefits of Bidirectional Data Transfer. Serial bus interfaces, such as PCIe and NVLink, support bidirectional data transfer that delivers close to $2\times$ bandwidth of unidirectional transfer. For example, we observe that for a pair of local GPUs, the unidirectional bandwidth is 13GB/s, whereas the bidirectional bandwidth is 25GB/s, in an instance from the San Diego Supercomputing Center.

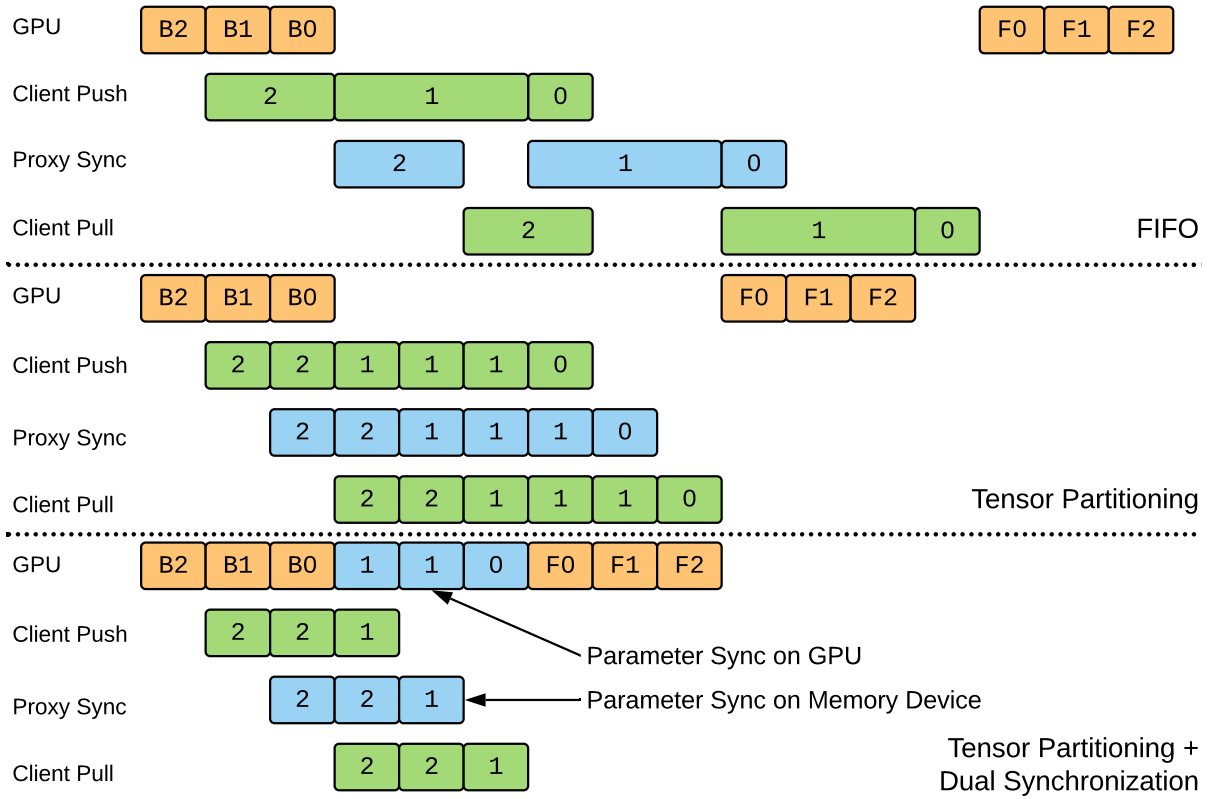


Figure 3.9: Tensor partitioning and Dual synchronization.

Tensor Partitioning. In COARSE, parameter synchronization includes three steps: client push, proxy synchronization, and client pull. Serial bus bidirectional data transfer allows the client push/pull to be processed concurrently. But with unequal-sized tensors, this bidirectional transfer is not fully utilized: as shown in Figure 3.9 FIFO case, there is no client push/pull between the tensor 0 push and tensor 1 pull.

To fully utilize bidirectional bandwidth, COARSE clients partition the tensors into equal-sized small shards and synchronize them as a pipeline. With tensor partitioning, the communication pipeline is filled without idle time (Figure 3.9), and the proxy synchronization can start as soon as the first tensor shard arrives.

Dynamic Partitioning. COARSE adopts a profiler to build the tensor routing table and determine the tensor partitioning size at runtime.

To build the routing table for a client, the profiler measures the client’s communication bandwidth and latency to each proxy, then chooses a proxy with the lowest latency ($LatProxy$) and one with the highest bandwidth ($BwProxy$). If $LatProxy$ and $BwProxy$ are not the same one, the profiler further measures the communication time T under different sized data, and determines a size S that makes $T_{LatProxy}(S) = T_{BwProxy}(S)$. The client considers a parameter latency-sensitive if it’s smaller than S , and sends it to $LatProxy$.

To find the tensor partition size, the profiler measures communication bandwidth under various sized data between a client and its $BwProxy$. The profiler determines the smallest data size S' among all tested data sizes that achieve the full bandwidth. The client partitions large tensors into S' sized small shards.

COARSE uses a dynamic profiling mechanism: before the training starts, COARSE profiles the communication performance to determine the initial routing table and partitioning size; while training is in progress, COARSE periodically profiles the communication and updates the routing and partitioning strategies accordingly.

3.3.6 Dual Parameter Synchronization

To efficiently synchronize the parameters, we propose (1) a priority-based dual synchronization mechanism to exploit the tensor locality and (2) a queue-based synchronization scheme to prevent synchronization deadlocks.

Dual Synchronization. In a DL model backward pass, parameters are updated in reverse order. Therefore, tensors from the first few layers are updated at the end of a training iteration while immediately consumed by the forward pass of the next iteration. These tensors need to be prioritized and synchronized as fast as possible to start the next iteration. To this end, COARSE adopts dual synchronization (Figure 3.9): the first few layers’ tensors are synchronized by worker GPUs, while the rest layers’ tensors are pushed to proxies and synchronized by proxies. In dual

synchronization, high priority tensors are synchronized immediately on worker GPUs, instead of being pushed to proxies and waiting until low priority tensor synchronizations to be finished.

COARSE determines the dual synchronization strategy with a performance estimation: suppose there are p worker GPUs and p memory devices; the DL model synchronizes n bytes of parameters in each iteration, the first m bytes are sent to memory devices, and the rest is synchronized by worker GPUs.

Using ring AllReduce synchronization, each proxy sends $\frac{2(p-1)}{p} \times m$ bytes to one neighbor proxy, and receives the same amount. So the proxy synchronization time is $T_{sync}(proxy) = (\frac{2(p-1)}{p} \times m) / BW_{proxy}$, where BW_{proxy} is the communication bandwidth between proxies. Similarly, the GPU synchronization time is $T_{sync}(GPU) = (\frac{2(p-1)}{p} \times (n - m)) / BW_{GPU}$.

Let T_{FP} denote forward pass computation time and T_{BP} denote backward pass time. The training iteration time is estimated as:

$$T_{train} = \max \{ T_{FP} + T_{BP} + T_{sync}(GPU), T_{FP} + T_{sync}(proxy) \}$$

p is assigned by the user, n is defined by the DL model, BW_{proxy} and BW_{GPU} are measured by the profiler (Section 3.3.5), T_{FP} and T_{BP} are measured by running a few iterations of training.

COARSE calculates the optimal value for m that minimizes the training time T_{train} . This m is then used for dual synchronization.

Deadlock Avoidance. A first-come-first-serve (FCFS) synchronization scheme may cause deadlocks, as shown in Figure 3.10: client 0 send tensor 1 to proxy 0 and tensor 2 to proxy 1, for synchronization. In FCFS scheduling, proxy 0 waits to synchronize tensor 1 while proxy 1 waits for tensor 2, and thus causing deadlock.

To prevent such deadlocks, COARSE adopts a queue-based synchronization if one proxy is shared by multiple clients: A proxy maintains a queue for each client that stores the tensors

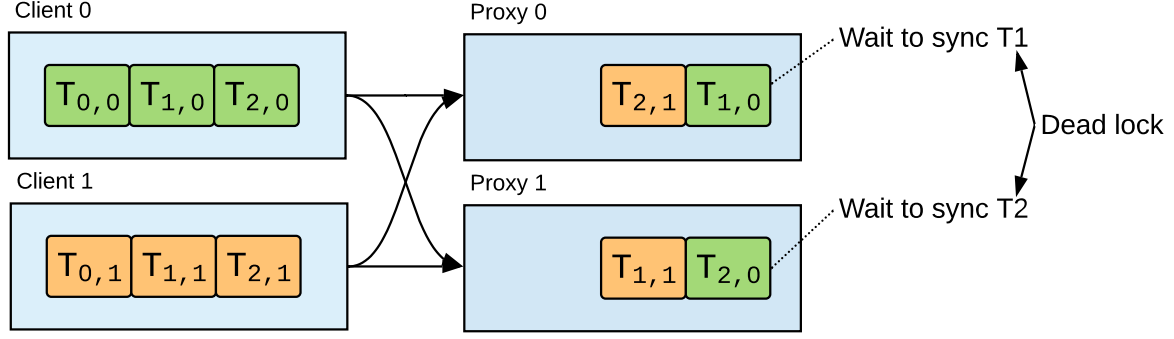


Figure 3.10: Deadlock in first-come-first-serve synchronization. $T_{i,j}$ represents tensor i from client j .

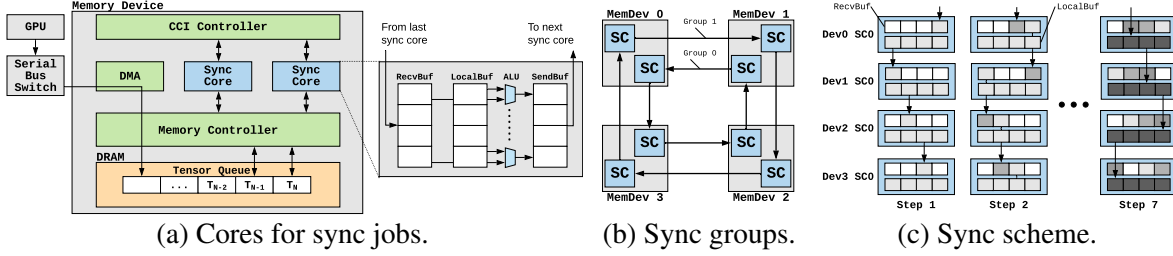


Figure 3.11: Parameter synchronization with sync cores (denoted as SC in figure b and c).

pushed by this client. The proxy synchronizes all tensor queues concurrently so that it's not blocked on a single tensor synchronization.

3.4 Implementation

In this section, we describe the implementation details of COARSE and the FPGA-based disaggregated memory prototype.

3.4.1 Sync Cores

COARSE offloads parameter synchronization jobs to memory devices. These jobs rely on simple computation operations such as the sum of float numbers, which do not require the full features of a generalized processor. Executing synchronization jobs on low-power generalized

processors, such as ARM Cortex-A53 on our Bittware Soc-250 FPGA, does not achieve high performance due to the limited number of arithmetic logic units. COARSE adopts specialized near memory processing cores – *Sync Cores* – to execute synchronization jobs with low design cost and high parallelism.

Sync Core Design. As shown in Figure 3.11a, each memory device is equipped with a set of sync cores, which can access local DRAM through the memory controller, and remote CCI memory device through the on-device CCI controller. Each sync core communicates with remote sync cores on other memory devices to synchronize the parameters. Each sync core maintains three buffers, a *RecvBuf*, a *LocalBuf*, and a *SendBuf*, to store the intermediate synchronization results. The sync core maps these buffers to CCI address space to accommodate direct read and write access from remote sync cores. To perform the arithmetic operations in parallel, each sync core employs a set of arithmetic logic units (ALU) that computes with the first two buffers' entries and stores the result into *SendBuf*.

Sync Scheme. To synchronize parameters, the sync cores perform group-based collective communication through CCI, where multiple groups synchronize different parameters in parallel. Each group consists of sync cores from each memory device, and uses ring-based topology for communication. Two adjacent groups use different ring directions to fully utilize the bidirectional bandwidth. Figure 3.11b shows an example configuration of two synchronization groups with four memory devices. In this configuration, group 1 use a reverse ring direction of group 0, so that communication between each pair of memory device is always bidirectional.

Figure 3.11c shows an example of the parameter synchronization scheme with one group of sync cores: Each sync core first loads a chunk of tensor data from local DRAM into *LocalBuf* and starts the iterative synchronization. In each iteration, each sync core sends an entry from *SendBuf* to the next sync core's *RecvBuf*, and receives an entry from the last sync core. It then applies the computation operation on the received entry and corresponding *LocalBuf* entry, stores

the result in SendBuf, and sends it in the next iteration. When all entries are synchronized, the sync core writes back the updated entries to DRAM and starts to synchronize the next chunk.

Fault Tolerance. Many existing ML frameworks rely on checkpointing to ensure fault tolerance: during the training process, the ML framework periodically takes a snapshot of ML model’s parameters and saves it to disk. In case any worker GPU fails during the training, the framework can recover its training progress with the latest checkpoint instead of training the model from the start. The checkpointing typically requires GPUs to transfer their model parameters to a centralized persistent storage and thus degrades the training performance.

To improve the checkpointing performance, COARSE leverages low-overhead copy-on-write with fine-grained snapshotting: when a memory device receives a parameter, it performs a copy-on-write procedure if the parameter contains updates to the previous version. Then at the end of each epoch, the memory device takes a snapshot of the current version of all parameters and saves it as a checkpoint.

3.4.2 COARSE

COARSE is implemented as a Python library with Python 3 and CUDA 11.4, and provides a plugin to TensorFlow framework [119]. It manages device memory, schedules synchronization, and handles requests between clients, proxies, and storage.

COARSE leverages GPUs to emulate the CCI memory device due to the lack of CCI support in existing CPUs and motherboards. COARSE accepts a user-defined GPU partition table that describes which GPU acts as a worker and which acts as a memory device.

Offline Profiling. Ahead of DL training, COARSE profiles the communication latency and bandwidth between GPUs and memory devices, as described in Section 3.3.5. This profiler is implemented as a CUDA program and measures peer-to-peer GPU communication by default.

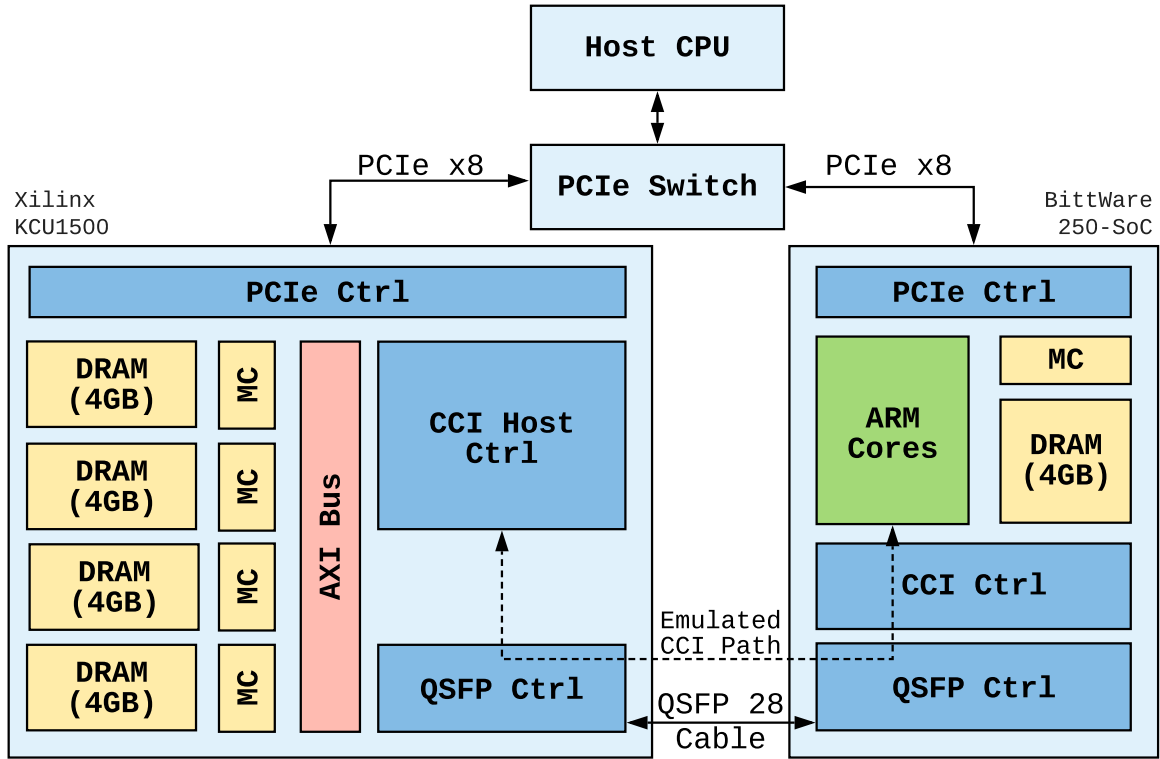


Figure 3.12: Disaggregated memory prototype with two FPGAs.

However, if GPUs are interconnected with NVLink, the profiler disables the NVLink feature to measure the PCIe bandwidth.

Client. A client maintains a tensor queue in worker GPU memory. When the DL model pushes a tensor, the client compares the tensor size with the partition threshold. If the tensor size is smaller than the threshold, it's enqueued directly without partitioning. The client partitions a large tensor into multiple shards and enqueues them separately, where each shard's size is equal to or larger than the threshold to maximize bandwidth utilization.

The client actively dequeues tensors, sends small tensors to the latency-friendly proxy, large tensor shards to the bandwidth-friendly proxy. After the first tensor is pushed to proxies, the client starts to pull the updated tensors and reconstruct the original tensors based on the partition history. These client operations are implemented in Python and integrated into the TensorFlow

computation graph.

Proxy and Storage. Each GPU-emulated memory device runs one parameter proxy and one parameter storage. The proxy leverages stream processors on existing GPUs as sync cores, and uses NCCL AllReduce to emulate collective synchronization operations between CCI memory devices. After each synchronization, the proxy copies the updated tensors to the storage. The parameter storage maintains a key-value table that maps a tensor id to the tensor value. To model the CCI communication characteristics, we implement a set of CUDA kernels that inject delays to communications based on data sizes.

TensorFlow Integration. COARSE provides a *distribution strategy* for TensorFlow version 2 or later. To use COARSE, the user just needs to import COARSE Python library and replace the original distribution strategy with COARSE strategy, which typically requires 2 lines of code change.

3.4.3 Disaggregated Memory Prototype

We implement a disaggregated memory prototype based on an industrial CCI protocol using two FPGAs, as illustrated in Figure 3.12.

The first FPGA works as a shared memory pool. We implement the host controller of the CCI protocol in the first FPGA. This controller manages the FPGA on-device DRAM and shares it with the host CPU and other FPGAs. To share the local memory with the host CPU, the controller exposes the on-device DRAM as a byte-addressable PCIe bar region. A program running on the host CPU can `mmap` [120] this region and use load/store instructions to access the FPGA memory. To share the local memory with other FPGAs, the controller accepts network requests from other FPGAs through QSFP connections.

The second FPGA works as a near-memory processing device. It contains the client

Table 3.1: Machines for DL training

Instance	CPU & Memory	GPU	Network
AWS g4dn	Intel 8259CL 384 GB	8 * NVIDIA T4 PCIe	100GBE
AWS p3	Intel E5-2686 v4 768 GB	8 * NVIDIA V100 NVLink, PCIe	100GBE
SDSC	Intel E5-2680 v3 128 GB	4 * NVIDIA P100 PCIe	100GBE

controller of the CCI protocol and connects to the first FPGA through a QSFP cable. The ARM core on this FPGA can access the shared memory through the CCI client controller.

3.5 Evaluation

We first profile the CCI prototype performance and build a performance model based on the profiling result. We then use the performance model to evaluate the DL training with COARSE, on three different systems.

3.5.1 Experiment Setup

The disaggregated memory prototype adopts a Xilinx KCU1500 FPGA [121] and a BittWare 250-SoC FPGA [122] to emulate the shared memory pool. Two FPGAs are interconnected with one QSFP cable [123] so that the ARM cores on 250-SoC can access the shared memory pool through CCI. KCU1500 exposes its shared memory as a byte-addressable PCIe bar region for the host CPU to access. The host system includes an Intel 8700K CPU, 32GB DDR4 memory, and an NVIDIA GTX1080 GPU. The motherboard adopts two PCIe switches enabling the GPU to access KCU1500 through peer-to-peer PCIe communication.

We profile the performance of this prototype and build a performance model based on profiling results to estimate the CCI bandwidth under different access sizes. We use this performance model to evaluate DL training time *only* in the baseline *DENSE* case (Figure 3.16).

We evaluate DL training with COARSE on three machine instances, as listed in Table 3.1.

On each machine, we use half of the GPUs to emulate the CCI memory devices. On AWS p3 machine, we provide an additional 2:1 configuration where each memory device is shared by two worker GPUs.

3.5.2 CCI Prototype Performance

We map the disaggregated memory prototype’s memory to the host CPU memory space using `mmap` [120], and evaluate its bandwidth in three use cases: (1) *CCI*: Host CPU accessing CCI memory using memory load/store instructions. (2) *GPU Indirect*: GPU accessing CCI memory by first copying data to host CPU memory. (3) *GPU Direct*: GPU directly accessing CCI memory using `cudaMemcpy` with prototype’s PCIe address.

Figure 3.13 shows the CCI bandwidth under different access sizes. Figure 3.13a shows that CCI read bandwidth remains stable under all access sizes. The difference between CCI line and GPU Indirect line is not visible in Figure 3.13a, which means the GPU Indirect read bandwidth is bounded by CCI bandwidth. However, the GPU Direct read bandwidth achieves 9x-17x speedup compared to CCI. Figure 3.13b shows that GPU Direct write bandwidth achieves 1.25x-4x speedup. This bandwidth speedup shows that CCI memory is more beneficial to the serial bus devices (e.g., GPUs) if CCI enables peer-to-peer communication.

We further profile the FPGA DMA bandwidth to estimate the full device bandwidth. As shown in Figure 3.14, DMA read and write achieves max bandwidth with an access size of 2MB or higher.

We build a bandwidth versus data size performance model for the following DL training evaluations. We assume the GPU Direct method achieves full serial bus bandwidth, and use correlated speedup/slowdown to derive CCI and GPU Indirect bandwidth in the DENSE system.

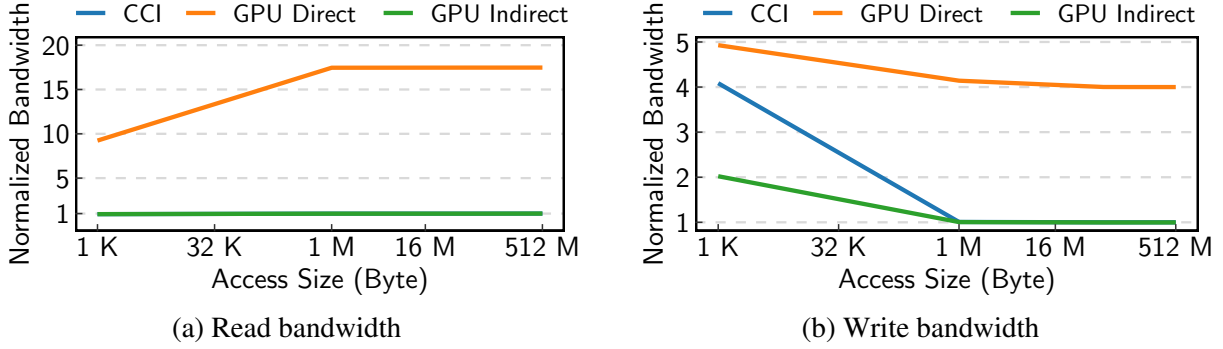


Figure 3.13: CCI bandwidth with CPU load/store instructions (*CCI*), GPU indirect copy through CPU (*GPU Indirect*), and GPU direct peer-to-peer copy (*GPU Direct*). Numbers are normalized to the CCI bandwidth with 512MB access size.

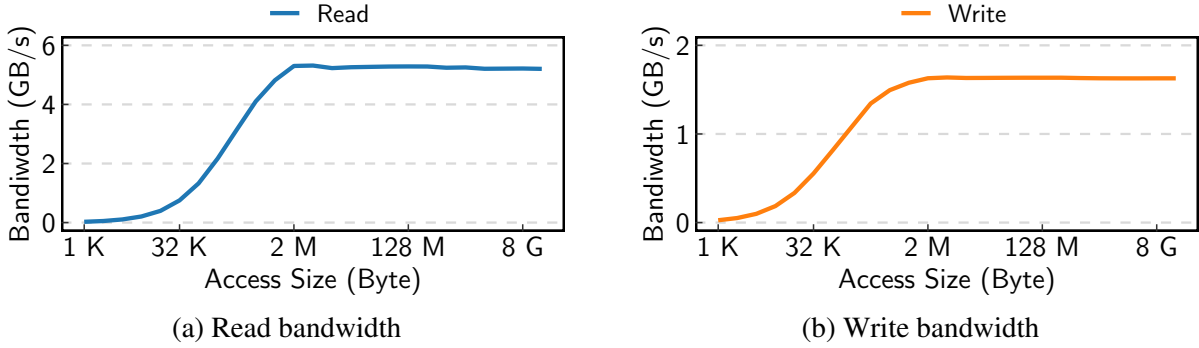


Figure 3.14: PCIe DMA bandwidth from host CPU to FPGA.

3.5.3 Tensor Routing

As discussed in Section 3.3.5, to build the tensor routing table for a client, COARSE profiles the communication performance from the client to each proxy. Figure 3.15 shows the profiling result for one client on each machine. Each figure shows client communication to (i) a *local* proxy that shares the PCIe switch with the client, and (ii) a *remote* proxy that achieves the highest bandwidth.

On AWS machines with T4 GPUs (Figure 3.15a), communication to local or remote proxy makes no difference, so each client always communicates with its local proxy.

On AWS machines with V100 GPUs (Figure 3.15b), accessing the local proxy achieves lower latency, while the remote proxy achieves higher bandwidth. So a client communicates with

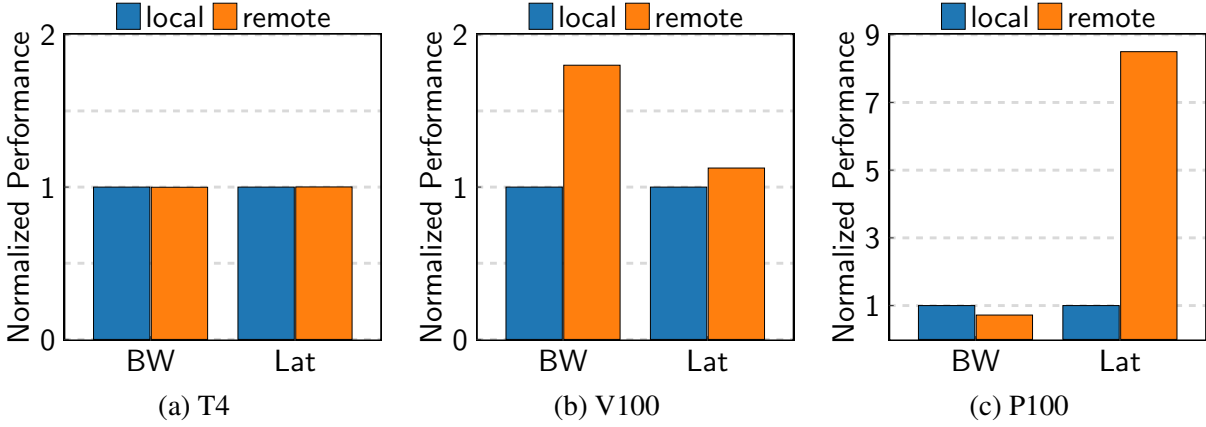


Figure 3.15: Communication performance from one client to its local and remote proxies. The local proxy shares the PCIe switch with the client. Numbers are normalized to local proxy performance.

the local proxy for small tensors, and the remote proxy for large tensors.

On SDSC machines with P100 GPUs (Figure 3.15c), accessing the local proxy achieves lower latency and higher bandwidth. So the client always communicates with its local proxy.

3.5.4 Training Speedup

We choose two DL models, ResNet50 [124] and BERT [104], to evaluate our design. We run ResNet50 training with ImageNet [125] dataset, using a per-GPU batch size of 64. We run BERT fine-tuning with SQuAD 1.1 [126] dataset, using a per-GPU batch size of 2.

We evaluate three communication schemes, (1) a naïve disaggregated CCI memory design as shown in Figure 3.5 (*DENSE*), (2) AllReduce using NCCL without CCI memory (*AllReduce*), and (3) COARSE where the number of worker GPUs is equal to the number of CCI memory devices (*COARSE*). We use the training speedup (compared to the *DENSE* case) as the major metric.

To estimate the communication time in the *DENSE* case, we run the DL training with a parameter server running on CPU, and estimate the GPU to CPU bandwidth according to the CCI prototype performance model (Section 3.5.2). This parameter server is provided by TensorFlow’s

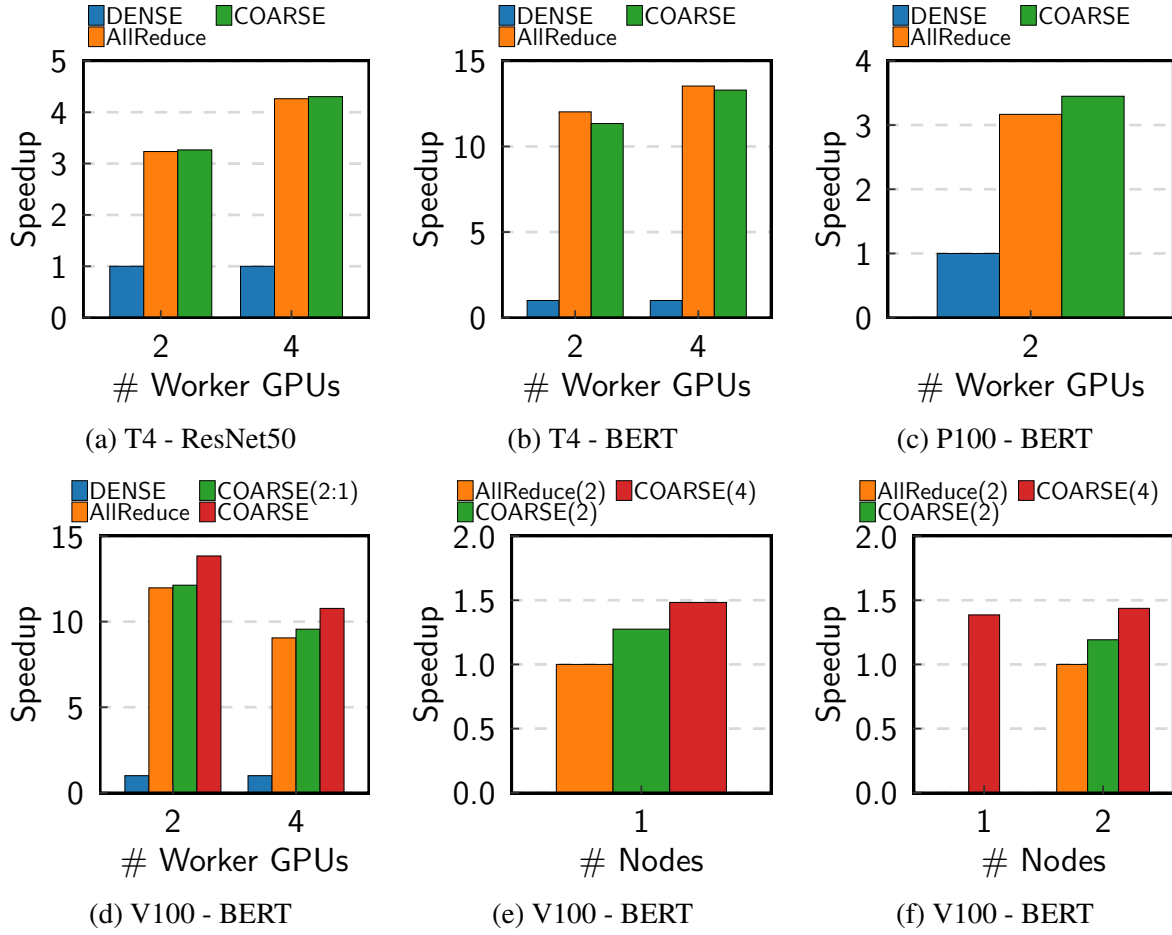


Figure 3.16: DL training on (a-b) AWS g4dn instance with T4 GPUs, (c) SDSC machine with P100 GPUs, (d) AWS p3 instance with V100 GPUs, (e) AWS single node with V100, (f) AWS multi nodes with V100. In (d) *COARSE(2:1)*, each memory device is shared by two worker GPUs. In (e-f), the baseline is *AllReduce* which is equivalent to *AllReduce* in (d); the numbers in legends represent training batch sizes.

ParameterServer distribution strategy.

Figure 3.16(a-b) show the evaluation on AWS with NVIDIA T4 GPUs. COARSE achieves $3.3\times$ and $4.3\times$ speedup for ResNet50, and $11.3\times$ and $13.3\times$ speedup for BERT, which is slightly lower than AllReduce. COARSE does not work efficiently on this platform because there's no unbalanced bandwidth (Figure 3.15a) and this platform does not support GPU p2p communication.

Figure 3.16c shows the evaluation on SDSC with P100 GPUs, where COARSE achieves $3.4\times$ speedup for BERT. On the AWS machine with V100 GPUs (Figure 3.16d), COARSE

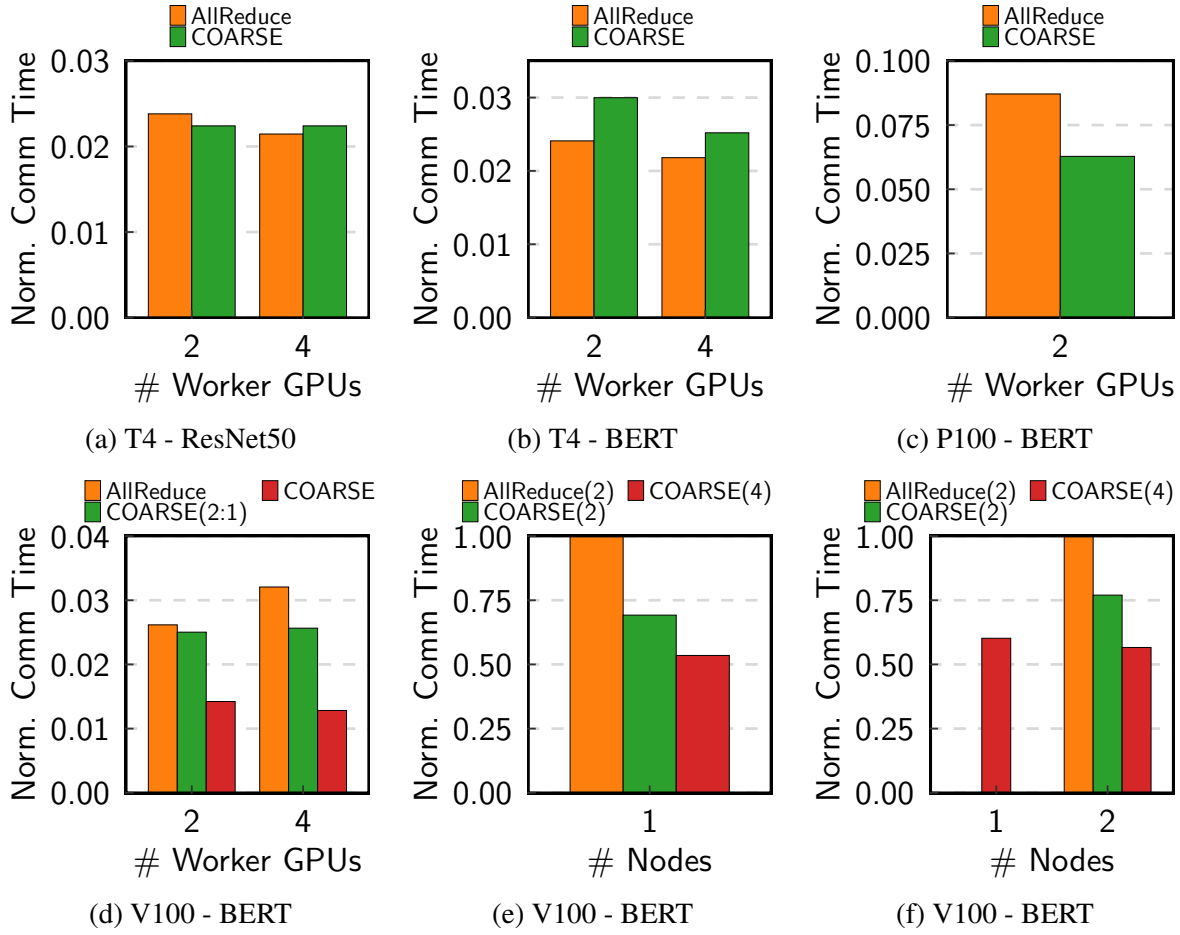


Figure 3.17: Blocked communication time in DL training. In (a-d) numbers are normalized to the CCI case; in (e-f) numbers are normalized to the AllReduce case. The CCI bar is ignored for better comparison between AllReduce and COARSE.

achieves $10.8\times$ - $13.8\times$ speedup for BERT. On these two machines, COARSE beats AllReduce by leveraging non-uniform bidirectional bandwidth.

Figure 3.16(e-f) show single- and multi-node training performance. The baseline is set to *AllReduce* because DENSE does not assume a multi-node system. On the single node system (Figure 3.16e), AllReduce can only use a batch size of 2 due to memory capacity limitation, while COARSE can use a batch size of 4 and achieves 48.3% faster training. On the multi-node system (Figure 3.16f), COARSE is up to 42.7% faster than AllReduce. Even a single node COARSE with a large batch size of 4 can achieve 38.6% speedup compared to a two-node AllReduce training.

3.5.5 Communication Time

To understand the speedups in DL training, we break down the training time and measure the communication time that blocks the training computation. Figure 3.17 shows the result of blocked communication time where numbers are normalized to (1) CCI communication time in single-node evaluations (Figure 3.17(a-d)), and (2) AllReduce in multi-node evaluations (Figure 3.17(e-f)). AllReduce and COARSE reduce the communication to less than 10% compared to the naïve parameter server running on CCI memory. AllReduce achieves this speedup by using GPUs without external parameter storage. While COARSE provides additional parameter storage through CCI memory, with comparable or even better performance.

On AWS machine with T4 GPUs (Figure 3.17b), COARSE introduces additional 18%-20% blocked communication for BERT model. This is because COARSE exploits non-uniform bandwidth for faster communication, but this machine does not have such a bandwidth feature. On SDSC machine with P100 GPUs (Figure 3.17c), COARSE reduces the communication time by 28% for BERT. On AWS machine with V100 GPUs (Figure 3.17d), COARSE reduces the blocked communication time by 20%-42% for BERT. On single node and dual node AWS machines with V100 GPUs (Figure 3.17(e-f)), COARSE reduces the communication time by 23%-46%. These results show that COARSE is highly efficient in reducing communication overhead in distributed training.

3.6 Related Work

To our knowledge, this is the first paper to explore the cache-coherent disaggregated memory system to accelerate communication in DL training. This section discusses previous works related to our study.

Parameter Server. Li et al. [127] propose a distributed parameter server design based on Ethernet, while COARSE leverages cache-coherent interconnection. GeePS [128] is a GPU-specialized parameter server which allows the parameter server to store part of the parameters on GPU memory, so that parameter push/pull can be processed faster. GeePS enables the multi-server-node training but limits to one GPU per node, while COARSE works with multiple GPUs per node. ParameterHub [129] proposes a parameter server with software-hardware co-design to optimize the rack-scale Ethernet latency, while COARSE reduces serial bus latency using cache-coherent interconnection.

Speedup Communication in DL Training. Hop [99] is a queue-based parameter synchronization in decentralized DL training. Hop leverages bounded staleness training, while COARSE targets synchronized training and thus is orthogonal with this work. Prague [111] proposes a group-based AllReduce synchronization scheme for distributed DL training. Prague uses partial synchronization to achieve high performance, while COARSE provides full synchronization without losing precision. Zhao et al. [94] propose a hierarchical multi-level parameter server design with GPU memory, CPU memory, SSD, and network. This parameter server is helpful in recommendation systems where most parameters are sparse. It relies on CPU-GPU communication to exchange parameters, while COARSE enables GPU direct access to CCI memory device to exploit full serial bus bandwidth. ByteScheduler [130] introduces a communication scheduler for DL training. It is beneficial when using low bandwidth Ethernet communication, but provides less speedup when using intra-node serial buses for communication. Compared to ByteScheduler, COARSE leverages CCI disaggregated memory to speed up the communication on fast serial buses and support large DL models. Blink [86] is a collective communication scheme optimized for GPUs interconnected by both NVLink and PCIe. It runs parameter synchronization jobs on GPUs while COARSE offloads these jobs to CCI memory devices to further improve GPU utilization. Klenk et al. [131] propose an in-network accelerator for collective communication.

This work does not support extended memory space for GPUs, while COARSE leverages CCI memory devices to enable larger models to be trained. In addition, COARSE exploits the non-uniform serial bus bandwidth to further improve the performance. As a result, COARSE achieves 48.3% BERT training speedup over AllReduce, while this prior work achieves less than 10% speedup [131] in the same model training.

Disaggregated Memory Lim et al. [132, 133] propose a memory disaggregation architecture with a memory blade connected over PCIe to expand CPU accessible memory. COARSE takes one step further to explore the potential of disaggregated memory using CCI. Kim et al. [134] propose a memory network for GPU and CPU using packet routing from HMCs, while COARSE does not rely on specific memory media technology. Kwon et al. [135] propose a memory-centric architecture for distributed DL training. This work assumes unified communication bandwidth in serial bus, while COARSE exploits the non-uniform bandwidth.

Near Memory Processing for DL Training TensorDIMM [136] provides a near memory processing design to offload embedding operations in recommendation systems. It accelerates the specialized operations while COARSE provides generic acceleration to parameter synchronization operations. iSwitch [137] proposes a parameter synchronization design in Ethernet switch memory. It is beneficial to reinforcement learning where model size is small enough to fit in switch memory, while COARSE targets large models and provides extended parameter storage.

3.7 Conclusion

In this paper, we propose COARSE, a distributed parameter synchronization scheme based on disaggregated memory, for distributed DL training. COARSE exploits the non-uniform serial bus bandwidth and bidirectional bandwidth to accelerate the parameter synchronization in DL training. It leverages disaggregated memory to offload the parameter synchronization jobs and

hence improves the GPU utilization. It combines the emerging cache-coherent interconnection with MPI-like collective communication to provide low-latency parameter synchronization. Our evaluation shows COARSE significantly speeds up the DL training compared to the centralized parameter synchronization design with CCI.

3.8 Acknowledgements

Chapter 3, in full, is a reprint of the material as it appears in IEEE International Symposium on High-Performance Computer Architecture, 2022. Zixuan Wang, Joonseop Sim, Euicheol Lim, Jishen Zhao. “Enabling Efficient Large-Scale Deep Learning Training with Cache Coherent Disaggregated Memory Systems”. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Attacking and Securing Heterogeneous Systems

4.1 Introduction

Microarchitectural side channels allow attackers to leak information from other users co-located on shared computing resources. Researchers have demonstrated such attacks by exploiting various hardware resources that are shared among untrusted users [36, 138–142]. For example, an attacker can construct a timing side channel based on the shared CPU cache and use this to break cryptography [143, 144], steal keystrokes [145], and violate the privacy of encrypted databases [146]. These side channels are also the basic block to developing more advanced microarchitectural attacks that leak arbitrary data and undermine the confidentiality of several isolation domains on modern systems [147–154].

The research community has put a lot of effort into proposing defenses against microarchitectural attacks, but these proposals are not comprehensive. For attacks that target components internal to the CPU core [139, 140, 155], the recommendation is to isolate these resources for security-critical operations spatially and temporally [156, 157]. On existing CPUs with no

fine-grain support for isolation, this can be achieved by flushing CPU resources across context switching or making sure untrusted threads are not simultaneously executed on the same core. For attacks on the shared CPU cache and its directory structure [142, 158], mitigations based on partitioning the cache and randomizing cache accesses are proposed [159–162]. These CPU mitigations, promising for protecting security-sensitive computation, come with a performance penalty. Unfortunately, systems practically benefit little from paying the corresponding performance penalty to defend against CPU side channels if attackers can exploit other shared hardware resources such as the on-chip interconnect [163–167] and the DRAM banks and row buffers [168]. This gap motivates us to look into attacks on new memory subsystems.

We study the security implications of scalable server-grade non-volatile RAM (NVRAM) DIMMs as implemented by Intel’s Optane DIMM [13]. NVRAM DIMMs enable a larger memory capacity and support for data persistence long desired by server developers. Recent performance characterization studies [2, 169] have shown that the Optane DIMM delivers its high levels of performance and scalability by employing various optimizations including multi-level buffers, internal address remapping schemes, and wear-leveling mechanisms. This combination leads to a discrepant performance behavior compared to what researchers expected before the product release [1, 170, 171]. Although previous studies have investigated the microarchitecture of the Optane DIMM and analyzed its performance, its security implications remain largely unexplored. In this study, we investigate microarchitectural covert/side channels enabled by Optane DIMM, their impact on the security of real-world applications, and how we can improve system security against potential side channels. More specifically, we contribute the following:

1. Reverse-engineering (Section 4.3). We perform reverse-engineering of the opaque design of NVRAMs, which helps us uncover new information leakage sources. Our goal is to detail the on-DIMM cache structures and configurations, control policies, and performance behaviors. We develop carefully crafted microbenchmarks that run in both kernel and user spaces to achieve this.

These microbenchmarks trigger specific memory behaviors, which lead to detectable performance variances that reveal the corresponding hardware designs. As a result, we unveil a much more detailed picture of Optane DIMM microarchitecture compared to previous works [2, 169]. Our findings include the on-DIMM cache structures and wear-leveling policies, which we then exploit to develop new information leakage attacks.

2. Constructing covert channels (Section 4.4). We develop and quantify new covert/side-channel attacks to empirically verify the existence of information leakage via the uncovered knowledge of Optane DIMM microarchitecture. First, we exploit the previously-undocumented on-DIMM cache structure to construct a cross-VM covert-channel attack. We show that cross-VM covert channels using the NVRAM cache are stable and achieve high channel capacity and low noise by solving several challenges. Second, we construct a covert channel that exploits the NVRAM wear-leveling mechanism to leak updates to a filesystem, which allows an attacker to monitor whether a victim updates its file without requiring elevated permission.

3. Side-channel attack case studies (Section 4.5). Next, we show that our findings go beyond covert communication channels and affect the security of real-world applications. We demonstrate several side-channel attacks exploiting the NVRAM cache, under the umbrella of NVLeak attacks, applicable to everyday use cases of NVRAM:

First, we demonstrate several attacks in the scenario where NVRAM is used as persistent storage, compromising the privacy of a SQL database or key-value storage. Although an attacker who shares the NVRAM with a victim does not have access to the victim’s database/storage file and its queries, they can learn about its queries through NVRAM cache access patterns. Ultimately, an attacker can learn the details of queries and parameters, and previous work [172] shows that such information leakage is devastating for the privacy of encrypted databases.

Then, we demonstrate an NVLeak attack in the scenario where NVRAM is deployed like a volatile memory (like the DRAM). In this common scenario, to speed up workloads that don’t

require persistent storage, we show NVLeak can spy on code pages and detect which execution path is taken by a program whose code pages are stored in the NVRAM. Ultimately, we show that this has consequences for security-critical applications like cryptographic schemes.

4. Mitigations (Section 4.6). We propose a set of mitigation mechanisms to defend against the NVCache-based side channels based on the reverse engineering results and side-channel attacks. We first propose a software-based L2 NVCache mitigation that allows a victim application to allocate memory blocks from isolated NVCache sets that are not shared with other applications, including the attackers, thus preventing information leakages. We develop this mitigation into a PMDK [65] key-value store to make it resistant to NVCache-base side channels. The experimental results show that this mitigation’s performance overhead is $< 4\%$. We then propose a software-based mitigation for L1 NVCache and WPQ, and a hardware-level mitigation for the entire NVRAM hierarchy.

Open source and responsible disclosure. We open source our code¹ in the hope of facilitating future off-chip memory security research. We have disclosed the vulnerabilities and our code to Intel.

4.2 Background

In this section, we provide some background knowledge of the NVRAM and microarchitectural side-channel attacks.

4.2.1 NVRAM

Like DRAM, Non-Volatile RAM (NVRAM) provides a byte-addressable interface that allows the CPU to access NVRAM directly. NVRAM promises low-latency and high-bandwidth

¹<https://github.com/TheNetAdmin/NVLeak>

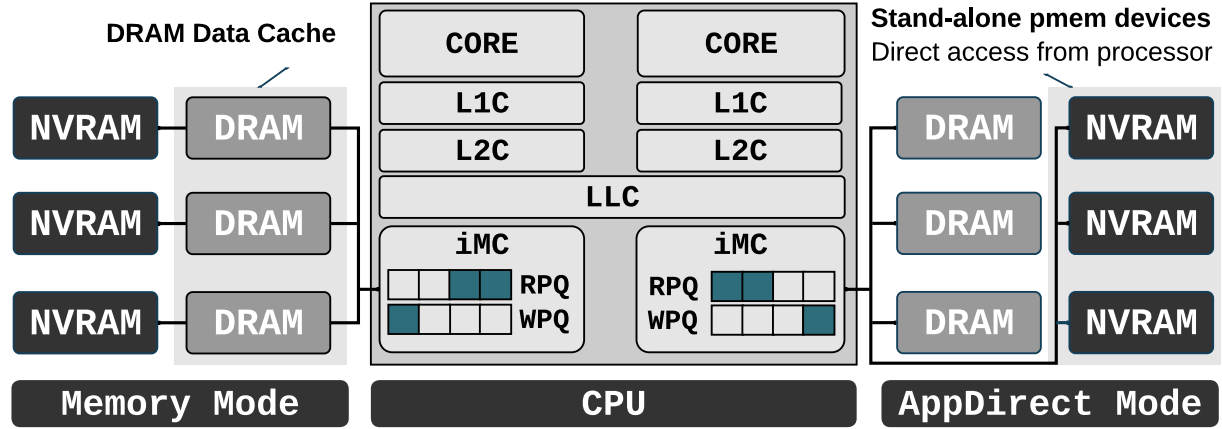


Figure 4.1: Memory hierarchy equipped with Optane DIMMs.

performance that is comparable to DRAM and faster than conventional NAND-based persistent storage devices [1]. On the other hand, NVRAM can also work as persistent storage, guaranteeing its data survives power reset, i.e., a user can rely on an NVRAM-aware filesystem for persistent storage [23]. Intel Optane DIMM [173] is the most widely adopted commercial NVRAM product.

NVRAM-based server systems. Figure 4.1 shows an example architecture of a server system equipped with Optane DIMM. Optane DIMMs (denoted as NVRAMs) are connected to the memory bus along with DRAM DIMMs. Intel’s Cascade Lake processors, the first microarchitecture to support Optane DIMM [29], can support up to 6 Optane DIMMs per processor. The processor’s integrated memory controllers (iMCs) manage access to the memory bus and connected memory modules, including the CPU’s last level cache (LLC) and off-chip memory media. The iMCs have read-pending-queues (RPQs) and write-pending-queues (WPQs) to process accesses to Optane DIMMs. WPQ, which is interesting for our work, stores the pending cache line data to be written back to NVRAM and ensures its data is safely persisted during power failures. Intel only documented the existence of WPQ [32] without further details such as WPQ size.

NVRAM operation modes. As shown in Figure 4.1, Optane DIMMs can be configured to operate in two different modes: Memory Mode or AppDirect Mode. In Memory Mode, a DRAM DIMM becomes a data cache for the Optane DIMM in the same channel. This mode allows users to enjoy the extra capacity provided by Optane DIMM, but it does not provide data persistence. In contrast, in AppDirect Mode, the Optane DIMMs are used as stand-alone persistent memory that is directly accessible to the software running on the CPU cores. The programmers can use regular load/store instructions to the NVRAM region and access in-NVRAM data structures. In this case, system software and NVRAM hardware should guarantee that the data structures are recoverable during system crashes or power outages [23, 31].

NVRAM wear leveling. Persistent storage devices such as SSDs use wear-leveling mechanisms to prolong the device service life: the persistent media’s cells generally only support a limited number of write cycles [174], after which the cell is worn-out and not reliable to store data. The wear-leveling mechanism tries to evenly distribute write accesses to the media cells by migrating data from one cell to another after a number of write cycles. Optane DIMM uses 3D-Xpoint as its persistent media [33], which also requires wear-leveling to reduce the number of bad media cells. Currently, there is no publicly available documentation about the details of its wear-leveling algorithms.

NVRAM microarchitecture. Recent works [2, 169] unveil several previously unknown microarchitectural details of Optane DIMM. These works identify the existence of on-DIMM buffers organized as a hierarchy and the presence of a wear-leveling mechanism and a multi-DIMM interleaving scheme. In this work, we dive deeper into its microarchitecture, revealing more details and correcting some of the findings of the prior work. Thus, this paper represents the most comprehensive analysis of NVRAM to date, which also leads us to the first security analysis of an NVRAM microarchitecture.

4.2.2 Microarchitectural Side Channels

Microarchitectural side-channel attacks exploit contention on shared hardware resources such as shared caches [138, 175–178], translation lookaside buffers (TLBs) [179], branch predictor structures [139, 180], DRAM buffers [168], and CPU execution ports [155, 181] to bypass software isolations and leak private information from other users. We discuss memory-related attacks in more detail due to the focus of our work:

Cache attacks. In a cache-based side-channel attack (cache attack), an attacker who cannot directly access the victim’s data can observe the cache state changes made by the victim application. The attacker can monitor the state of the cache and derive the victim’s secret information from its cache-access pattern. A typical cache attack comprises three main steps: (1) the attacker *prepares* the cache into a desired state so that the victim’s cache activities are measurable. For example, the attacker prepares the cache by evicting specific cache lines from the shared cache. (2) the attacker *waits* for the victim to modify the cache state. (3) the attacker *measures* the cache state to determine whether the victim has accessed a target cache line or not. An attacker can achieve this by measuring the access time of its cache lines [175, 177] or the time of the victim’s execution [138].

Cache attacks can be categorized into different classes based on their method for the preparation step. Flush-based attacks [176, 177, 182] assume a shared memory (e.g., a shared library) between the attacker and the victim, so the attacker can prepare the cache by simply executing a `clflush` [183] instruction to evict target cache lines. In contrast, conflict-based attacks (e.g., Prime+Probe [175]) do not rely on shared memory. Instead, to prepare the cache states, the attacker identifies and loads multiple cache lines mapped to the same cache set as the target address, forcing the cache to evict the victim’s target cache line due to conflicts.

DRAM row-buffer conflicts. Previous research has also attacked on-DIMM structures. These attacks leak secrets by exploiting DRAM-specific optimizations and features such as row buffers [168] and row activation [184]. DRAMA [168], for example, exploits the DRAM row buffer, an on-DIMM SRAM memory that caches an entire DRAM row and provides faster access than DRAM banks. DRAMA performs a Prime+Probe attack by leveraging the timing difference between row buffer hits and misses to detect victim activities such as keystrokes.

4.3 NVLeak Reverse-Engineering

In this section, we perform reverse-engineering of the NVRAM’s microarchitecture. We build our experiments on top of LENS [2], a recent performance characterization study of Optane DIMM. Our results confirm some of LENS’ findings, including the capacity of DIMM caches and their block size on two different versions of the Optane DIMM. Additionally, we recover undocumented details such as the set associativity of the DIMM caches and their indexing schemes (Section 4.3.2). Finally, we correct some essential details about the wear-leveling policy and its implementation (Section 4.3.3).

4.3.1 NVRAM System Configuration

Table 4.1 shows the configuration of the NVRAM-equipped machines we use for our experiments. We configure all DIMMs to be non-interleaved to ensure that our experiments always run on a single DIMM. Additionally, we configure Optane DIMMs to operate in AppDirect mode. For our reverse-engineering microbenchmarks, we mount a dummy filesystem on an Optane DIMM which provides identity memory mapping on the Optane DIMM memory region. To minimize the noise in our experiments, we turn off the CPU hardware prefetchers, disable simultaneous multithreading (SMT), and boost the CPU to performance mode with the help of the CPU scaling governor. This section presents results using Server A (Table 4.1), and Section 4.A.2

Table 4.1: NVRAM-equipped server system configuration.

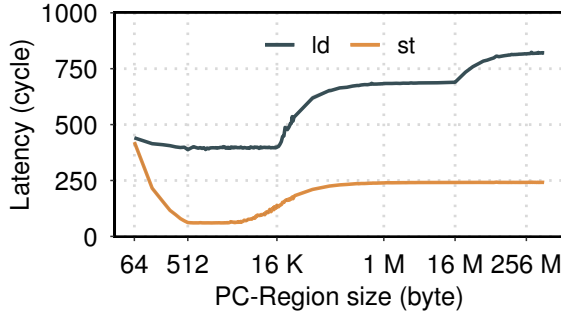
	Server A	Server B
CPU	Intel Xeon Cascade Lake 24 Cores per socket, 2 sockets HyperThreading off	Intel Xeon Gold 6230 20 Cores per socket, 2 sockets
L1 Cache	32 KiB 8-way I-Cache, 32 KiB 8-way D-Cache, private	
L2 Cache	1 MiB, 16-way, private	
L3 Cache	33 MiB, 11-way, shared	27.5 MiB, 11-way, shared
DRAM	DDR4, 32 GiB, 2666MHz	DDR4, 16 GiB, 2666MHz
NVRAM	256 GiB, 2666 MHz Firmware: 01.01.00.5253	128 GiB, 2666 MHz Firmware: 01.02.00.5355
Kernel	Linux 5.4.0	

shows results on Server B.

4.3.2 Recovering Details of NVCache and WPQ

We design a pointer chasing microbenchmark that reveals the structure of the NVRAM cache (NVCache) and CPU’s write-pending-queue (WPQ). Here, we first describe the microbenchmark, then briefly discuss how we instantiate this program to confirm the results of LENS on cache capacity, block size, and WPQ structure. We then discuss how we recover NVCache set associativity, the number of sets, and NVCache indexing scheme.

Pointer chasing microbenchmark. We divide a contiguous memory region—a pointer chasing region (PC-Region)—into equal-sized blocks (PC-Blocks). We initialize each PC-Block with a pointer to another random PC-Block. Recursively dereferencing the pointers results in accessing all PC-Blocks in random order. This also defeats a cache prefetcher if that exists. To minimize the effects of CPU caches on our measurements, we set the NVRAM memory region as *uncacheable* through x86 Memory Type Range Register (MTRR) [183]. We ensure that every read or write operation reaches the NVRAM DIMM by using non-temporal load and store instructions. In an environment where MTRR is not available, e.g., user-space processes or virtual machines, one



(a) Pointer Chasing

PC-Block	L1 NVC	L2 NVC	WPQ
64B	1.73	1.21	6.65
128B	1.43	1.17	3.80
256B	1.00	1.17	1.85
512B	1.00	1.12	1.00
1K	1.00	1.07	1.00
2K	1.00	1.03	1.00
4K	1.00	1.00	1.00

(b) Amplification

Figure 4.2: Pointer chasing microbenchmark results and amplification factors for each architectural components.

can instead insert a `clflush` instruction after each memory access instruction to imitate the uncacheable effects.

Recovering cache capacity. To identify the cache capacity, we follow the LENS [2] methodology and run the pointer chasing microbenchmark with varying region sizes (PC-Region) while keeping the PC-Block size constant. The results of this experiment (Figure 4.2a) confirm the presence of two levels of NVCaches with 16 KiB and 16 MiB capacities on both of our machines. Hereafter, we refer to the 16 KiB cache as L1 NVCache and the 16 MiB one as L2 NVCache.

Recovering L1/L2 NVCache block size. To confirm the block size reported by LENS [2], we use the pointer chasing microbenchmark with varying PC-Block sizes and measure the read/write *amplification factor*. The read/write amplification factor is the ratio of data that is read/written to the size of the requested data. For example, for a cache with 256 B block size, a single read request of 64 B will bring a 256 B block into the cache; thus, we will have a read amplification of 4. Our read/write amplification results (details in Figure 4.2b) confirm that the two NVCaches have different block sizes. L1 has 256 B blocks while the L2 NVCache has 4 KiB blocks.

Recovering CPU WPQ structure. Figure 4.2a shows that pointer chasing write latency drops until 512 B region size, which indicates a buffer or queue on the write path. Given its

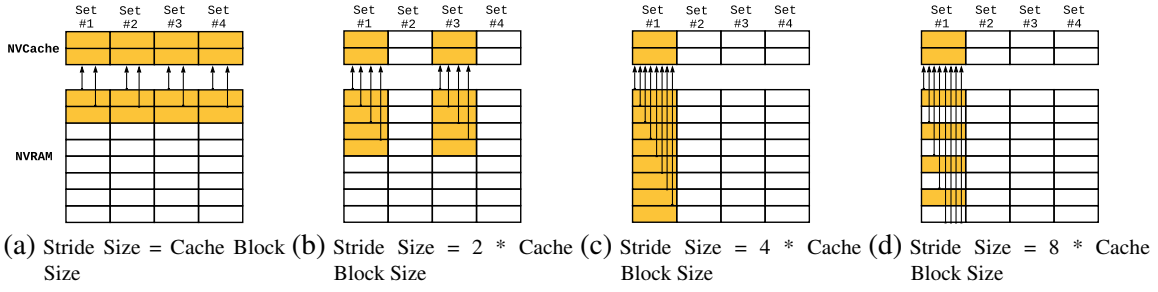


Figure 4.3: Detecting NVCache set structures with a strided pointer chasing microbenchmark. In this example, NVCache has 4 sets, where each set has 2 cache blocks. The strided access reaches a maximum cache contention once the stride size reaches 4 times the cache block size. Figure (d) is cut off with 4 more memory block accesses out of the figure range.

small size and presence only on the write path, we believe it is the write pending queue (WPQ) embedded in the Intel CPU’s on-chip memory controller [32]. The WPQ’s write amplification factor (Figure 4.2b) drops to 1 when using 512 B PC-Block; considering the pointer chasing microbenchmark inserts one `mfence` instruction after each PC-Block access, we believe that one `mfence` instruction flushes the entire WPQ.

Recovering NVCache associativity and number of sets. We run the pointer chasing microbenchmark with strides to identify the number of cache sets, which prior works [2, 169] have not revealed. As illustrated by Figure 4.3, increasing the stride size directs the pointer chasing accesses to fewer cache sets. For example, assuming a target cache has four sets (unknown to the attacker) where each set stores two cache blocks, and the cache uses a linear indexing function to index the sets. If the attacker runs the pointer chasing code to access eight continuous cache blocks, the attacker can fill up the entire cache (Figure 4.3a). But if the attacker doubles the stride size (Figure 4.3b), its accesses will skip every other set, and the attacker can only use half of the cache sets. Finally, once the attacker reaches a stride that matches the number of sets, all memory accesses will be mapped to only one cache set (Figure 4.3c). At this point, doubling the stride size does not reduce the available cache capacity anymore (Figure 4.3d).

Using this method, we scan all possible stride sizes and measure the available cache

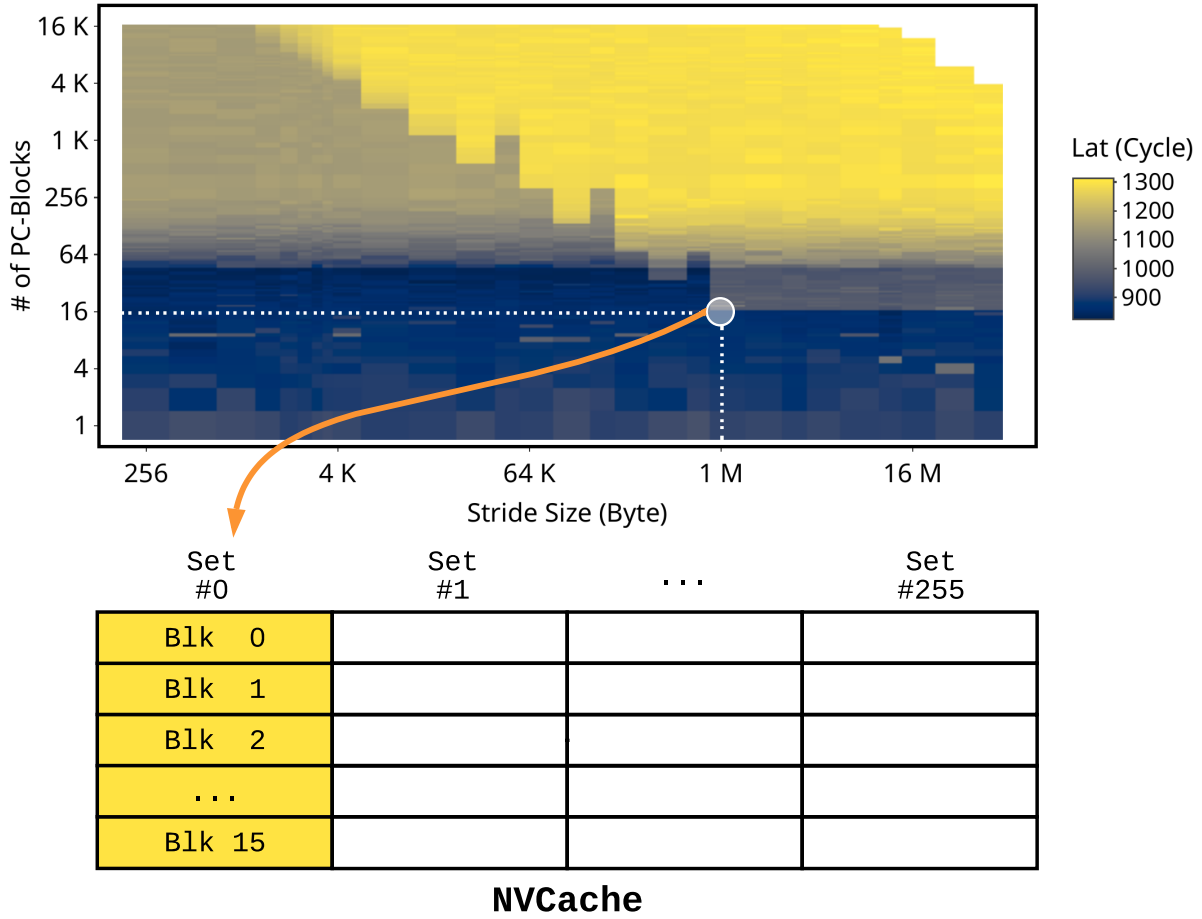


Figure 4.4: Pointer chasing microbenchmark results. The heatmap shows the pointer chasing latency under varying striding sizes and number of blocks. When we have 16 blocks or less, the latency does not increase for any stride size, indicating that we have at least 16 block available, i.e., associativity of 16.

capacity, and find a point where the cache capacity stops decreasing. We can then use that stride size to deduce the number of sets in the cache, and since we already know the total number of blocks in the cache, we can also derive its set associativity.

Figure 4.4 shows a heatmap of the pointer chasing read access with different stride sizes and different numbers of blocks. We repeat the pointer chasing with strides for multiple rounds (16 or more) and measure the average latency to get a stable result. We use a 64 B pointer chasing

block size² so that each NVCache block gets only one memory access.

L1 NVCache set structure. We start with a stride size of 256 bytes. This stride size matches the L1 NVCache block size, so all pointer chasing accesses reside in different consecutive L1 blocks. We observe a rapid sizable increase in the read latency at 44 blocks. This overflow point is close to L1 total blocks, so we believe this reflects the L1 NVCache overflow. This overflow point remains at 44 blocks under various stride sizes, which means that the mapping of blocks to cache sets does not depend on their addresses, indicating a fully-associative structure. These results also suggest that L1 employs a replacement policy different from Least Recently Used (LRU). An LRU cache should overflow when it is full, but L1 starts to overflow at 44 instead of 64 blocks (the maximum capacity of the L1 NVCache).

L2 NVCache set structure. When stride size is set as 4 KiB, which matches the L2 block size, every memory access falls into a different L2 block, and we observe an additional distinguishable latency change at 4096 blocks. The first overflow point at 44 blocks reflects the L1 NVCache capacity, while the second one matches the size of L2 NVCache ($4096 \text{ blocks} \times 4 \text{ KiB} = 16 \text{ MiB} = \text{Size}(\text{L2 NVCache})$). Unlike L1, the L2 overflow point changes with the pointer chasing stride size: it overflows at 4096 blocks with 4 KiB stride size, then at 2048 with 8 KiB stride, etc. The overflow point stops at 16 blocks with 1 MiB stride and does not change with larger stride sizes. This indicates that L2 is a set-associative cache because its overflow point changes and each L2 set has 16 cache blocks because the overflow point stops moving at 16 blocks. This result also shows L2 likely uses LRU replacement, as L2 starts to overflow with 4096 blocks which match the number of L2 blocks, i.e., L2 overflows when it is full. L2 NVCache is likely to use a linear indexing scheme because there is a linear relationship between the number of PC-Blocks and the stride size for L2 overflow points.

²64 B is the AVX512 register size we use and is unrelated to the NVCache parameters. SSE and AVX2 registers can also be used with non-temporal load/store instructions.

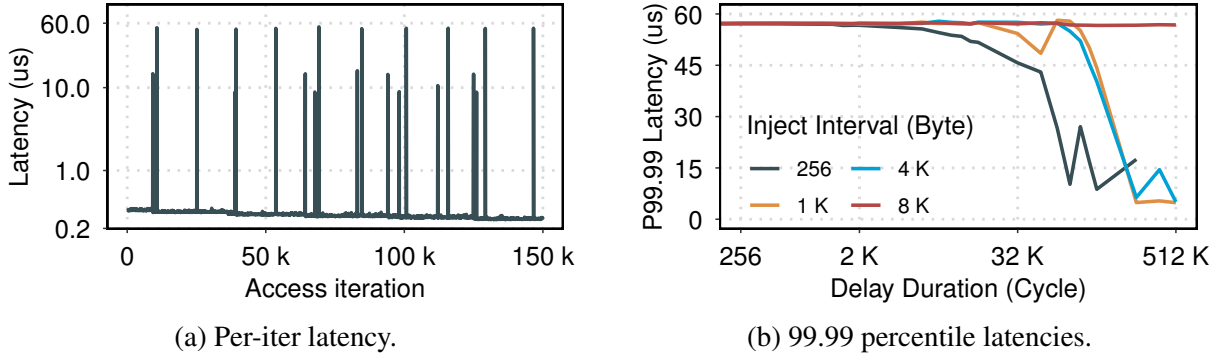


Figure 4.5: Overwrite microbenchmark results. (b) shows a sample of results from Figure 4.6.

4.3.3 Recovering Wear-Leveling Policy

Next, we recover the wear-leveling policy used in the Optane DIMM. To that end, we extend the *overwrite* microbenchmark from LENS [2] to measure the timing of the data migration and its spatial granularity (i.e., the block size).

Triggering a data migration. We start with a variant of the overwrite microbenchmark that repeatedly writes 256 B regions and measures the latency of each write operation. If a write in this region triggers a data migration, e.g., by a wear-leveling algorithm, the subsequent writes cannot be issued until the migration completes. As a result, a write stalled by a migration exhibits a latency over a magnitude higher than an average write. We estimate the migration latency using the elevated tail latency. Furthermore, we can calculate the migration frequency by measuring the time intervals between two consecutive migrations.

Figure 4.5a shows the results of the overwrite microbenchmark for data migration latency analysis. The results of Figure 4.5a show that long-latency writes (~ 60 us) happen at regular intervals. We see one after writing every ~ 4 MiB (~ 16 K write operations) to the memory region when using non-temporal stores. This confirms the behavior observed by previous work [2]. We also performed the same overwrite experiment on DRAM and did not observe such a long latency effect. Thus we consider the cause of this NVRAM long write latency is its data migration for

wear-leveling or thermal control.

Data migration timing conditions. To further investigate the conditions upon which a data migration happens, which is not covered by prior works [2, 169], we examine the effects of write frequency in triggering long delays. We inject delays with varying lengths in the overwrite microbenchmark and measure the 99.99 percentile latency. We can inject delays via two knobs: the delay duration, which indicates the number of CPU cycles that we wait during each delay, and the delay injection interval, which determines how much data we write between two consecutive injected delays. By changing the delay duration and injection interval and observing the tail latency, we can deduce the timing conditions for data migrations, i.e., the correlation between the data write frequency and the data migration frequency.

Figure 4.6 shows the results of this experiment. Each point represents a 99.99 percentile latency of the overwrite microbenchmark that writes 512 MiB data to the same 256 B NVRAM region. We observe that injecting longer delays reduces the frequency of long latency writes significantly. This might be due to the data migration process hidden in the delays. So, when the delay duration is too long, the migration process is likely to happen when we execute delays, and thus none of our writes encounters a long latency. In addition, we observe that overwrites trigger the maximum amount of long latencies (i.e., highest P99.99 latency) if the delay is infrequent (injection interval ≥ 8 KiB) or delay duration is short (duration < 32 K cycle).

Data migration spatial granularity. We run two threads writing data to NVRAM simultaneously but on different regions to detect the migration granularity, i.e., the block size for each migration. Two threads inject delays at different frequencies: one *scratch* thread injects no delays, and the other *probe* thread injects delays that do not trigger the full amount of long latency (the migration timing conditions identify this delay setting). When two threads are working within the same migration block, the probe thread observes the full amount of long latencies due to the scratch thread’s activity. Once two threads’ regions are distant enough to guarantee they do not

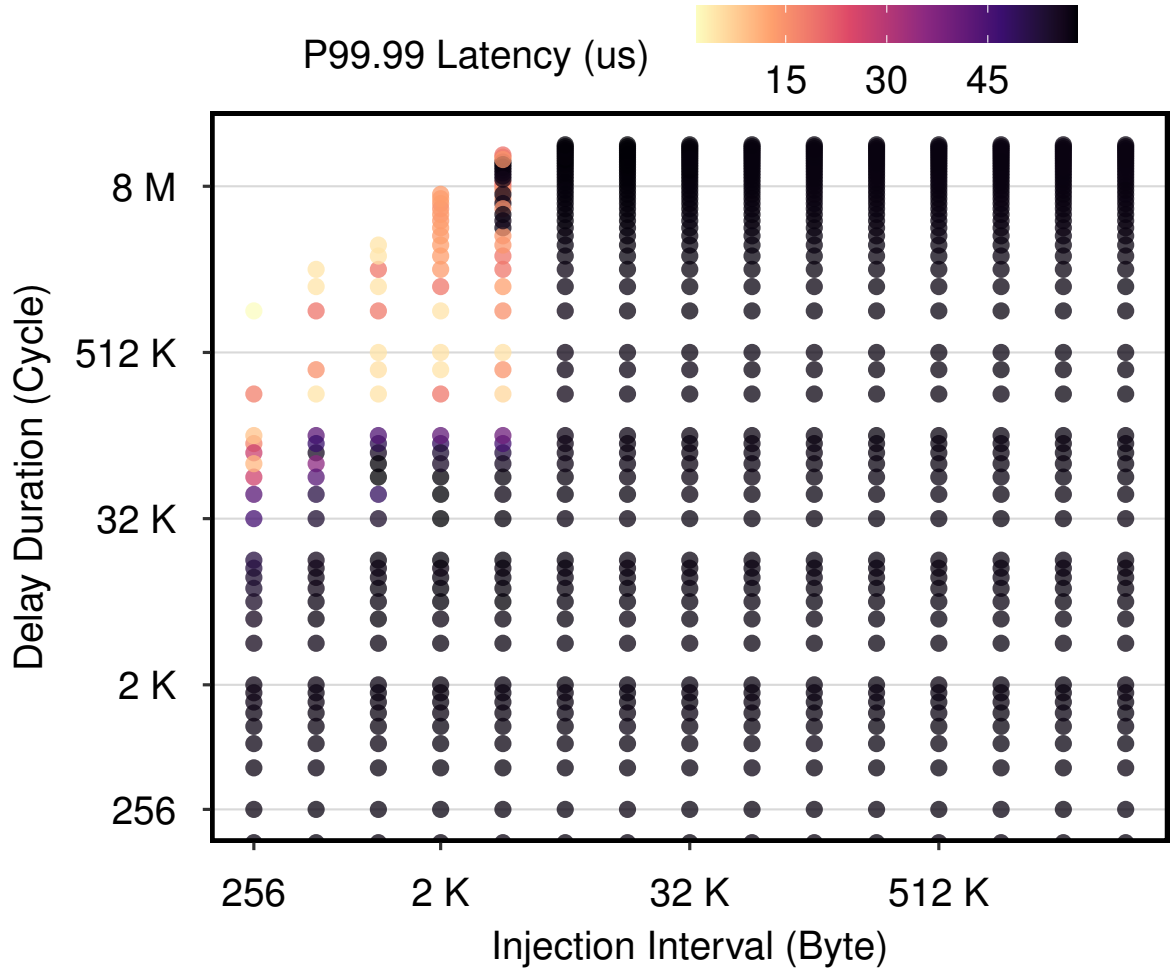


Figure 4.6: Overwrite 99.99 percentile (P99.99) latencies under various delay duration and delay injection interval settings.

share a migration block, the probe thread is the only writer working on its region, thus observing few long latencies.

On our machine, we set the probe thread to inject a 128 K cycle of delay after each 256 B write, gradually increase the delta between the two threads' starting addresses, and observe that the probe thread's P99.99 latency drastically drops when the delta reaches 4 KiB or larger. This indicates that two threads are now separated into two different data migration blocks. Thus, the migration block size is 4 KiB.

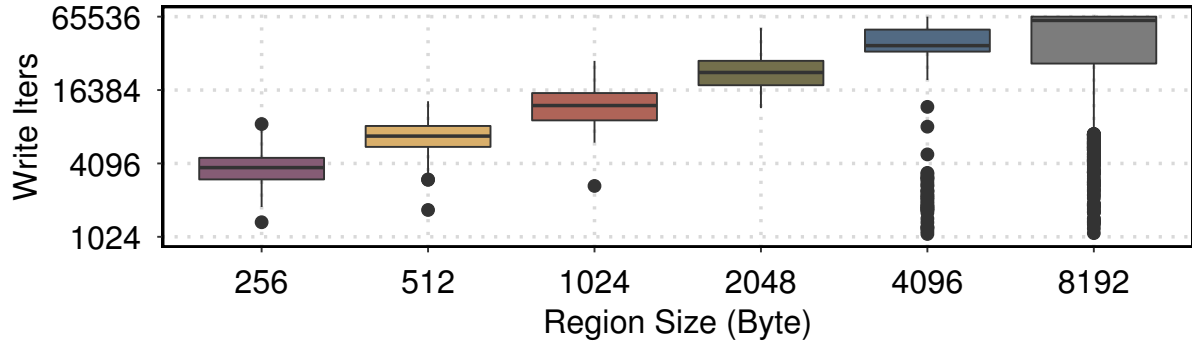


Figure 4.7: Data migration robustness when using various memory regions. The y-axis shows the number of 256 B writes it requires to trigger one data migration. Each bar shows the distribution of write iterations.

Data migration robustness. Next, we examine the distribution of the number of writes needed to trigger a single data migration. We repeatedly write to a memory region, where each repeat consists of sequential writes to each 256 B block within the region. We then count how many writes it requires to trigger one data migration (latency over 100 K CPU cycle). In this experiment, we use the `clwb` instruction instead of the non-temporal stores because `clwb` executes faster and generates more frequent writes to NVRAM, which trigger more frequent data migrations and give us a more accurate observation of the data migration robustness.

Figure 4.7 is a box plot that shows the results of the above-mentioned experiment. The figure shows that it is easier to trigger a migration when overwriting a smaller region, and at minimum, it takes $\sim 3,700$ iterations (i.e., $3,700 \times 256 = 925$ KiB) to trigger one data migration. This also shows that for a region size of 256 B, 50% of the time the required number of writes is within the range of $[3027 : 4552]$. The migration happens at 4 KiB granularity, i.e., one migration affects the entire 4 KiB page, but any of its 256 B blocks can trigger it. We also observe a more frequent data migration in Figure 4.7 (every ~ 4 K writes) compared to Figure 4.5a (every ~ 16 K writes) due to `clwb`'s faster execution than non-temporal stores.

Feasibility of side channel. We analyze the feasibility of establishing a side channel based on this data migration mechanism, using a proof-of-concept code: the victim and the attacker shares

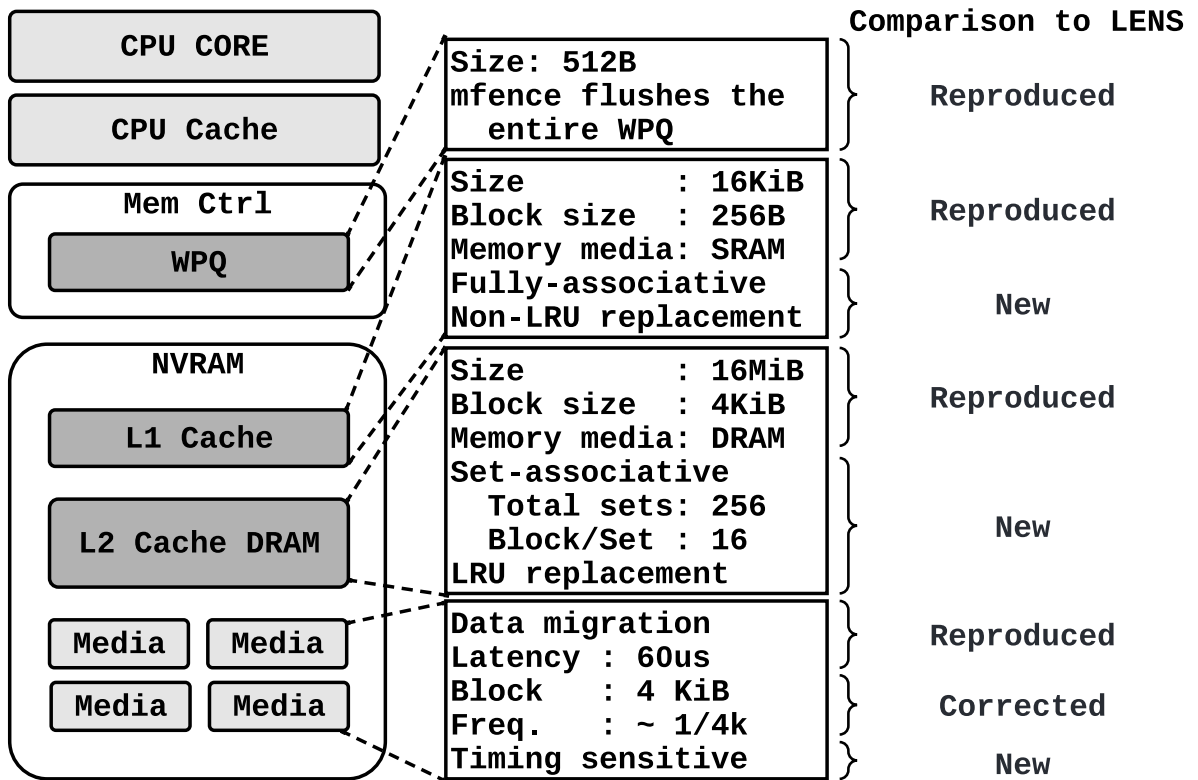


Figure 4.8: Overview of the NVRAM findings. *Reproduced* results are from LENS, *corrected* results are from NVLeak and correct observations from LENS, and *new* results are from NVLeak and not discovered by LENS.

the write access to a 4 KiB page and update different regions. The victim writes to its region at the highest frequency, while the attacker updates the region at a frequency that does not trigger data migration on its own. In this POC, the attacker observes frequent data migrations when the victim is updating the shared page, compared to when the victim is idle. We evaluated this POC and found that the attacker can detect if the victim has written more than one MiB of data to the shared page, which aligns with the data migration robustness analysis (it takes ~ 925 KiB writes to trigger one migration). This channel may leak the information when the victim application repeatedly updates metadata or a file: e.g., an attacker times a database row update to detect if there are updates to nearby rows by the victim.

4.3.4 Summary of Findings

Figure 4.8 shows an overview of our findings:

- Optane DIMM has two data caches (NVCaches), organized as two-level inclusive caches: The Level 1 NVCache is a fully-associative cache with 16 KiB size and 256 B block size. The Level 2 NVCache is a 16-way set-associative cache with 16 MiB size and 4 KiB block size.
- The size of the write-pending-queue [32] is 512 B.
- There is a long-latency effect (60 us) when frequently flushing data to NVRAM.
- The data migration happens after ~ 4 K writes to the same migration block of 4 KiB.

Compared to LENS [2], we have corrected a set of observations, and discovered previously unknown architecture designs, as shown in Figure 4.8.

4.4 NVLeak Covert Channels

Based on our findings from the previous section, we demonstrate covert channels across isolated security domains. We first demonstrate a cross-VM covert-channel attack based on the Prime+Probe attack technique [175] exploiting the on-DIMM NVCaches. We then demonstrate a wear-leveling-based covert-channel attack across filesystem boundaries. These attacks are demonstrated on Server A (Table 4.1).

4.4.1 Cross-VM NVCache Channel

This section presents our cross-VM covert-channel attack exploiting the level 2 NVCache.

Threat model. We assume the sender is a program running inside a guest VM that maps an NVRAM memory region from the host system (the virtualization hypervisor) to the guest VM.

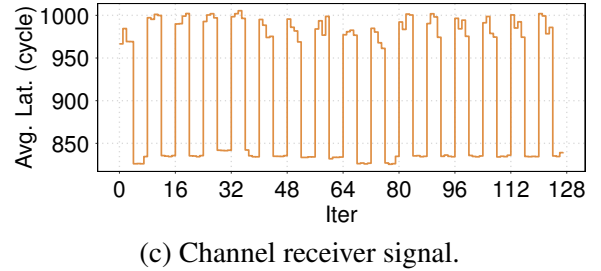
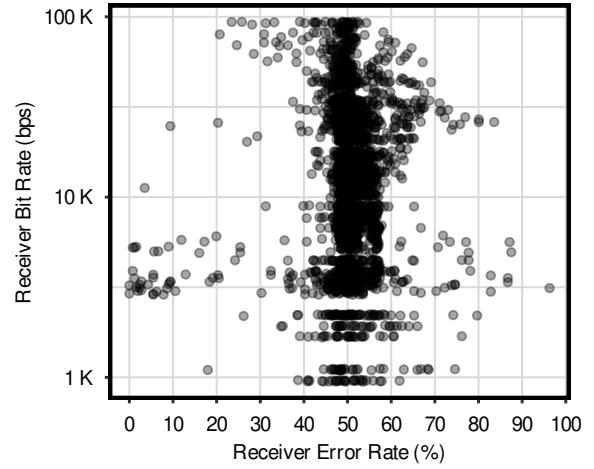
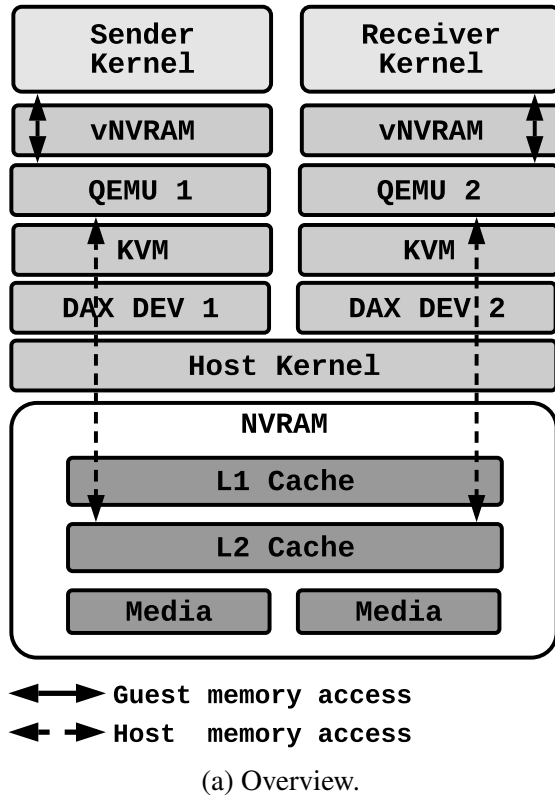


Figure 4.9: Cross-VM covert channel. (c) is a receiver signal from (b) that achieves 11 kbps bandwidth with 3.5% error rate. This channel uses 14 blocks, 1 MiB stride size and 16 repeat rounds. Y-axis shows the average per-block latency.

This region appears as a virtual NVRAM device in the guest VM. We assume the receiver is another guest VM that maps a different NVRAM memory region. We assume the sender does not have any shared memory with the receiver; their NVRAM memory regions are not overlapping and can belong to two different partitions, as long as these regions are both on the same NVRAM DIMM. We assume the sender and receiver have full control of their own guest VM’s software, i.e., they can replace OS kernels or run code at the kernel level within their guests.

Challenges. To obtain a stable, low-noise, and high-resolution channel, we overcome the following challenges:

Challenge 1: Bypassing L1 NVCache. Our reverse-engineering results (Section 4.3) showed that the L1 NVCache is a fully-associative on-NVRAM cache shared by all cores. Its fully-associative

property introduces noise and reduces the resolution of the channel. To bypass the L1 NVCache effect in our Prime+Probe attack, we must flush the L1 NVCache before every Prime step. To achieve this, we read data from an NVRAM region different from the Prime memory region. This guarantees the Prime step's memory accesses are cache misses in the L1, thus bypassing the L1 and directly accessing L2.

Challenge 2: Targeting L2 NVCache sets. We need to Prime and Probe each L2 NVCache set separately to construct a high-resolution channel. We rely on the knowledge of L2's indexing scheme we discovered in Section 4.3—L2 takes bits [13:20] of each address—and use it as an index to a set. Thus, to evict a specific L2 set, we first identify the set address of the target L2 set, then create our conflict set with a set of 16 blocks that are based on the set address and are 1 MiB apart. This ensures that all of these blocks go to the desired set.

Challenge 3: Avoiding noises due to cache prefetchers. There are multiple prefetchers for on-CPU caches and on-NVRAM caches in our system. These prefetchers cause latency fluctuations in the Prime+Probe accesses, adding noises to the channel. To defeat these prefetchers, we use our pointer chasing access pattern, which accesses all memory blocks, but in a random order so that prefetchers cannot detect a predictable pattern.

Challenge 4: Synchronizing the parties. Without synchronization, the sender and receiver can easily drift, making the received bit sequence unreliable, especially for long bit sequences. To overcome this challenge, we implement a timing-based synchronization, where the sender and receiver use the same predefined time interval for each bit send/receive iteration: After the pointer-chasing operation finishes, the thread waits until the time interval is finished, then starts the next iteration. From our evaluation, this timing-based sync introduces $\sim 10\%$ additional waiting time compared to the pointer-chasing operation time.

Evaluation. We configure a single-DIMM NVRAM into two non-overlapping DAX devices (Figure 4.9a). Each DAX device is passed to KVM and QEMU to create a virtual NVRAM for

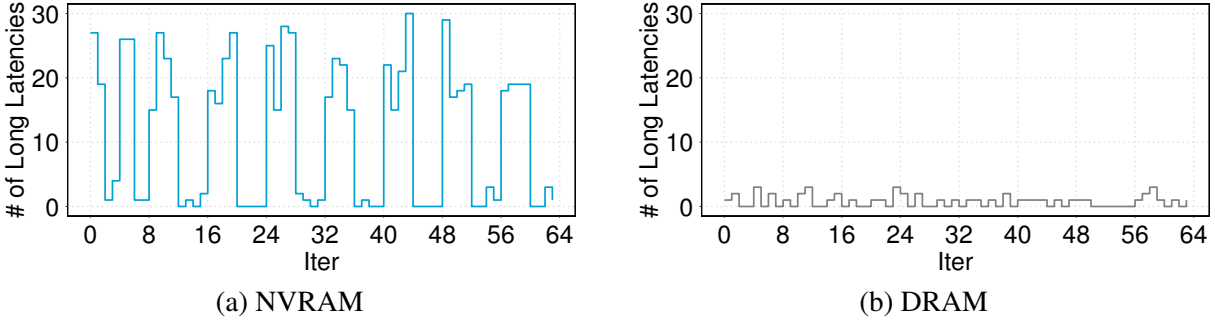


Figure 4.10: Filesystem inode-based covert channel on NVRAM and DRAM.

the guest VM.

We use the KVM-unit-tests framework [185] to implement two OS kernels for the sender and receiver to run at ring-0 within the guest VM. KVM-unit-tests provide a set of libraries to build customized OS kernels, and each of its test cases is a small kernel that boots up from a guest VM and runs a piece of testing code. We implement the sender and receiver as two test cases in KVM-unit-tests that run covert channel code at kernel level or ring-0 level.

Figure 4.9b shows the performance of the cross-VM covert channel. Each point in the figure represents one covert-channel experiment that sends/receives a binary data sequence. Different points use different configurations for the underlying pointer-chasing microbenchmark, including region size, block size, stride size, repeat rounds, and flushing L1 NVCache or not. From Figure 4.9b we find the best configuration achieves 11 kbps bandwidth with 3.5% error rate (Figure 4.9c).

4.4.2 Data-Migration-Based Channel

This section describes how we construct a covert channel that exploits the NVRAM-specific wear-leveling data migration mechanism.

NVRAM data migration. As described in Section 4.3, the Optane DIMM uses a wear-leveling data migration mechanism that triggers long latencies in response to frequent writes to the same

migration block. This migration mechanism is sensitive to the memory write frequency: the migration happens less frequently (relative to the write traffic) once the write frequency drops to a certain point (e.g., one access per 32 K cycles as shown in Figure 4.6).

We exploit the migration latency to construct a covert-channel attack where the receiver and the sender write to the same NVRAM data migration block. The receiver uses a low-frequency write and measures write latencies. If the sender frequently writes to the same migration block, the block gets migrated more regularly, which results in long-latency writes detected by the receiver.

This attack requires the receiver and the sender to write the same 4 KiB migration block. However, this write access does not need to be direct. For example, we can assume a receiver and a sender own two separate files, but the files' metadata resides in the same migration block. In such cases, any write to the sender's file results in a metadata update, and the metadata update involves a write to the migration block. Neither thread has direct access to the metadata, but they leak information through underlying filesystem operations.

Threat model. We assume the receiver is a user-space program with access to a file. This file's *inode*—Linux filesystems metadata—is stored in a metadata page that also stores the sender's file inode. The filesystem maintains the inode, and the user space program cannot directly modify a file's inode. However, a user space program can use (1) `futimes()` to update a file's access and modify time, or (2) `ftruncate()` to update the file size, without the overhead of an actual file data update. The size of an inode is typically sub-page-size, e.g., inodes in the EXT4 filesystem are 256 bytes [186]. Thus one metadata page (4 KiB) stores multiple inodes, enabling the receiver to access a file that shares a metadata page with the sender's file.

Cross-filesystem covert channel. We construct a covert channel based on the filesystem's inode updates triggered by the data migrations. The sender program updates its file's metadata at maximum frequency if sending bit 0; otherwise, it remains idle for a predetermined period to

communicate bit 1. The receiver constantly updates its file at a low frequency to not trigger data migrations on its own but to detect that of the sender. The receiver also times these inode update functions and identifies the data migration as a long latency. The receiver picks up a bit 0 if it detects data migrations.

The widely-adopted EXT4 filesystem allocates inodes sequentially, i.e., for each newly created file, EXT4 allocates the first available inode from a list of free inodes sorted by the inode number. The receiver can exploit this allocation mechanism to create its file at roughly the same time the sender creates its files so that their inodes are placed on the same metadata page with a high probability.

Evaluation. In our evaluation, we configure the NVRAM into filesystem DAX mode and create an EXT4 filesystem on the NVRAM. We mount the EXT4 filesystem in DAX mode, which disables the filesystem page cache and allows direct access to the underlying NVRAM. This is the preferred use case of a filesystem for NVRAM [187].

Figure 4.10a shows the receiver signal on an NVRAM-based inode channel, where the sender sends `0xcc` followed by repeated `0xf0`. In this example, this channel achieves 217 bps bandwidth with 0% error rate. The bandwidth can be further improved to 1.5 kbps if we reduce the number of inode updates for each bit. For comparison, we also evaluate the possibility of covert communication if we use DRAM instead of NVRAM. As shown in Figure 4.10b, the DRAM does not achieve a stable channel as it does not implement the data migration for wear-leveling.

4.5 NVLeak Side-Channel Attacks

We demonstrate three types of attacks on Server A (Table 4.1) where victims access NVRAM through different interfaces: (1) attacking a database which accesses NVRAM through conventional filesystem interface, where the attacker exploits L2 NVCache to learn information about SQL queries (Section 4.5.1); (2) attacking an NVRAM-optimized key-value store which

```

-- U1: Update 100 records in 'info' table
UPDATE info SET name = "New Name" WHERE npi == 1144223363;
-- U2: Update 1 record in 'address' table
UPDATE address SET city = "New City" WHERE npi == 1144223363;
-- C1: Count records in 'address' table
SELECT COUNT(*) FROM address WHERE LOWER(city)='athens';
-- C2: Count records in 'info' and 'address' tables
SELECT COUNT(*) FROM info, address WHERE
    info.npi = address.npi AND LOWER(city)='athens';
-- C3: Count records in 'info' and 'address' tables
SELECT COUNT(*) FROM info, address WHERE
    info.npi = address.npi AND LOWER(city)='houston';
-- Q1: Query both tables
SELECT * FROM info, address WHERE
    info.npi = address.npi AND info.npi == 1144223363;
-- I1: Insert 10,000 records in 'info' table
INSERT INTO info values (...), (...);
-- S1: Sort records in 'address' table
SELECT * FROM address WHERE LOWER(city)='athens'
    ORDER BY city DESC LIMIT 1;

```

Figure 4.11: Evaluate SQL operations.

accesses NVRAM through NVRAM-aware filesystem and libraries, where the attacker learns the in-flight key-value pair information (Section 4.5.2); (3) attacking a victim program that links with shared libraries that are stored in NVRAM and accessed without page cache, where the attacker exploits L1 NVCache to learn the victim code execution path (Section 4.5.3).

4.5.1 Database Operation Leakage

This section describes an NVCache-based side-channel attack that can violate the privacy of encrypted database applications.

Threat model. Motivated by previous work on cache-based side-channel attacks on encrypted databases [146], we assume the victim is a database application that accesses data stored in NVRAM, and the attacker runs in another process or virtual machine. We assume an attacker who does not have direct access to the database and whose goal is to learn about the queries that the victim runs. The attacker repeatedly probes all the NVCache sets to detect the victim’s NVRAM access pattern. The attacker first records access patterns of different database SQL operations and uses them to categorize newly detected patterns.

System and software configuration. We configure an Optane DIMM into two partitions, one for the victim database to store files, and the other one for the attacker to probe the NVCache. We use sqlite3 [188] as the victim database, and create the database tables from the NPPES dataset [189], which results in a 286 MiB database file. There is an *info* table with users’ basic info (e.g., full names) and National Provider Identifiers (NPI) as primary keys. There is another *address* table that stores user addresses, including city and state, and NPI as a foreign key referencing the info table.

Leak SQL statements. Figure 4.12a to Figure 4.12g show the access pattern under different victim database operations (Figure 4.11), detected by the attacker program. U1 and U2 are updating to two different tables, and their access patterns are visually different: U1 only affects 1-2 attacker iterations, while U2 affects 70 iterations. U1 and U2 also have different latency distributions, as shown on top of the access pattern figure. U2’s latencies are more concentrated around 900 cycles, while U1’s are more evenly distributed. All other operations have different access patterns and latency distributions.

We use the k-nearest neighbors algorithm [190] (k-NN) with Pearson correlation coefficients to analyze and categorize these access patterns quantitatively: We first identify a latency threshold range to filter out the non-relevant latencies and then encode the latencies into a binary array, where bit 1 represents the corresponding latency within the threshold. We use this binary

array as the *feature* of the corresponding database operation.

To categorize a newly detected database operation, we calculate the Pearson correlation coefficients between its feature and previously-recorded features, and use their coefficients as *distances* in the k-NN algorithm. We categorize this newly detected operation as one of the previously-known operations that takes the majority of the k nearest neighbors of the new operation. To evaluate this categorization algorithm, we run each SQL operation 100 times, then for each operation, randomly choose 70 samples as the k-NN training set and use the rest 30 samples as the test set. In this evaluation, 240 SQL operations are tested and 202 of them are correctly categorized, resulting in an accuracy of 84%.

Leak SQL range query. We then demonstrate that the NVCache-based side channel can leak more detailed information in database accesses. As demonstrated by previous work [146, 172], leaking range query details is specifically critical for encrypted databases where learning about these queries can almost result in a complete loss of privacy. As an example, we assume the victim executes a range query (Figure 4.13a) based on the creation date of NPPES data records; And the attacker performs the side-channel attack to leak the victim’s memory access pattern.

This range-query attack improves the attack resolution and analysis correctness compared to the SQL execution attack (Figure 4.12): (1) in the range query attack, the attacker probes the memory in a per-set order, i.e., the attacker triggers the victim SQL query repeatedly and probes different NVCache sets for each repeat. Compared to the previous per-iter probing that sequentially probes all NVCache sets during each iteration, this per-set probing runs at a much higher frequency ($256\times$ faster) and retrieves information with a much higher resolution. (2) the range-query attack uses a two-dimension Pearson correlation as distance in the k-NN algorithm to categorize results, i.e., the attacker calculates the correlation for each cache set and uses the average correlation coefficients as k-NN distances. This improves the categorization quality because range query attacks generate much more iterations, leading to high-noise results when

using the one-dimension correlation.

Figure 4.13b to Figure 4.13e shows the side channel results of four victim queries using different date ranges. The color represents the memory access latency, which further indicates the victim’s memory access in the corresponding NVCache set: Darker color represents a higher attacker latency and indicates the victim has more memory accesses in the cache set. These figures show that these queries have different memory access patterns in terms of access addresses and timings. We run each ranged query 100 times and use 70% of the results as training samples for the k-NN algorithm. In this evaluation, 120 queries are tested and 86 of them are correctly categorized, resulting in a 72% accuracy.

4.5.2 PMDK Key-Value Store Leakage

We mount the NVCache-based side channel attack to leak information from NVRAM-optimized applications.

NVRAM-aware applications. NVRAM-aware applications are designed to leverage NVRAM features, including byte-addressability, persistence, and DRAM-like performance. These applications (1) issue memory load and store instructions to access NVRAM instead of relying on filesystem read/write APIs; (2) issue cache line flush and memory fence instructions to flush data to NVRAM for persistence, instead of using conventional `fsync()`; and (3) configure the filesystem to bypass the DRAM page cache, thus ensuring CPU data is directly flushed to NVRAM, without relying on `fsync()` to flush the page cache.

It is non-trivial to implement such applications from scratch as the programmer has to manually insert the flush and fence instructions to proper code sites. Thus, programmers typically rely on NVRAM programming libraries that abstract these low-level operations and provide library functions to reduce the NVRAM programming complexity. PMDK [65] is one of the most mature and widely-adopted NVRAM programming libraries.

In this attack, we leak information from the applications that use PMDK to access NVRAM.

Threat model. The Threat model is similar to the database operation leakage (Section 4.5.1). The major difference is the victim in this attack is an NVRAM-aware application, which inherently flushes data to NVRAM at a high frequency to maintain fine-granularity crash consistency. This frequent memory access pattern enables attackers to detect more detailed information compared to the conventional filesystem access approach (Section 4.5.1).

Leak PMDK KV store accesses. We target an example key-value store application provided by PMDK developers [191]. This example uses the libmemobj-cpp library [192] and stores key-value pairs in NVRAM. To access them, it uses the PMDK library to achieve per KV-pair persistence and crash consistency. We mount the NVCache-based side-channel attack on this KV-store and leak which specific key-value pair is being accessed: Figure 4.14 shows four example memory access patterns that are visually distinguishable. These patterns can be categorized using the k-NN categorization described in Section 4.5.1. We evaluated 8 different key-value updates, where each operation is executed 100 times and 70% of the results are used as the k-NN training set. In this evaluation, 182 out of 240 samples are correctly categorized, resulting in a 76% accuracy.

In conclusion, by targeting the NVRAM-aware victim applications that directly access the NVRAM, the attacker can detect a detailed NVRAM access pattern that is not filtered by the DRAM page cache, thus further learning high-resolution victim activities.

4.5.3 Code Execution Path Leakage

Motivated by the NVRAM memory mode (Figure 4.1), this section describes an attack that leaks code execution paths, where the attacker leverages the L1 NVCache to detect which

shared library function is being executed by the victim.

Threat model. We assume the NVRAM is running in the memory mode (Section 4.2.1), acting as a large system memory instead of persistent storage. We assume the attacker and the victim share library code pages, similar to the Flush+Reload attack [177], and shared code pages reside in the NVRAM and are cached in the DRAM. The attacker repeatedly (1) flushes the shared library code pages to NVRAM, (2) waits for the victim’s activities, (3) then reads the code pages back and times page loads. Whenever the victim calls a shared library function, the corresponding code page is loaded, which is further detected by the attacker as a low latency page load, thus leaking the victim’s execution pattern of the shared library code.

System configuration. Instead of setting up the entire system to use NVRAM in the memory mode, we emulate this attack environment based on existing settings: The shared library is stored in an NVRAM-aware filesystem which enables direct access to the NVRAM and bypasses the DRAM page cache. This emulates the library code stored using the memory mode, just without the DRAM cache, as the attacker can follow prior approaches [193] to flush the code pages from DRAM. To achieve high temporal resolution, the attacker assumes the victim mostly runs on a certain CPU core, which is a realistic assumption. We emulate this setting by pinning the victim process to one CPU core.

Temporal resolution. Figure 4.15a shows the victim code, which calls the library function, flushes the code page, and waits for predefined cycles. And Figure 4.15b shows the attacker code that times the library code page load, flushes the code page, and then flushes the L1 NVCache to prepare for the next iteration. Figure 4.15c shows that the attacker can detect victim function calls that take ≥ 22 us, with a call interval ≥ 5 us. The attacker can detect function calls shorter than 22 us with a risk of losing precision. This proof-of-concept shows that even if systems deploy mitigations for protecting on-chip side channels [159, 161], an attacker can still measure

non-constant time code with relatively high spatial (256 B) and temporal resolution (22 us).

Feasibility to attack cryptographic operations. Based on the code execution leakage, we analyze the feasibility of attacking real cryptographic code, the wolfSSL library [194] version 4.2.0. We identified a secret-key-dependent branch in wolfSSL’s `_fp_exptmod()` function as part of the RSA private key decoding process³. This function loops over the key material of an RSA private key using a sliding window algorithm which calls three extra functions in a set of secret-dependent `if` statements, `fp_sqr()`, `fp_mul()`, and `fp_montgomery_reduce()`, depending on secret values. We further assume the victim is vulnerable to code page flushes [193]. Hence, the attacker can flush the shared code pages and apply the code execution analysis on these functions, to detect when secret-dependent branches are taken. The victim runs the full RSA decoding algorithm instead of just the isolated functions vulnerable to attacks.

Figure 4.15d shows the attacker’s latency measurements of the `fp_sqr()` function, where low latencies indicate the victim calls of this function. In this example, the victim has called this function 275 times, and the attacker detected 225 calls. We repeat this attack 100 times with different secret keys and find that the attacker detects approximately 77% of the calls.

Recovering RSA private key bits. The detected function calls indicate that an attacker can recover significant parts of the key material from such non-constant-time implementations: The attacker monitors all three above-mentioned secret-dependent functions to learn: (1) the first bit of each 6-bit sliding window data, and (2) all the zero bits between sliding windows. This on average recovers 28.55% bits from 1000 randomly generated 2048-bit RSA keys. According to prior attacks exploiting partial information [195–197], a partial RSA key (with $\geq 27\%$ bits recovered) has a high probability for an attacker to recover the full RSA key using the branch and prune technique [195].

³We disable wolfSSL’s `harden` feature to attack the original version of this function, which is vulnerable to timing side channels.

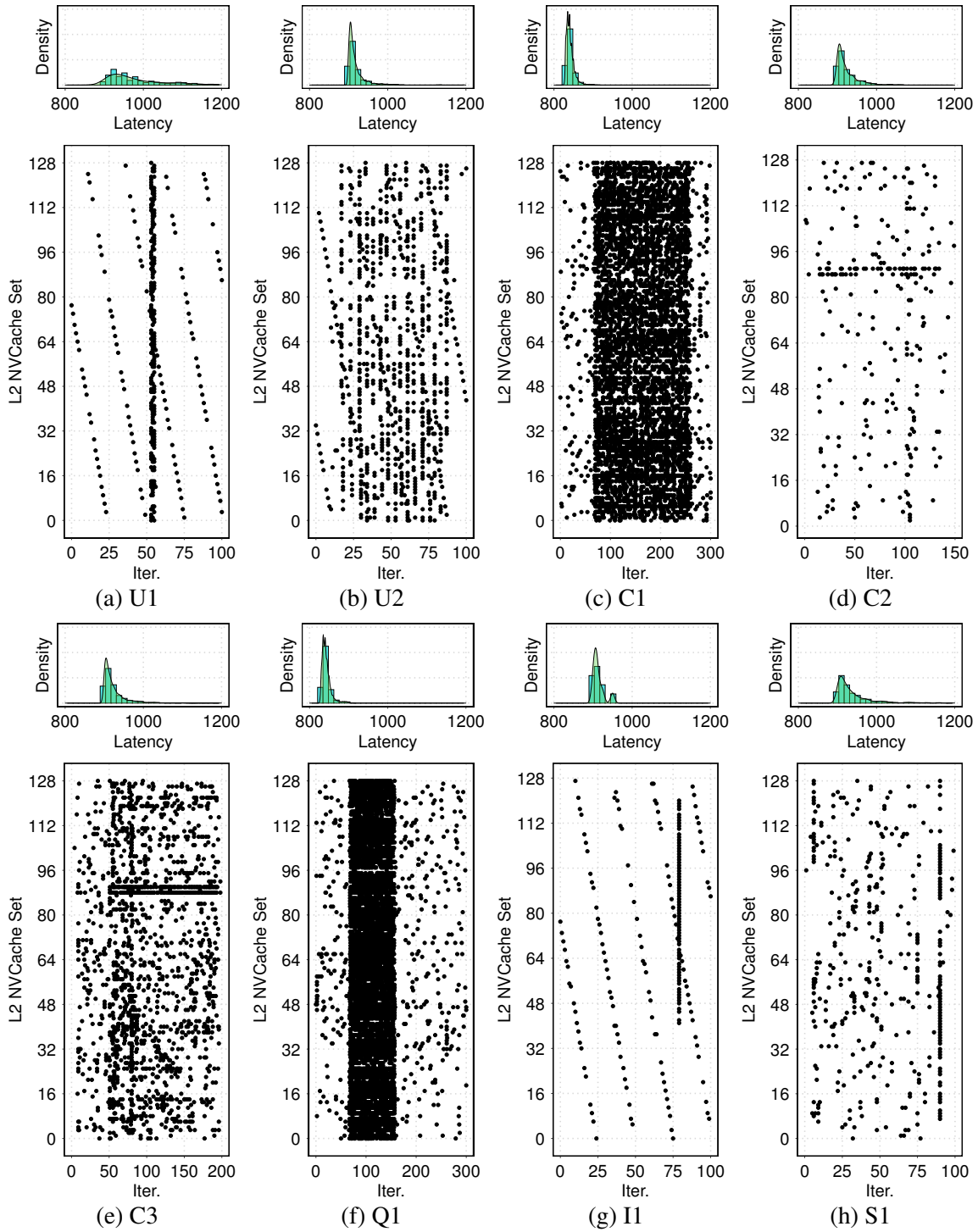
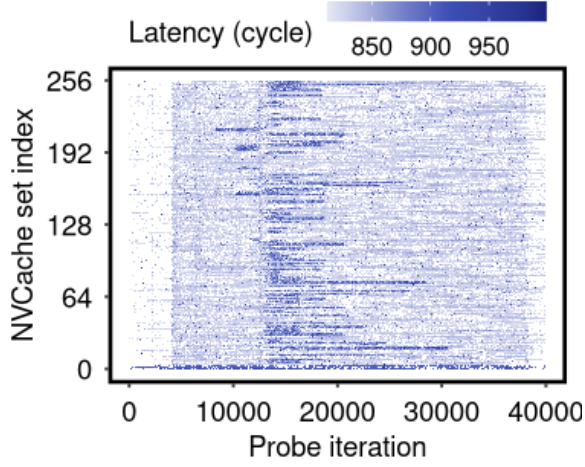


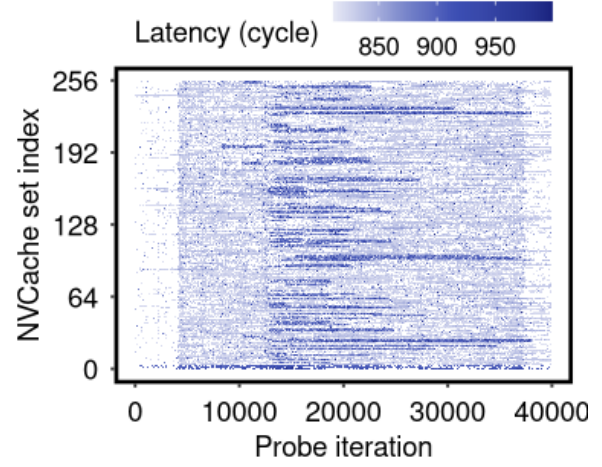
Figure 4.12: Access patterns of database operations. (a)-(h) each shows the access pattern for a database operation: the bottom figure shows the latencies in a concentrated range of iterations and sets, and the top figure shows the latency distributions.

```
-- Query records using a ranged condition
SELECT * FROM info WHERE create_date BETWEEN 06-01-2005 AND 07-01-2005;
```

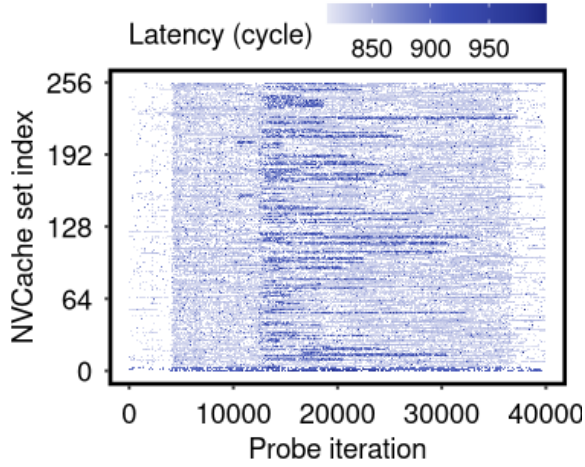
(a) Ranged query (1st month)



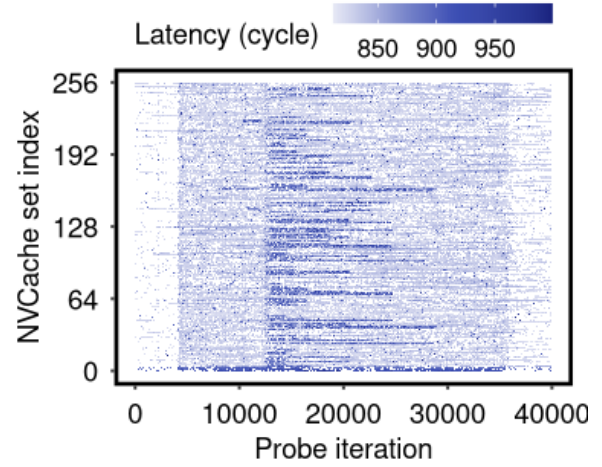
(b) 1st month



(c) 2nd month



(d) 3rd month



(e) 4th month

Figure 4.13: Memory access pattern of SQLite ranged query.

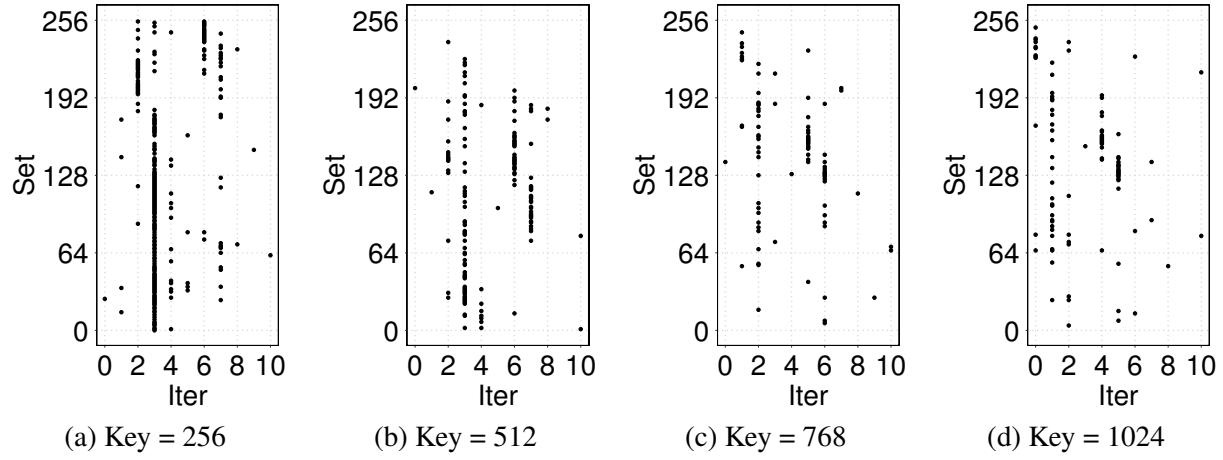


Figure 4.14: PMDK key-value store's memory access pattern when updating different key-value pairs.

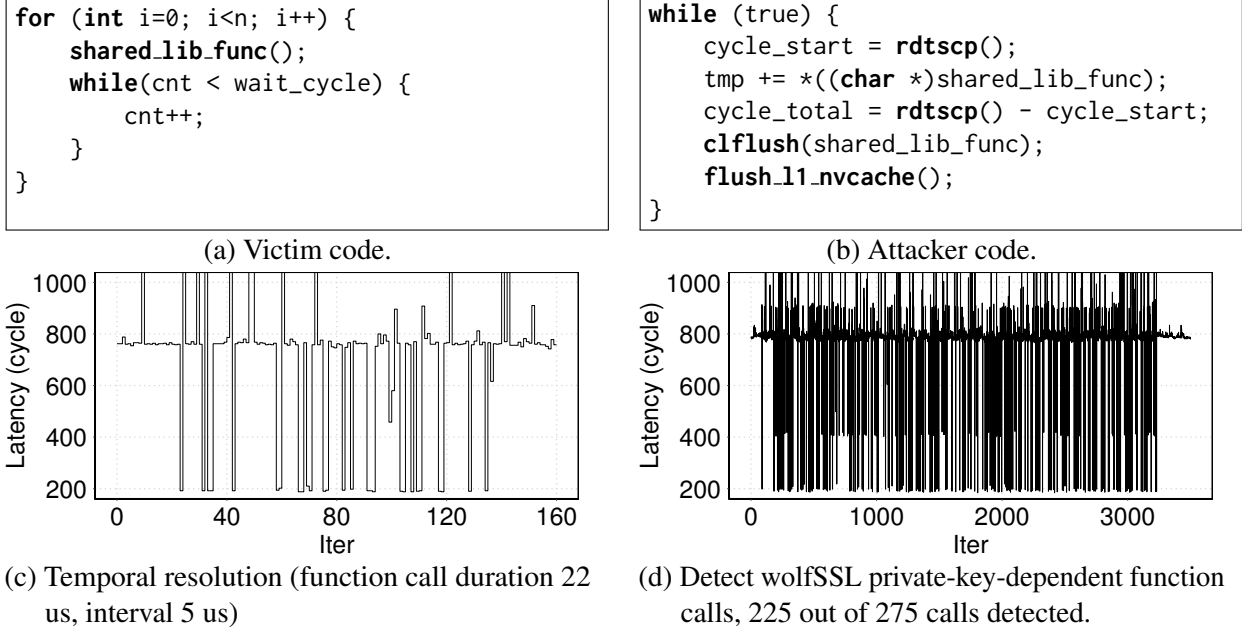


Figure 4.15: Code execution leakage and wolfSSL analysis.

4.6 Mitigations

We propose three mitigations based on reverse engineering results (Section 4.3) and attacks (Section 4.4 and Section 4.5). We first propose a software-based partitioning mechanism to protect L2 NVCache and evaluate its effectiveness and performance. We then propose a software mitigation and a hardware mitigation to protect the NVCache hierarchy, based on prior works that are proved to be effective in defending against cache side-channels [198–202].

Software-based mitigation for L2 NVCache. The NVLeak attacks are made possible mainly because the performance of a security domain can be influenced by the memory accesses of another as they compete for shared on-DIMM resources such as L1 or L2 NVCaches. In particular, in our attacks, the victim and the attacker can compete for a single set in the L2 NVCache. Therefore, we propose a simple yet effective software-only mitigation scheme that isolates the L2 NVCache sets that belong to disjoint security domains. Since we know the exact indexing scheme used by the Optane DIMM (Section 4.3), we can modify the memory allocation such that the memory of a process can be mapped to specific sets in the L2 NVCache. This software mitigation leverages the set-associativity and thus only protects the L2 NVCache.

We implement the NVLeak mitigation at the persistent memory library level where all the persistent memory allocations are handled by PMDK [65]. We modify the PMDK’s memory allocator, `make_persistent()`, to take an additional bitmap parameter that specifies a list of NVCache sets to be used. The allocator then allocates memory in the next block whose NVCache set is 1 in the bitmap. Hence, the allocator avoids the L2 NVCache sets that may leak information to an attacker.

We converted PMDK persistent data structures to use this secure memory allocator, and then converted the key-value store (Section 4.5.2) and a PMDK benchmark suite [192] to use the secure data structures.

Figure 4.16a shows how the memory allocated to different keys in the PMDK key-value

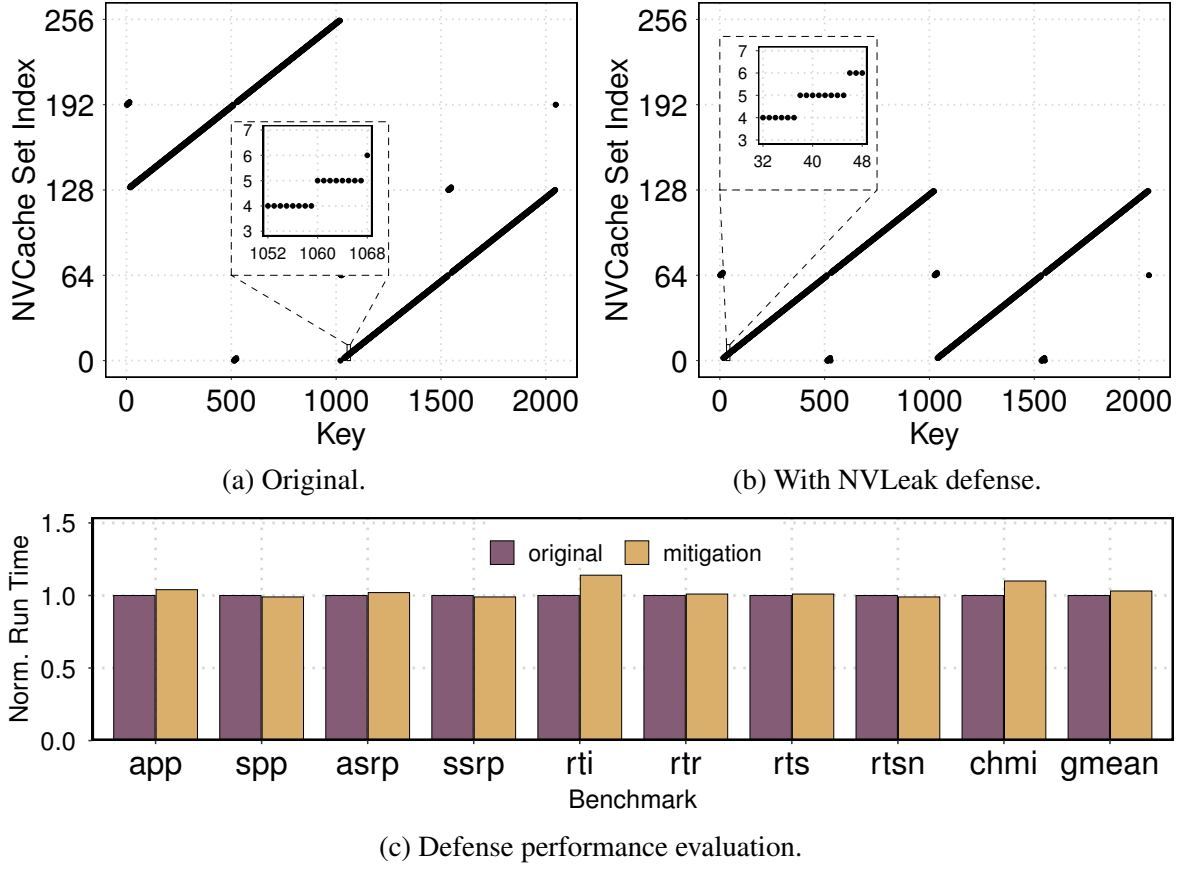


Figure 4.16: NVLeak mitigations applied to (a-b) PMDK key-value store and (c) a PMDK benchmark suite.

store is mapped into different sets in the L2 NVCache. Without our defense, the memory allocation is unrestricted, and any key can be mapped into any NVCache set. However, when we enable our defense (Figure 4.16b), the memory allocator only allocates the memory regions that are guaranteed to be mapped into the lower sets in the L2 NVCache, eliminating any potential contention. We then perform the side channel attack described in Section 4.5.2 and confirm that the attacker is not able to detect the victim’s L2 NVCache activities when applying this mitigation. Figure 4.16c shows the performance evaluation of our defense using a PMDK benchmark suite. The full description of the benchmarks can be found in Table 4.2. On average, our partitioning mitigation incurs less than 4% performance overhead.

Table 4.2: PMDK Benchmarks in Figure 4.16c

Name	Data Structure	Operation
app spp	Persistent Pointer	Assignment Swap
asrp ssrp	Self-Relative Pointer	Assignment Swap
rti rtr rts rtsn	Radix Tree	Insert Remove Search Search (key not present)
chmi	Concurrent Hash Map	Insert

Software-based mitigation for WPQ and L1 NVCache. The WPQ and L1 NVCache leakage can be mitigated in software by flushing the WPQ and L1 NVCache before accessing secret data in NVRAM, or randomizing memory accesses in the program. This software mitigation can significantly reduce the possibility of leakages in these fully-associative structures [198], while the hardware mitigation is preferred to prevent these leakages at a lower performance cost [202, 203].

Hardware-based mitigation. To prevent information leakage in NVRAM, the hardware and software need the ability to distinguish memory requests from different security domains at the NVRAM DIMM level. This is similar to the Intel Cache Allocation Technology [204] (Intel CAT), which dynamically partitions CPU caches for different processes. Thus it's possible to extend the Intel CAT to partition the WPQ and NVCache, and provide a software interface for processes to allocate their private partitions. This prevents the sharing of these queue and cache structures and thus prevents the corresponding information leakages [199–202].

4.7 Discussion

In this paper, we present the reverse engineering and attacks in NVRAM systems, demonstrate the results using Intel Optane DIMM, and propose mitigation designs. These techniques and ideas are generic and can be applied to investigate future hardware designs because: (1) Future NVRAM systems are likely to adopt similar design choices as Intel Optane DIMM, such as using on-NVRAM cache to fill the performance gap between fast interconnection and slow NVRAM media. These designs are likely to be vulnerable to off-chip attacks similar to the ones described in our paper. (2) Our tools do not rely on specific CPU instructions to access Intel Optane DIMM, thus can be used on future NVRAM devices as long as they support direct access through CPU memory load/store instructions (e.g., using CXL [101] technology). We hope our methodologies and tools influence the future NVRAM system design choices.

4.8 Related Work

To our knowledge, we are the first to develop covert and side-channel attacks on real NVRAM systems. This section discusses related works.

NVRAM characterization. Wang et al. [2] are the first to reverse engineer the Optane DIMM architecture with LENS, an NVRAM reverse engineering tool. That work revealed the NVRAM on-DIMM buffer sizes and their block sizes, but failed to reveal more detailed structures such as cache set sizes and indexing schemes. Zhang et al. [169] reverse engineered Optane DIMMs with GORDON – an FPGA-based profiling tool – and confirmed most of the LENS observations. Xiang et al. [205] investigated the on-DIMM buffer structures on the first and second generations of Optane DIMM and confirmed many observations from LENS [2]. Several recent studies [1, 171] observed that repeated writes to a concentrated Optane DIMM memory area lead to high write latency. Compared to these prior works, our paper depicts more detailed NVRAM architectural

designs that raise security concerns; our paper also revises several inaccurate conclusions from these previous works.

NVRAM architecture security. Liu et al. [206] presented a set of side-channel attacks on Optane DIMM, which is a concurrent work. Previous NVRAM architecture security works primarily focus on protecting NVRAM data integrity: Triad-NVM [207] proposed a new persistence mechanism of Merkle Tree and encryption counters to enable secure recovery of NVRAM data. SuperMem [208] facilitated a write-through counter cache to guarantee the security and atomicity of data writes at a low overhead. Freij et al. [209] proposed a set of optimizations to reduce the overhead of persisting the Bonsai Merkle Tree that is used to encrypt the NVRAM data. Xu et al. [210] proposed a framework to mitigate persistent memory object corruption vulnerabilities.

Off-chip microarchitectural side channels. Most previous attacks that bypass cache side-channel defense focus on on-chip components [158, 163–167, 211], but our work is an off-chip attack that directly affects the applications running on the CPU, so we will discuss such attacks further. Pessl et al. [168] showed that the DRAM row buffer could be used to spy on memory operations. However, a single row buffer with 8 KB spatial resolution is shared across many workloads on the system, which makes attacks relying on them highly noisy and impractical. Gruss et al. [193] demonstrated the page-cache attack, which essentially exploits the contention of memory pages within the DRAM. Page-cache attacks can be combined with NVLeak when NVRAM is used in memory mode. On the other hand, page-cache attacks are noisier than attacks based on the set-associative L2 NVCache and have a lower spatial resolution than the L1 NVCache with 256 B granularity. Aside from the off-chip DRAM, attackers can also exploit accelerators such as GPU [212] and FPGA [213] connected to the memory subsystem over PCIe. A major limitation of these attacks compared to NVLeak is that the memory shared by these accelerators and the CPU is generally not as privacy/security-sensitive as the file system and code pages, which are backed by NVRAM.

4.9 Conclusion

We conclude that NVLeak is a practical concern for the security of NVRAM storage. We demonstrate that attacks based on NVRAM’s cache structure can break cross-core and cross-VM isolation and leak information from real-world applications such as databases, NVRAM-aware key-value stores, cryptography libraries, and more; We also demonstrate a leakage channel based on NVRAM-specific wear-leveling mechanism, which breaks the filesystem isolation and leaks Linux file metadata updates. Our empirical attack study motivates the development and deployment of new defense against NVRAM-based microarchitectural side channels, as we also came up with such mitigation. Ultimately, we hope that our work encourages the community to investigate more into the security of NVRAM and, in general, more holistic approaches to prevent side channels on complex computing systems.

4.10 Acknowledgements

Chapter 4, in full, is a reprint of the material as it appears in USENIX Security Symposium, 2023. Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, Jishen Zhao. “NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems”. The dissertation author was the primary investigator and author of this paper.

4.A Additional Reverse Engineering Results

4.A.1 Additional Results on Server A

We present the majority of the Server A (Table 4.1) reverse engineering results in the main paper, this section provides additional results of NVCache reverse engineering: Figure 4.17a to Figure 4.17o are NVLeak’s strided pointer chasing results, which are detailed illustrations of the

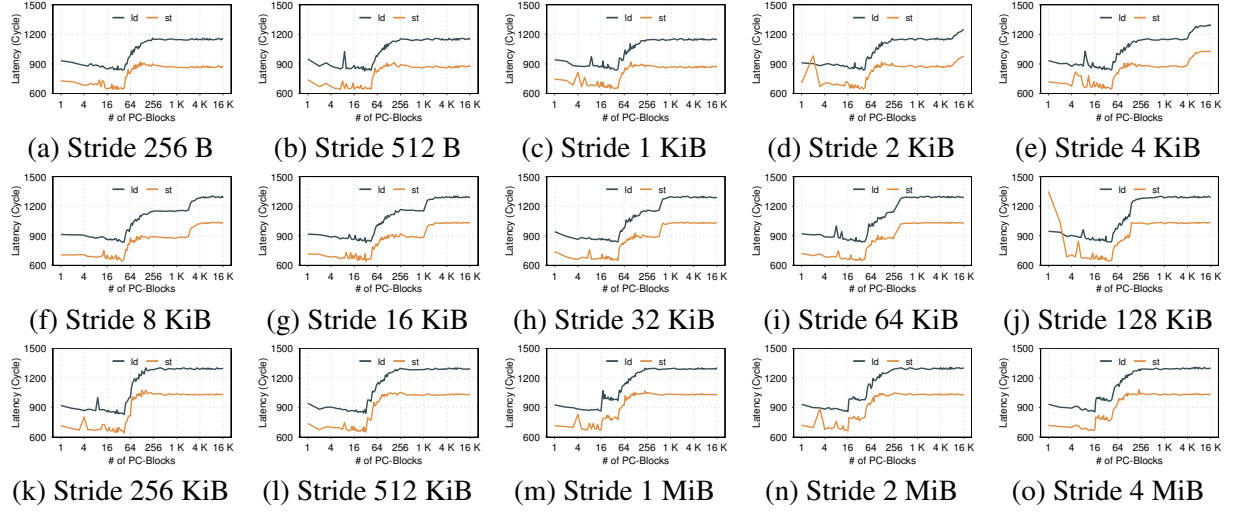


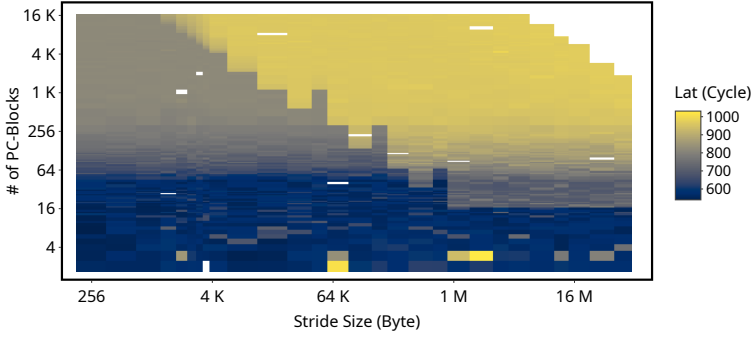
Figure 4.17: Pointer chasing latency under various stride sizes.

Figure 4.4.

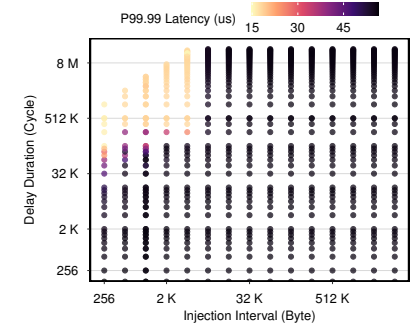
4.A.2 Additional Results on Server B

In this section, we provide reverse engineering on an additional server machine (Server B from Table 4.1) which equips Optane DIMMs of different sizes and firmware versions compared to Server A.

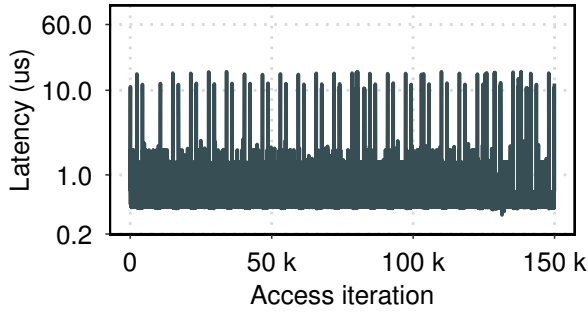
Figure 4.18 shows the NVLeak reverse engineering results. Following the analysis in Section 4.3, we conclude that Server B’s Optane DIMMs have similar architecture design and performance characteristics as in Server A: Server B Optane DIMM has two levels inclusive NVCaches (Figure 4.18a), where L1 is 16 KiB fully associative cache with 256 B blocks, and L2 is a 16-way set associative cache with 4 KiB blocks. This Optane DIMM also has a long latency effect triggered by repeated writes (Figure 4.18c), and this latency is timing-sensitive (Figure 4.18b and Figure 4.18d). This Optane DIMM’s data-migration trigger conditions (Figure 4.18e) under 256 B region have a median of 2,802 write iterations, with 50% of them falling in the $[2782 : 2814]$ range, which is much more concentrated than in Server A (Figure 4.7).



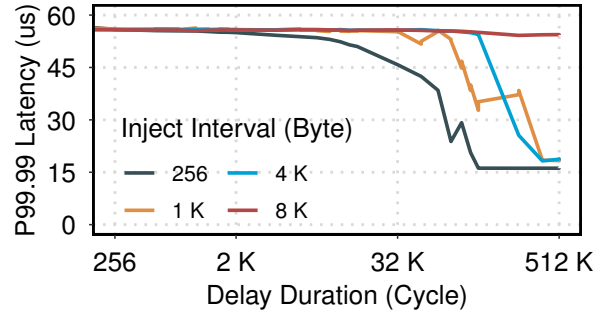
(a) Pointer chasing strided.



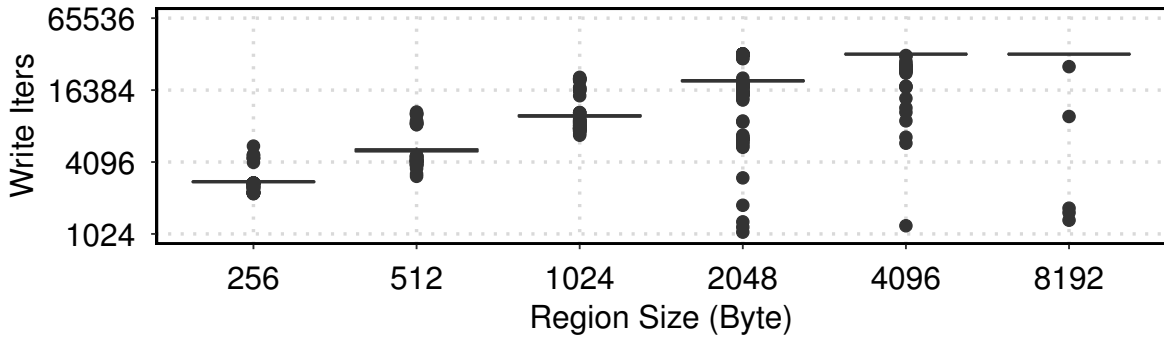
(b) Overwrite 99.99 percentile (P99.99) latencies.



(c) Overwrite per-iter latency.



(d) Overwrite P99.99 latencies.



(e) Data migration trigger distribution.

Figure 4.18: Reverse engineering results on Server B (Table 4.1).

Chapter 5

Programming for the Future Heterogeneous Systems

5.1 Introduction

Modern workloads heavily rely on heterogeneous systems. Such systems integrate multiple types of specialized computing and memory devices toward high-performance computing. GPU and high-bandwidth memory (HBM) support high-performance machine learning systems, SmartNICs accelerate network processing in cloud computing, and processing-in-memory (PIM) improves computing efficiency in memory-intensive workloads. With the advancement of device interconnection technologies such as PCIe, NVLink, and Compute Express Link (CXL), more types of accelerators and memory will be integrated into heterogeneous systems to support irregular performance requirements of workloads such as large language models, graph processing, and serverless.

Software support remains one critical limitation of heterogeneous systems' scalability. Historically, accelerators have established unique protocols to exchange data and manage tasks, relying on specialized operating system drivers and programming libraries. Thus, to use het-

```

#include <device_library>
int main() {
    int r[N], s[N];    // Results and inputs
    for (int i = 0; i < TOTAL_DEVICES; i++) {
        device_func(&(r[i]), &(s[i]));
    }
}
void device_func(int *result, int *input){
    device_copy(input, device_memory); // Explicit memory copy through PCIe
    device_calculate();                // Call external library and compiler
    device_wait_result();              // Device-specific synchronization
    device_copy(device_memory, result);
}

```

(a) Heterogeneous code example.

```

int main() {
    std::thread t[N]; // N threads
    int r[N], s[N];   // Results and inputs
    for (int i = 0; i < N; i++){
        t[i] = std::thread(thread_func, std::ref(r[i]), std::ref(s[i]));
        t[i].join(); // Native synchronization support
    }
}
void thread_func(int &result, const int &input) {
    result = calculate(input); // C++ native code, transparent memory sharing
}

```

(b) Multithreading code example.

Figure 5.1: Heterogeneous programming is generally more complex than multi-threading programming due to the need of using external libraries, system drivers, and compiler toolchains.

erogeneous systems, applications must employ specialized code for accelerators while ensuring compatibility between them, and the operating system must manage different drivers. Such programming complexity drives the development of programming supports for heterogeneous computing. However, they either introduce unconventional programming models [214–216] requiring rewriting applications, or targeting domain specific use cases such as machine learning [217–219] making them hard to generalize.

Programming heterogeneous systems is a special type of parallel programming where CPUs send data and code to accelerators, signal them to start tasks, and wait for results. In this process, CPUs and accelerators work in parallel and communicate through PCIe or similar

interconnection buses. Such parallel computing in heterogeneous systems (Figure 5.1a) is conceptually similar to multi-CPU systems (Figure 5.1b), where multiple CPU cores run different program threads and communicate through core-to-core buses. However, multi-CPU programming has a much more intuitive programming model where the parallel tasks are defined as native functions and spawned by `fork` or `clone` system calls which are natively supported by commonly used languages such as C, C++, Rust. Such multi-threading models are enabled by (1) the unified CPU architecture that enables the code to be implemented in a unified language, and (2) the cache-coherent connections between CPUs that enables different threads to share memory transparently without relying on explicit calls to libraries. Historically, these two architectural properties are not available in heterogeneous systems, because different accelerators implement their unique architectures, and they are typically interconnected through non-cache-coherent peripheral buses such as PCIe. With the recent advancement of Compute Express Link (CXL) as a general cache-coherent interconnection, we see an opportunity to generalize heterogeneous computing.

We present CodeFlow, a unified programming model for heterogeneous computing. CodeFlow leverages WebAssembly System Interface (WASI) [220] to build a language runtime system, which serves as an intermediate layer between higher-level language and lower-level heterogeneous architectures. This runtime system allows programs to be implemented in a single language and compiled to a single runtime representation, without the need to explicitly implement code for different architectures and compile with multiple toolchains. At low level, CodeFlow leverages CXL to enable coherent memory sharing among heterogeneous accelerators, thus minimizing explicit library calls for cross-devices data movements.

We demonstrate that CodeFlow can significantly simplify heterogeneous programming. The heterogeneous code can now be written as plain multi-threading code, which leverages the language’s native supports for standard multi-threading mechanisms, such as synchronization and memory sharing. The runtime schedules different threads to run on different accelerators, handles

memory sharing through CXL, and just-in-time compiles code for accelerator architectures. CodeFlow also allows conventional multi-threading programs—designed for multi-CPU systems only—to leverage heterogeneous systems by re-compiling with CodeFlow toolchain.

5.2 Background and Motivation

Heterogeneous computing is similar to multi-threading in many ways. We observe a few fundamental limitations in existing heterogeneous computing that hold back the system scalability and point out potential solutions with the recent advancements in programming and architecture techniques.

5.2.1 Multi-Threading Programming

Multi-threading is a programming approach that allows a process to be divided into multiple threads, with each thread capable of executing concurrently. This model enables applications to perform multiple operations simultaneously, leveraging the computational power of modern multi-CPU systems more effectively. In such systems, cache coherence plays a critical role at architecture level, ensuring the consistency of data cross the caches of a multi-core or multi-processor systems. When multiple threads running on different CPU cores access shared data, cache coherence mechanisms ensure that all modifications are propagated across all caches. This guarantees that any thread accessing a shared data will see the most recent update, irrespective of which core the updating thread runs on.

Cache coherence fundamentally simplifies the multi-threading programming model while ensuring correctness and improving performance. Cache coherence allows developers to write multithreaded applications without the cumbersome need to manually synchronize data between threads or processors. This abstraction reduces the complexity of coding for concurrency and helps applications scale efficiently by adding more CPU cores.

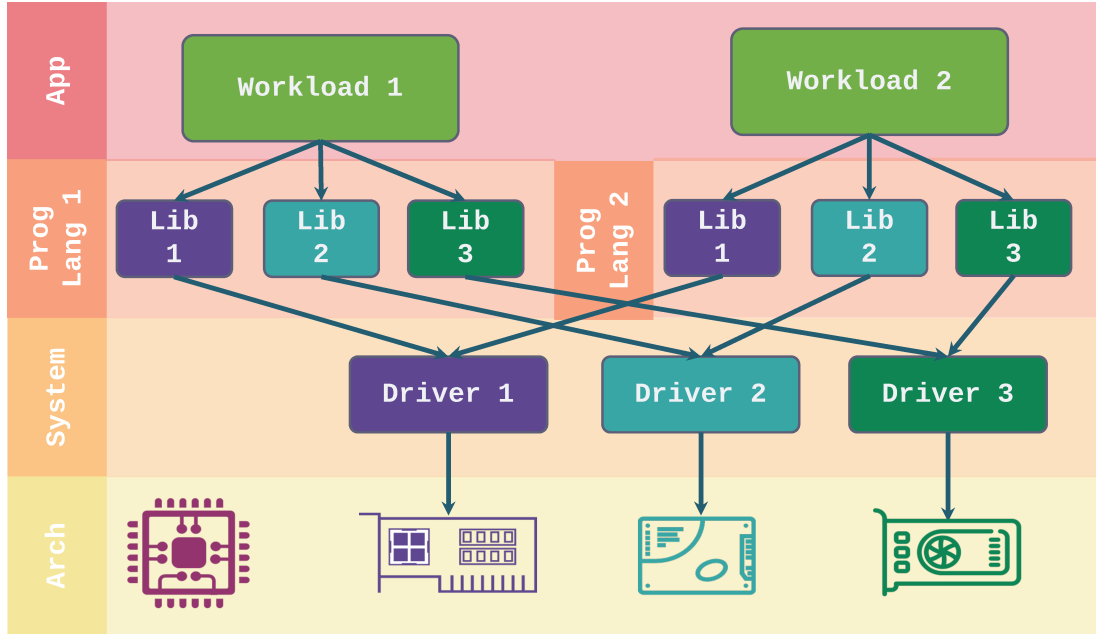


Figure 5.2: Heterogeneous system stack with conventional programming techniques, where workloads have to use multiple libraries and compilers to utilize underlying heterogeneous architectures. Such programming complexity limits the system’s scalability of adding new architectures.

5.2.2 Heterogeneous Systems

Heterogeneous systems integrate diverse processors and accelerators, such as CPUs, GPUs, FPGAs, and DSPs, to leverage their specialized computational capabilities for enhanced performance and efficiency across various applications. This integration, while beneficial for addressing the irregular demands of modern computing tasks such as machine learning and graph processing, introduces significant programming challenges. As shown in Figure 5.2, developers are imposed to utilize multiple programming models and languages, alongside the use of specialized libraries designated to each architecture. Although many libraries and languages emerges to serve as an intermediate layer between developers and heterogeneous systems, aiming to provide easy-to-use programming interfaces, they either introduce new programming models [214, 215] or target scoped use cases, requiring application rewriting and raising the complexity of adopting new architectures.

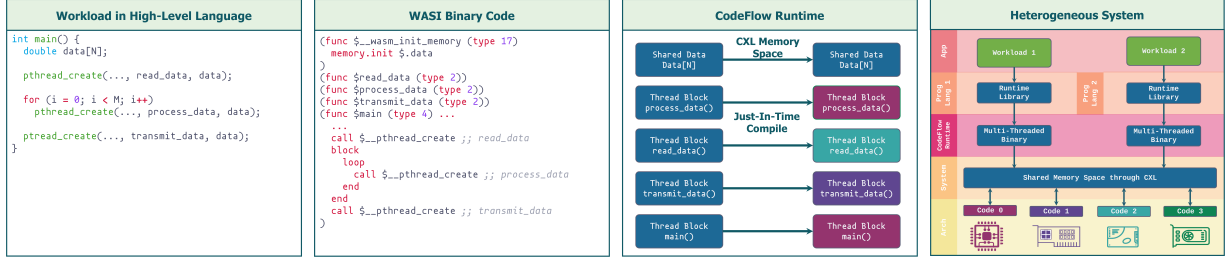


Figure 5.3: Heterogeneous system programming flow with CodeFlow.

Such programming complexity in heterogeneous systems is mainly caused by (1) the absence of cache coherence and (2) the diversity of architectures. The lack of a unified cache coherence mechanism leads to data inconsistencies, requiring explicit data management and synchronization efforts. Moreover, the need to navigate and optimize for different execution models and memory systems adds to the development burden. These two challenges underscore the need for unified programming models and toolchains that can simplify data management and exploit the full potential of heterogeneous systems. In following sections, we describe two key techniques that enable us to address the cache coherence and architecture heterogeneity challenges respectively.

5.2.3 Compute Express Link

Compute Express Link (CXL) [101] is an open industry-standard interconnection protocol, designed to facilitate high-speed communications between processors, accelerators (such as GPUs and SmartNICs), and memory devices in computing systems. A unique feature of CXL is its capability at hardware level to enable coherent memory access across heterogeneous architectures. This coherence ensures that various processors and accelerators can transparently share memory resources, as though they were part of a unified system. By leveraging CXL, we can simplify the memory access operations in heterogeneous programming models.

5.2.4 WebAssembly System Interface

WebAssembly is a language runtime system initially designed to accelerate web application. It is a hardware-agnostic binary instruction format designed as a portable compilation target for high-level languages like C, C++, and Rust. It enables code to run with near-native performance across various platforms, initially within web browsers to improve the performance of web applications. Recently, WebAssembly System Interface (WASI) has been proposed to provide WebAssembly applications with a standardized system interface, allowing them to interact with the operating system's underlying capabilities, such as file systems, network sockets, and other system resources. With WASI, high-level code can now compile to WebAssembly, run in WASI environment, access system resources through standardized interface, and execute on any supported architecture, all without rewriting or recompiling the code. This portability makes WASI a perfect candidate for us to design an abstraction layer between high-level code and low-level heterogeneous architectures.

5.3 Design

This section introduces the design of CodeFlow framework, from compilation, runtime management, to CXL-based shared memory and heterogeneous system mapping.

5.3.1 Compile High-Level Code

As shown in Figure 5.3, the workload code is written in high-level language as a multithreading program. The workload is divided as a few sub-tasks each running as a standalone thread, sharing the access to the shared memory. In this example, each thread is responsible for a specific job, from reading the data from the file, to processing the data in parallel, and finally transmitting the data. All threads are implemented in native language (C in this example) without using any hardware-specific library or compiler. The program invokes the C standard libraries

provided by WASI [221], which implement native library functionalities while inherently interact with the WebAssembly runtime system instead of the bare metal operating system. The workload can call third-party libraries for specific tasks such as network I/O, and all libraries are compiled to WASI binary format with the workload.

WASI defines a hardware-agnostic assembly binary representation by extending the WebAssembly binary representation with capabilities to access operating system resources such as filesystem. The high level workload is compiled once as WASI binary, where each thread is represented by a WASI code block, as shown in Figure 5.3. Such WASI binary is then executed by CodeFlow runtime system which schedules threads in the heterogeneous system.

5.3.2 Runtime System

CodeFlow runtime system executes WASI multi-threading binaries in the CXL-interconnected heterogeneous system. On loading a WASI binary, CodeFlow analyzes each thread’s code and detects a processor or accelerator to execute the thread. E.g., a file access thread will be assigned to CPU or an in-storage processor, a parallel data processing thread will be scheduled to use GPU, and network traffic is scheduled to use SmartNICs. Developers can annotate the workload source code to assist CodeFlow’s device detection process. Once devices are enumerated, CodeFlow performs just-in-time (JIT) compilation that compiles the WASI binary to the device’s code and executes it. This code generation is built on-top-of conventional heterogeneous computing libraries, including device libraries such as CUDA, and frameworks such as PyTorch [217] and SYCL [214]. The workload’s shared memory is placed in the CXL memory space shared by processors and accelerators to provide coherent access across threads. To minimize the JIT overhead, the developer can choose to perform ahead-of-time (AOT) compilation that compiles the WASI binary to device code ahead-of-time, thus reducing the JIT time consumption at runtime.

Table 5.1: System for Evaluation

CPU	Intel SapphireRapids 4416+ 2 Sockets, 20 cores per socket
Memory	256 GiB DDR5
CXL	Intel Agilex-7 FPGA CXL hard IP type 2 & type 3 64 GiB DDR4 SODIMM
OS	Ubuntu 22.04, Linux 6.7

5.3.3 CXL Interconnection

The heterogeneous system is interconnected through CXL, where processors and accelerators share a coherent memory address space, allowing all connected devices to coherently access such memory space. Following CXL’s specification, an accelerator with device memory is defined as a CXL Type 2 device, and a pure memory device is defined as a CXL Type 3 device. The heterogeneous system could be composed by a combination of Type 2 and 3 devices and their on-device memory has divergent performance characteristics. CodeFlow places shared data in CXL memory space for coherent accesses from different devices. To maximize the workload performance, CodeFlow plans the data placement in device memory (e.g., GPU memory) as part of its profiling-guided optimizations at runtime, and such data can be migrated between devices to maximize temporal access performance [222].

Table 5.2: CXL Performance

Component	Performance	
	Latency (ns)	Bandwidth (GiB/s)
Local DDR5	108.2	105.0
Remote DDR5	171.5	59.1
Local CXL	371.2	17.4
Remote CXL	538.0	9.0

5.4 Evaluation

We build an experimental system with CXL-capable CPUs and FPGAs. As shown in Table 5.1, we use a two-socket Intel CPU machine. We configured the BIOS to enable CXL connections between CPU and devices on PCIe bus. We implemented CXL devices in Intel Agilex-7 FPGAs by invoking Intel’s CXL hard IP on the FPGA, connecting these IPs with on-board memory, and presenting the FPGA as a CXL device to the system. We modified the Linux kernel’s CXL driver to recognize our CXL devices due to the incompatibilities between our CXL 1.1/2.0 IPs and the Linux kernel’s implementation. We implemented CodeFlow on top of Wasmtime [223] to be compatible with WASI SDK 21 with thread extension, and evaluated the system’s performance.

5.4.1 CXL Performance

We built a memory performance microbenchmark suite based on prior research on heterogeneous memory systems [2, 5], and use it to profile the CXL device performance. This benchmark suite uses strided memory access with pointer-chasing pattern to measure memory access latency. Table 5.2 shows our evaluation of latency and bandwidth of CXL compared to CPU system memory. The *Local* represents performance between a local NUMA CPU and a local

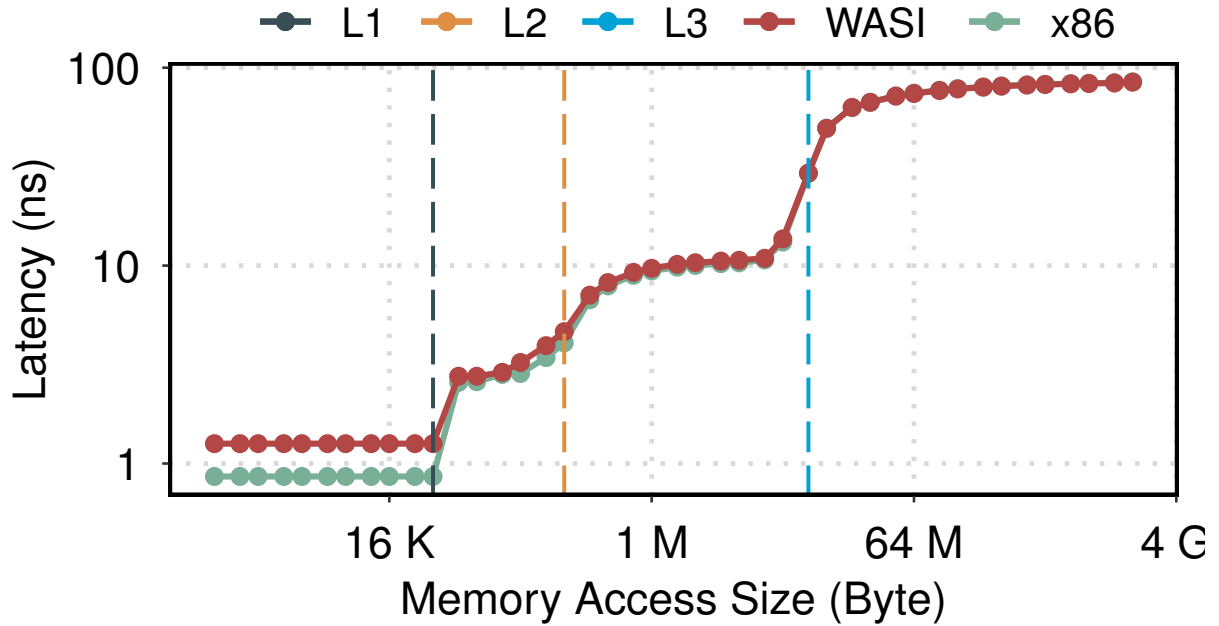


Figure 5.4: CodeFlow runtime performance compared to running native x86 code, using pointer-chasing read.

NUMA/CXL memory and *Remote* represents performance between remote NUMA nodes. We find that local CXL latency is much lower than remote CXL latency, while being higher than CPU system memory. The CXL memory bandwidth is lower than remote CPU memory bandwidth. This is because our Intel FPGA has DDR4 on-board memory which is slower than DDR5, and FPGA CXL controller adds up latencies due to FPGA frequency limited to 475MHz. We expect production CXL memory to have higher performance.

5.4.2 WASI Performance

We adapt our microbenchmark suite to run in WebAssembly and profile the CodeFlow runtime system performance. Figure 5.4 shows our evaluation of memory access latency given different memory access sizes. We use a pointer chasing memory access pattern to minimize the system noises (from cache prefetchers and compiler optimizations) in latency measurements. We observe that CodeFlow has the same latency compared to x86 except a slightly higher latency

when memory region is with CPU L1 cache size. This shows that CodeFlow introduces very minimal memory footprint overhead when accessing memory.

5.5 Future Directions

We propose a new programming model in heterogeneous systems allowing workload to be written in multi-threading programming model which is widely used in existing software. To support this programming model, we present our initial effort on implementing a language runtime system, CodeFlow, based on the WebAssembly System Interface. CodeFlow abstracts the underlying architecture and generates code at runtime, relying on just-in-time compilations in WASI. Looking forward, we envision this programming model is beneficial for future workloads to easily adopt heterogeneous systems. We are working on extending CodeFlow’s compiler and code generation system to support more types of accelerators. At architecture level, we are implementing CXL accelerators on CXL FPGA to support high-level workloads’ requirements, and converting conventional workloads to utilize heterogeneous systems through CodeFlow.

5.6 Conclusion

In this paper, we present CodeFlow, a unified programming model for heterogeneous programming. CodeFlow employs the emerging CXL interconnection to provide coherent data exchange between heterogeneous devices, and leverages WASI’s portable binary format as an intermediate representation between high-level workload and the underlying heterogeneous architecture. With CodeFlow, application developers can implement workloads as multithreading programs using native language supports, compile it once, and let CodeFlow to execute these threads on heterogeneous architectures. This programming model decouples the complexity in heterogeneous programming, allowing programmers to focus on designing and optimizing

high-level logics, leaving device-specific implementations to system developers, and leveraging a multithreading programming model as a unified intermediate layer. We envision that with CodeFlow, future workloads can be ported to various heterogeneous systems without rewriting or recompiling code, thus incentivizing the adoption of heterogeneous computing.

5.7 Acknowledgements

Chapter 5, in part, is currently being prepared for submission for publication of the material. Zixuan Wang, Jishen Zhao. “Fork is All You Need in Heterogeneous Systems”. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion

In this dissertation, we have explored the challenges in current and future heterogeneous systems and presented a full-stack approach to address issues across the computing stack. We took a bottom-up path: This dissertation started from the *architecture* level, discovered scalability challenges, and moved up to the system layer for solutions. We then attacked and secured the *architecture* and *system*. And finally, we proposed new *programming* models for future heterogeneous systems.

To recapitulate, we start with LENS and VANS to characterize and model the performance of emerging memory *architecture*. We observed that new memory architectures introduce scalability issues due to their requirement to use new programming interfaces, which led to COARSE, our exploration at *system* level to seamlessly integrate heterogeneous memory architecture into machine learning training systems. During the architecture- and system-level research, we reverse-engineered many memory architecture properties that are not publicly documented and are vulnerable to architectural attacks. We further studied the *security* at the architecture and systems level, which led to our work NVLEAK, where we presented a software framework for reverse engineering and attacking heterogeneous memory architecture. Throughout these prior research at lower levels, we found that current heterogeneous systems encounter *programming*

challenges due to the lack of a unified programming interface for various accelerators and memory architectures. We then proposed a new programming model to rethink heterogeneous programming as a multi-threading model, allowing a simplified and well-established programming model to be used in heterogeneous systems.

Bibliography

- [1] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module”. In: *arXiv* (2019).
- [2] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. “Characterizing and Modeling Non-Volatile Memory Systems”. In: *MICRO*. 2020.
- [3] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. “Characterizing and Modeling Nonvolatile Memory Systems”. In: *IEEE Micro Top Picks from 2021 Computer Architecture Conferences* (2021).
- [4] Zixuan Wang, Joonseop Sim, Euicheol Lim, and Jishen Zhao. “Enabling Efficient Large-Scale Deep Learning Training with Cache Coherent Disaggregated Memory Systems”. In: *HPCA*. 2022.
- [5] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. “NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems”. In: *USENIX Security*. 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/wangzixuan>.
- [6] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. “Architecting Phase Change Memory As a Scalable DRAM Alternative”. In: *ISCA*. 2009.
- [7] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. A. Wolf, A. W. Ghosh, J. W. Lu, S. J. Poon, M. Stan, W. H. Butler, S. Gupta, C. K. A. Mewes, T. Mewes, and P. B. Visscher. “Advances and Future Prospects of Spin-Transfer Torque Random Access Memory”. In: *IEEE Transactions on Magnetics* (2010).
- [8] H. -. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. “Metal-Oxide RRAM”. In: *IEEE* (2012).
- [9] Kosuke Suzuki and Steven Swanson. *The Non-Volatile Memory Technology Database (NVMDB)*. Tech. rep. CS2015-1011. Department of Computer Science & Engineering, University of California, San Diego, 2015. URL: <http://nvmdb.ucsd.edu>.
- [10] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. “Evaluating STT-RAM as an energy-efficient main memory alternative”. In: *ISPASS*. 2013.

- [11] E. Vianello, O. Thomas, G. Molas, O. Turkyilmaz, N. Jovanović, D. Garbin, G. Palma, M. Alayan, C. Nguyen, J. Coignus, B. Giraud, T. Benoist, M. Reyboz, A. Toffoli, C. Charpin, F. Clermidy, and L. Perniola. “Resistive Memories for Ultra-Low-Power embedded computing design”. In: *IEDM*. 2014.
- [12] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. “A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology”. In: *ISCA*. 2009.
- [13] Intel. *Intel Optane DC Persistent Memory*. 2019. URL: <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory>.
- [14] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. “Evaluation of Intel 3D-Xpoint NVDIMM Technology for Memory-intensive Genomic Workloads”. In: *MEMSYS*. 2019.
- [15] V. Young, Z. A. Chishti, and M. K. Qureshi. “TicToc: Enabling Bandwidth-Efficient DRAM Caching for Both Hits and Misses in Hybrid Memory Systems”. In: *ICCD*. 2019.
- [16] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. “System Evaluation of the Intel Optane Byte-Addressable NVM”. In: *MEMSYS*. 2019.
- [17] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. “An Early Evaluation of Intel’s Optane DC Persistent Memory Module and Its Impact on High-Performance Scientific Applications”. In: *SC*. 2019.
- [18] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. “Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules”. In: *MEMSYS*. 2019.
- [19] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. “System Software for Persistent Memory”. In: *EuroSys*. 2014.
- [20] Intel. *Intel® 64 and IA-32 Architectures Software Developer’s Manuals*. 2019.
- [21] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. “SplitFS: Reducing Software Overhead in File Systems for Persistent Memory”. In: *SOSP*. 2019.
- [22] Youmin Chen, Youyou Lu, Bohong Zhu, and Jiwu Shu. “Kernel/User-level Collaborative Persistent Memory File System with Efficiency and Protection”. In: *arXiv* (2019).
- [23] Jian Xu and Steven Swanson. “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories”. In: *FAST*. 2016.
- [24] Lu Zhang and Steven Swanson. “Pangolin: A Fault-Tolerant Persistent Memory Programming Library”. In: *ATC*. 2019.
- [25] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks”. In: *ASPLOS*. 2019.

- [26] Jian Yang, Joseph Izraelevitz, and Steven Swanson. “Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks”. In: *FAST*. 2019.
- [27] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. “Kiln: Closing the performance gap between systems with and without persistence support”. In: *MICRO*. 2013.
- [28] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. “ThyNVM: Enabling software-transparent crash consistency in persistent memory systems”. In: *MICRO*. 2015.
- [29] Intel. *2nd Generation Intel® Xeon® Scalable Processors with Intel® C620 Series Chipsets (Purley Refresh)*. 2019.
- [30] M. A. Ogleari, E. L. Miller, and J. Zhao. “Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems”. In: *HPCA*. 2018.
- [31] Haris Volos, Andres Jaan Tack, and Michael M. Swift. “Mnemosyne: Lightweight Persistent Memory”. In: *ASPLOS*. 2011.
- [32] Andy M Rudoff. *Deprecating the PCOMMIT Instruction*. 2016. URL: <https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction>.
- [33] Micron Technology, Inc. *3D XPoint Technology*. 2018. URL: <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>.
- [34] Intel. *Intel Memory Latency Checker*. 2019.
- [35] Brendan Gregg. *Linux Perf Examples*. 2019. URL: <http://www.brendangregg.com/perf>.
- [36] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security*. 2016.
- [37] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. “Quartz: A Lightweight Performance Emulator for Persistent Memory Software”. In: *Middleware*. 2015.
- [38] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. “NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories”. In: *ASPLOS*. 2011.
- [39] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. “DRAMSim2: A Cycle Accurate Memory System Simulator”. In: *CAL* (2011).
- [40] Yoongu Kim, Weikun Yang, and Onur Mutlu. “Ramulator: A Fast and Extensible DRAM Simulator”. In: *CAL* (2016).
- [41] Intel. *A utility for configuring and managing Intel Optane DC persistent memory modules*. 2019. URL: <https://github.com/intel/ipmctl>.
- [42] Intel. *Utility library for managing the libnvdimm (non-volatile memory device) sub-system in the Linux kernel*. 2019. URL: <https://github.com/pmem/ndctl>.

- [43] Vish Viswanathan. *Disclosure of Hardware Prefetcher Control on Some Intel Processors*. 2014. URL: <https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors>.
- [44] Brian Beeler. *Intel Optane DC Persistent Memory Module (PMM)*. 2019. URL: <https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm>.
- [45] ServeTheHome. *Intel Optane DCPMM UTH DDR T Protocol*. 2019. URL: <https://www.servethehome.com/2nd-gen-intel-xeon-scalable-launch-cascade-lake-details-and-analysis/intel-optane-dcpmm-uth-ddr-t-protocol/>.
- [46] Patrick Kennedy. *A Close Look at Intel Optane DC Persistent Memory Modules*. 2019. URL: <https://www.servethehome.com/a-close-look-at-intel-optane-dc-persistent-memory-modules/>.
- [47] J. Zhao, O. Mutlu, and Y. Xie. “FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems”. In: *MICRO*. 2014.
- [48] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* (2011).
- [49] Micron Technology, Inc. *DDR4 SDRAM Verilog Model*. 2018.
- [50] Cadence Design Systems, Inc. *Simulation and Testbench Verification*. 2019.
- [51] Standard Performance Evaluation Corporation. *SPEC CPU 2006*. 2019. URL: <https://www.spec.org/cpu2006/>.
- [52] Standard Performance Evaluation Corporation. *SPEC CPU 2017*. 2019. URL: <https://www.spec.org/cpu2017/>.
- [53] Intel. *Intel Vtune Amplifier*. 2019.
- [54] WikiChip. *Cascade Lake - Microarchitectures - Intel*. 2020.
- [55] Dave Hansen. *Allow persistent memory to be used like normal RAM*. 2019. URL: <https://patchwork.kernel.org/cover/10829019/>.
- [56] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. “Amber: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources”. In: *MICRO*. 2018.
- [57] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavayan Ziahari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. “MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization”. In: *ISCA*. 2019.

- [58] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, Dejan Milojicic, and Vanish Talwar. “Using Active NVRAM for I/O Staging”. In: *PDAC*. 2011.
- [59] Intel. *Pmem-Redis: A version of Redis that uses persistent memory*. 2019. URL: <https://github.com/pmem/pmem-redis>.
- [60] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking Cloud Serving Systems with YCSB”. In: *SoCC*. 2010.
- [61] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. 2015.
- [62] Brad Fitzpatrick. “Distributed Caching with Memcached”. In: *Linux Journal* (2004).
- [63] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. “Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation”. In: *ICCD*. 2016.
- [64] Jens Axboe. *Flexible I/O Tester*. 2019. URL: <https://github.com/axboe/fio>.
- [65] Intel. *Persistent Memory Development Kit*. 2019. URL: <https://pmem.io/>.
- [66] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. “An Analysis of Persistent Memory Use with WHISPER”. In: *ASPLOS*. 2017.
- [67] Scott Lloyd and Maya Gokhale. “Evaluating the Feasibility of Storage Class Memory as Main Memory”. In: *MEMSYS*. 2016.
- [68] M. Poremba, T. Zhang, and Y. Xie. “NVMain 2.0: A User-Friendly Memory Simulator to Model Non-Volatile Memory Systems”. In: *CAL* (2015).
- [69] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. “FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory”. In: *ASPLOS*. 2020.
- [70] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory”. In: *FAST*. 2020.
- [71] Swapnil Haria, Mark D. Hill, and Michael M. Swift. “MOD: Minimally Ordered Durable Datastructures for Persistent Memory”. In: *ASPLOS*. 2020.
- [72] Amir Roth and Gurindar S. Sohi. “Effective Jump-Pointer Prefetching for Linked Data Structures”. In: *ISCA*. 1999.
- [73] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. “Prefetched Address Translation”. In: *MICRO*. 2019.
- [74] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539.
- [75] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”. In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 82–97. ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2205597.

- [76] Xuedong Huang, James Baker, and Raj Reddy. “A Historical Perspective of Speech Recognition”. In: *Communications of the ACM* 57.1 (Jan. 2014), pp. 94–103. ISSN: 0001-0782. DOI: 10.1145/2500887.
- [77] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. “Google’s Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 339–351. DOI: 10.1162/tac1\%0090\%00800065.
- [78] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. “Deep, Big, Simple Neural Nets for Handwritten Digit Recognition”. In: *Neural Computation* 22.12 (Dec. 2010), pp. 3207–3220. ISSN: 0899-7667. DOI: 10.1162/NECO_a_00052.
- [79] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'auelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. “Large Scale Distributed Deep Networks”. In: *Advances in Neural Information Processing Systems* 25. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1223–1231.
- [80] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. “Deep Learning Scaling Is Predictable, Empirically”. In: *arXiv:1712.00409 [cs, stat]* (Dec. 2017). arXiv: 1712.00409 [cs, stat].
- [81] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. “Project Adam: Building an Efficient and Scalable Deep Learning Training System”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. USA: USENIX Association, Oct. 2014, pp. 571–582. ISBN: 978-1-931971-16-4.
- [82] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. “Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059.
- [83] NVIDIA. *NVIDIA DGX-1 with Tesla V100 System Architecture White paper*. URL: <https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf>. (accessed: 08.19.2020).
- [84] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz,

- Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *ACM SIGARCH Computer Architecture News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246.
- [85] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. “PipeDream: Generalized Pipeline Parallelism for DNN Training”. In: *SOSP*. 2019.
 - [86] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. “Blink: Fast and Generic Collectives for Distributed ML”. In: *arXiv:1910.04940 [cs]* (Oct. 2019). arXiv: 1910.04940 [cs].
 - [87] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. “GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism”. In: *arXiv:1811.06965 [cs]* (July 2019). arXiv: 1811.06965 [cs].
 - [88] NVIDIA. *NVIDIA A100 TENSOR CORE GPU*. URL: <https://www.nvidia.com/en-us/data-center/a100/>. (accessed: 08.19.2020).
 - [89] Google. *Cloud TPU*. URL: <https://cloud.google.com/tpu>. (accessed: 08.19.2020).
 - [90] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. “Large Scale Distributed Deep Networks”. In: *NIPS*. 2012, pp. 1223–1231.
 - [91] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. “A Hierarchical Model for Device Placement”. In: *International Conference on Learning Representations*. 2018. URL: <https://openreview.net/forum?id=Hkc-TeZ0W>.
 - [92] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. “HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 307–321. ISBN: 978-1-939133-14-4. URL: <https://www.usenix.org/conference/atc20/presentation/park>.

- [93] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. “Device Placement Optimization with Reinforcement Learning”. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML’17. Sydney, NSW, Australia: JMLR.org, Aug. 2017, pp. 2430–2439.
- [94] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. “Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems”. In: *arXiv:2003.05622 [cs, stat]* (Mar. 2020). arXiv: 2003.05622 [cs, stat].
- [95] PCI-SIG. *PCI Express Base Specification Revision 3.1a*. URL: <https://pcisig.com/specifications>. (accessed: 06.24.2020).
- [96] NVIDIA. *NVLINK AND NVSWITCH*. URL: <https://www.nvidia.com/en-us/data-center/nvlink/>. (accessed: 06.24.2020).
- [97] NVIDIA. *NVIDIA Collective Communications Library (NCCL)*. URL: <https://developer.nvidia.com/nccl>. (accessed: 06.24.2020).
- [98] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *CoRR* abs/1802.05799 (2018). arXiv: 1802.05799 [cs.LG]. URL: <http://arxiv.org/abs/1802.05799>.
- [99] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. “Hop: Heterogeneity-Aware Decentralized Training”. In: *ASPLOS*. 2019.
- [100] Gen-Z Consortium. *The Gen-Z Consortium*. URL: <https://genzconsortium.org/>. (accessed: 06.24.2020).
- [101] CXL Consortium. *CXL Consortium releases Compute Express Link 3.0 specification*. URL: <https://www.computeexpresslink.org/pressroom>.
- [102] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.
- [103] Zhihao Jia, Matei Zaharia, and Alex Aiken. “Beyond Data and Model Parallelism for Deep Neural Networks”. In: *arXiv:1807.05358 [cs]* (July 2018). arXiv: 1807.05358 [cs].
- [104] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *ArXiv:1810.04805* (2018).
- [105] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. “Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems”. In: *arXiv preprint arXiv:2003.05622* (2020).

- [106] Misha Smelyanskiy. “Zion: Facebook next-generation large memory training platform”. In: *Hot Chips*. 2019, pp. 1–2.
- [107] TensorFlow. *TensorFlow benchmarks*. URL: <https://github.com/tensorflow/benchmarks>. (accessed: 06.24.2020).
- [108] NVIDIA. *NVIDIA Deep Learning Examples for Tensor Cores*. URL: <https://github.com/NVIDIA/DeepLearningExamples>. (accessed: 06.24.2020).
- [109] Amazon Web Services. *Amazon EC2 Instance Types*. URL: <https://aws.amazon.com/ec2/instance-types/>. (accessed: 06.24.2020).
- [110] NVIDIA. *Profiler :: CUDA Toolkit Documentation*. URL: <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>. (accessed: 06.24.2020).
- [111] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. “Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 401–416. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378499.
- [112] CCIX Consortium. *CCIX*. URL: <https://www.ccixconsortium.com/>. (accessed: 06.24.2020).
- [113] PCI-SIG. *PCI Express Base Specification Revision 4.0, Version 1.0*. URL: <https://pcisig.com/specifications>. (accessed: 07.24.2021).
- [114] PCI-SIG. *PCI Express Base Specification Revision 5.0, Version 1.0*. URL: <https://pcisig.com/specifications>. (accessed: 07.24.2021).
- [115] NVIDIA. *GPUDirect*. URL: <https://developer.nvidia.com/gpudirect>. (accessed: 08.19.2020).
- [116] UC San Diego. *San Diego Supercomputer Center*. URL: <https://www.sdsc.edu/>.
- [117] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. “Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (Jan. 2020), pp. 94–110. ISSN: 1558-2183. DOI: 10.1109/TPDS.2019.2928289.
- [118] Cirrascale. *Cirrascale SR3514: Unexpected performance inequality. Technical brief M901A-092014*. 2018.
- [119] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. “TensorFlow: A System for Large-Scale Machine Learning”. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI’16. USA: USENIX Association, Nov. 2016, pp. 265–283. ISBN: 978-1-931971-33-1.

- [120] Linux Programmer’s Manual. *mmap(2) — Linux manual page*. URL: <https://man7.org/linux/man-pages/man2/mmap.2.html>. (accessed: 06.24.2020).
- [121] Xilinx KCU1500. *BittWare 250-SoC*. URL: https://www.xilinx.com/support/documentation/boards_and_kits/kcu1500/ug1260-kcu1500-data-center.pdf. (accessed: 08.14.2020).
- [122] 250-SoC. *BittWare 250-SoC*. URL: <https://www.bittware.com/fpga/250-soc/>. (accessed: 08.14.2020).
- [123] N Ledentsov Jr, M Agustin, L Chorchos, NN Ledentsov, and JP Turkiewicz. “25.78 Gbit/s data transmission over 2 km multi-mode-fibre with 850 and 910 nm single-mode VCSELs and a commercial quad small form-factor pluggable transceiver”. In: *Electronics Letters* 54.12 (2018), pp. 774–775.
- [124] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Identity Mappings in Deep Residual Networks”. en. In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 630–645. ISBN: 978-3-319-46493-0. DOI: 10.1007/978-3-319-46493-0_38.
- [125] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “ImageNet: A Large-Scale Hierarchical Image Database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.
- [126] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. “SQuAD: 100,000+ Questions for Machine Comprehension of Text”. en. In: (June 2016).
- [127] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4.
- [128] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. “GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server”. en. In: *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys ’16*. London, United Kingdom: ACM Press, 2016, pp. 1–16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901323.
- [129] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. “Parameter Hub: A Rack-Scale Parameter Server for Distributed Deep Neural Network Training”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 41–54. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267840.

- [130] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. “A Generic Communication Scheduler for Distributed DNN Training Acceleration”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, Oct. 2019, pp. 16–29. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359642.
- [131] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. “An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. May 2020, pp. 996–1009. DOI: 10.1109/ISCA45697.2020.00085.
- [132] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. “Disaggregated Memory for Expansion and Sharing in Blade Servers”. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. New York, NY, USA: Association for Computing Machinery, June 2009, pp. 267–278. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555789.
- [133] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. “System-Level Implications of Disaggregated Memory”. In: *IEEE International Symposium on High-Performance Comp Architecture*. Feb. 2012, pp. 1–12. DOI: 10.1109/HPCA.2012.6168955.
- [134] Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. “Multi-GPU System Design with Memory Networks”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 484–495. DOI: 10.1109/MICRO.2014.55.
- [135] Youngeun Kwon and Minsoo Rhu. “Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2018, pp. 148–161. DOI: 10.1109/MICRO.2018.00021.
- [136] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. “TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, Oct. 2019, pp. 740–753. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358284.
- [137] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. “Accelerating Distributed Reinforcement Learning with In-Switch Computing”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA ’19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 279–291. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322259.
- [138] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *CT-RSA*. 2006.

- [139] Dmitry Evtvushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. “Branch-Scope: A New Side-Channel Attack on Directional Branch Predictor”. In: *ASPLOS*. 2018.
- [140] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. “MemJam: A false dependency attack against constant-time crypto implementations”. In: *International Journal of Parallel Programming* (2019).
- [141] Tao Zhang, Kenneth Koltermann, and Dmitry Evtvushkin. “Exploring branch predictors for constructing transient execution trojans”. In: *ASPLOS*. 2020.
- [142] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. “Last-level cache side-channel attacks are practical”. In: *S&P*. 2015.
- [143] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “Make sure DSA signing exponentiations really are constant-time”. In: *CCS*. 2016.
- [144] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. “”Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel Can Go a Long Way”. In: *CHES*. 2014.
- [145] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. “Practical keystroke timing attacks in sandboxed javascript”. In: *European Symposium on Research in Computer Security*. 2017.
- [146] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. “Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite”. In: *USENIX Security*. 2021.
- [147] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre attacks: Exploiting speculative execution”. In: *S&P*. 2019.
- [148] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading kernel memory from user space”. In: *USENIX Security*. 2018.
- [149] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *USENIX Security*. 2018.
- [150] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-privilege-boundary data sampling”. In: *CCS*. 2019.
- [151] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. “Fallout: Leaking data on meltdown-resistant CPUs”. In: *CCS*. 2019.

- [152] Jose Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Chris Fletcher, and David Kohlbrenner. “Augury: Using data memory-dependent prefetchers to leak data at rest”. In: *S&P*. 2022.
- [153] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. “LVI: Hijacking transient execution through microarchitectural load value injection”. In: *S&P*. 2020.
- [154] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. “Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks”. In: *USENIX Security*. 2021.
- [155] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Taveri. “Port Contention for Fun and Profit”. In: *S&P*. 2019.
- [156] Daniel Townley and Dmitry Ponomarev. “SMT-COP: Defeating side-channel attacks on execution units in smt processors”. In: *PACT*. 2019.
- [157] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. “SecSMT: Securing SMT Processors against Contention-Based Covert Channels”. In: *USENIX Security*. 2022.
- [158] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. “Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World”. In: *S&P*. 2019.
- [159] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. “Catalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *HPCA*. 2016.
- [160] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. “ScatterCache: Thwarting Cache Attacks via Cache Set Randomization”. In: *USENIX Security*. 2019.
- [161] Moinuddin K Qureshi. “CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping”. In: *MICRO*. 2018.
- [162] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. “Systematic analysis of randomization-based protected cache architectures”. In: *S&P*. 2021.
- [163] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. “MeshUp: Stateless Cache Side-channel Attack on CPU Mesh”. In: *S&P*. 2022.
- [164] Johanna Sepúlveda, Mathieu Gross, Andreas Zankl, and Georg Sigl. “Beyond cache attacks: Exploiting the bus-based communication structure for powerful on-chip microarchitectural attacks”. In: *ACM Transactions on Embedded Computing Systems (TECS)* (2021).
- [165] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cross processor cache attacks”. In: *ASIA CCS*. 2016.
- [166] Yao Wang and G Edward Suh. “Efficient timing channel protection for on-chip networks”. In: *International Symposium on Networks-on-Chip*. 2012.

- [167] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. “Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical”. In: *USENIX Security*. 2021.
- [168] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting Dram Addressing for Cross-Cpu Attacks”. In: *USENIX Security*. 2016.
- [169] Jialiang Zhang, Nicholas Beckwith, and Jing Jane Li. “GORDON: Benchmarking Optane DC Persistent Memory Modules on FPGAs”. In: *FCCM*. 2021.
- [170] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. “Architecting Phase Change Memory As a Scalable DRAM Alternative”. In: *ISCA*. 2009.
- [171] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory”. In: *FAST*. 2020.
- [172] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. “Pump up the volume: Practical database reconstruction from volume leakage on range queries”. In: *CCS*. 2018.
- [173] Intel® Optane™ Persistent Memory. URL: <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>.
- [174] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. “Design Tradeoffs for SSD Performance”. In: *ATC*. 2008.
- [175] Colin Percival. “Cache Missing for Fun and Profit”. In: *In Proc. of BSDCan 2005*. 2005.
- [176] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.
- [177] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security*. 2014.
- [178] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security*. 2015.
- [179] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks”. In: *USENIX Security*. 2018.
- [180] Dmitry Evtvushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking Branch Predictors to Bypass ASLR”. In: *MICRO*. 2016.
- [181] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. “SMoTherSpectre: Exploiting Speculative Execution through Port Contention”. In: *CCS*. 2019.
- [182] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *CCS*. 2016.

- [183] Intel. *Intel 64 and IA-32 Architectures Software Developer Manual*.
- [184] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. “RAMBleed: Reading Bits in Memory Without Accessing Them”. In: *S&P*. 2020.
- [185] Linux Kernel Organization. *KVM-unit-tests*. URL: <https://www.linux-kvm.org/page/KVM-unit-tests>.
- [186] Linux Kernel Organization. *ext4 Data Structures and Algorithms*. URL: <https://www.kernel.org/doc/html/latest/filesystems/ext4>.
- [187] Linux Kernel Organization. *Direct Access for files*. URL: <https://www.kernel.org/doc/Documentation/filesystems/dax.txt>.
- [188] SQLite Consortium. *SQLite*. URL: <https://www.sqlite.org/index.html>.
- [189] Centers for Medicare & Medicaid Services. *NPPES Dataset*. URL: https://download.cms.gov/nppes/NPI%5C_Files.html.
- [190] N. S. Altman. “An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression”. In: *The American Statistician* (1992).
- [191] Intel. *Libpmemobj-cpp Examples*. URL: https://github.com/pmem/libpmemobj-cpp/tree/master/examples/map_cli.
- [192] Intel. *Libpmemobj-cpp: C++ bindings and containers for libpmemobj*. URL: <https://github.com/pmem/libpmemobj-cpp>.
- [193] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. “Page Cache Attacks”. In: *CCS*. 2019.
- [194] WolfSSL Inc. *WolfSSL Embedded SSL/TLS Library*. URL: <https://www.wolfssl.com/>.
- [195] Nadia Heninger and Hovav Shacham. “Reconstructing RSA private keys from random key bits”. In: *CRYPTO*. 2009.
- [196] Gabrielle De Micheli and Nadia Heninger. *Recovering cryptographic keys from partial information, by example*. 2020.
- [197] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. “Sliding Right into Disaster: Left-to-Right Sliding Windows Leak”. In: *CHES*. 2017.
- [198] Ashay Rane, Calvin Lin, and Mohit Tiwari. “Raccoon: Closing Digital Side-Channels through Obfuscated Execution”. In: *USENIX Security*. 2015.
- [199] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. “SecDCP: Secure dynamic cache partitioning for efficient timing channel protection”. In: *DAC*. 2016.
- [200] Gururaj Saileshwar and Moinuddin Qureshi. “MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design”. In: *USENIX Security*. 2021.

- [201] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. “HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments”. In: *USENIX Security*. 2020.
- [202] Zhenghong Wang and Ruby B. Lee. “New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks”. In: *ISCA*. 2007.
- [203] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *USENIX Security*. 2019.
- [204] Intel. *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family*.
- [205] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. “Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering”. In: *EuroSys*. 2022.
- [206] Sihang Liu, Suraj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, Khan, and Samira. “Side-Channel Attacks on Optane Persistent Memory”. In: *USENIX Security*. 2023.
- [207] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. “Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories”. In: *ISCA*. 2019.
- [208] Pengfei Zuo, Yu Hua, and Yuan Xie. “SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads”. In: *MICRO*. 2019.
- [209] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. “Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory”. In: *MICRO*. 2020.
- [210] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. “Temporal Exposure Reduction Protection for Persistent Memory”. In: *HPCA*. 2022.
- [211] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. “Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems”. In: *ISCA*. IEEE. 2021.
- [212] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. “Rendered insecure: Gpu side channel attacks are practical”. In: *CCS*. 2018.
- [213] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. “JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).
- [214] The Khronos Group. *SYCL Overview*. 2024. URL: <https://www.khronos.org/sycl/>.
- [215] The Khronos Group. *OpenCL: Open Standard for Parallel Programming of Heterogeneous Systems*. 2024. URL: <https://www.khronos.org/opencl/>.

- [216] Intel. *oneAPI: A New Era of Heterogeneous Computing*. 2024. URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html>.
- [217] The Linux Foundation. *PyTorch*. 2024. URL: <https://pytorch.org/>.
- [218] TornadoVM. *Java Acceleration with TornadoVM*. 2024. URL: <https://www.tornadovm.org/>.
- [219] OpenAI. *Introducing Triton: Open-source GPU programming for neural networks*. 2024. URL: <https://openai.com/research/triton>.
- [220] WebAssembly Community Group. *The WebAssembly System Interface*. 2024. URL: <https://wasi.dev/>.
- [221] WebAssembly Community Group. *WASI libc implementation for WebAssembly*. 2024. URL: <https://github.com/WebAssembly/wasi-libc>.
- [222] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. “TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. 2023.
- [223] Bytecode Alliance. *Wasmtime: A fast and secure runtime for WebAssembly*. 2024. URL: <https://github.com/bytecodealliance/wasmtime>.