

**MEMORY SYSTEM OPTIMIZATIONS FOR PARALLEL AND
BANDWIDTH-INTENSIVE WORKLOADS**

A Dissertation
Presented to
The Academic Faculty

By

Divya Kiran Kadiyala

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Engineering
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2025

© Divya Kiran Kadiyala 2025

**MEMORY SYSTEM OPTIMIZATIONS FOR PARALLEL AND
BANDWIDTH-INTENSIVE WORKLOADS**

Thesis committee:

Dr. Alexandros Daglis
School of Computer Science
Georgia Institute of Technology

Dr. Yingyan (Celine) Lin
School of Computer Science
Georgia Institute of Technology

Dr. Moinuddin K. Qureshi
School of Computer Science
Georgia Institute of Technology

Dr. Puneet Sharma
Networking and Distributed Systems Lab
Hewlett Packard Enterprise Labs

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date approved: November 17, 2025

उद्यमः साहसम् धैर्यं बुद्धिः शक्तिः पराक्रमः |
षडेते यत्र वर्तन्ते तत्र देवाः सहायकृत् ||
- सुभाषित रत्नाकर

udyamaḥ sāhasam dhairyaṃ buddhiḥ śaktiḥ parākramaḥ |
ṣaḍete yatra vartante tatra devāḥ sahāyakṛt ||
- subhāṣita ratnākara

*“Industriousness, courage, determination, wisdom, strength, and valor.
Wherever these six qualities exist, there the Gods are helpful.”*

To my parents, grandparents, family and friends

ACKNOWLEDGMENTS

Research is never a solitary endeavor. Whatever I have been able to accomplish in this dissertation stands firmly on the shoulders of giants—mentors, collaborators, friends, and family whose guidance, generosity, and patience have shaped my journey in countless ways. As a researcher, I have reached this stage only because so many people chose to invest their time, trust, and encouragement in me. This thesis is, in many respects, a collective achievement, and I am deeply indebted to each person and institution who has supported me along the way. I am also very grateful for the financial support provided by the National Science Foundation under Award SHF-2333049 and by a research gift from Samsung. I additionally acknowledge the use of OpenAI’s ChatGPT tool for editorial assistance in refining the wording and organization of this dissertation; all ideas, analyses, and conclusions are my own.

To my thesis committee—Prof. Tushar Krishna, Prof. Moinuddin Qureshi, Prof. Celine Lin, Dr. Puneet Sharma, and Prof. Alexandros Daglis—thank you for your insightful and constructive feedback, thoughtful questions, and steady encouragement. Your expertise shaped the direction of this work and clarified its contributions at many critical junctures.

To my thesis advisor, Prof. Alexandros Daglis—thank you for believing in me more than I often believed in myself, and for your unwavering support, guidance, and mentorship. Under your supervision, I learned how to think at a “big-picture” level while still caring deeply about the fine details. You took me as a rough stone and helped polish me into a diamond. Your constant pursuit of deeper insights, clearer reasoning, and stronger arguments helped train my critical thinking and reshape how I approached research. Your insistence on academic rigor steadily pushed me to improve and sharpen my ability to see beyond the noise and focus on what truly matters. I learned to lean into difficult problems, to accept criticism as a tool rather than a setback, and to see obstacles as chances to refine my thinking. Time and again, each of our discussions gave me a clearer sense of direction

and renewed confidence; when I felt stuck, you helped turn uncertainty into a concrete path forward. Because of you, I leave this Ph.D. not just with a dissertation but with the unshakable belief that I can tackle hard problems and see them through—a lesson I will carry for the rest of my life. Thank you so much for being my advisor, my mentor, and a truly wonderful person.

I owe special gratitude to my early academic mentors who first sparked my interest in research. To Dr. Satya Srinivas Babu Patcha, for your unwavering support, belief, and extraordinary kindness. Your energy and passion for research and excellence are contagious, and they ignited my own dream of becoming a researcher. Your outlook on innovation—finding practical, impactful solutions to real societal problems through technology—has been a lasting inspiration. I feel fortunate to have been your student during my undergraduate years. To Dr. Lawrence Clark, thank you for giving me the crucial opportunity and confidence to pursue research. I am deeply grateful for your trust, for involving me in research projects, and for providing the guidance and resources that helped shape the researcher in me and set me on a path toward greater goals.

To my fellow Númenóreans at the Network Architecture and Systems Lab (NArSyL)—Dr. Hamed Seyedroudbari, Marina Vemmou, Albert Cho, Prachatos Mitra, and Peidi Song—thank you for your constant support and kindness. I truly could not have asked for better colleagues and friends. More than the Ph.D. itself, it is your friendship and generosity that I will carry with me for the rest of my life. Thank you for being there, cheering me on, and believing in me. A special thanks to our “MICRO Queen,” Marina, for the extensive support with ZSim and for helping me ramp up with the tool. Your insightful, thoughtful feedback had a profound impact on improving the quality of my papers and presentations.

A very special thanks to Hamed, our “HPCA King.” You are truly a God-gifted brother to me. You stood by me through all the highs and lows of this arduous journey, supporting and cheering me in difficult times and celebrating every success of mine as if it were your own. I genuinely cannot imagine how I would have survived at Georgia Tech and reached

this stage without you. I am truly blessed to have you as a friend, colleague, and family. My heartfelt thanks also go to Ameen, Soosan, Mariam, and your parents for welcoming me into your family and making me feel at home.

I am grateful to the members of the Synergy Lab—Dr. Saeed Rashidi, Dr. Geonhwa Jeong, Dr. Taekyung Heo, Dr. William Won, Abhimanyu Bambhaniya Rajeshkumar, and Changhai Man. I learned a great deal from working with you all, and collaborating with Synergy Lab was a turning point that provided the mentorship and structure I needed at a crucial stage in my research career. I am especially thankful to Prof. Tushar Krishna for giving me the golden opportunity to collaborate with your group, and for the chance to learn and do research alongside some of the brightest people I have met.

My sincere thanks also go to the members of the TINKER Lab and to Anirudh Jain for being a wonderful collaborator and mentor throughout my Ph.D. journey. I am likewise grateful to Anirudh Sarma and Jinsun Yoo from the Embedded Pervasive Lab (EPL), and to Anish Saxena from the FAST Lab, for your support, discussions, and encouragement.

I would like to thank my industry collaborators and mentors at Hewlett Packard Labs. To Dr. Lianjie Cao and Dr. Puneet Sharma at the Network and Distributed Systems Lab (NDSL), thank you for being outstanding mentors and for helping me push the boundaries of innovation together. I feel fortunate to have interned at NDSL and to have met such wonderful people at HPE Labs. I also thank Michael Choi and Dr. Sudarshan Srinivasan for giving me the opportunity to work with you and for your mentorship and productive collaborations.

Special thanks are due to the TSO Research Technologists, whose behind-the-scenes work made our research possible: thank you for always being responsive and for helping resolve our technical issues quickly enough to keep research moving. We are incredibly fortunate to have such a reliable and supportive team behind us. In particular, I would like to thank David Howard, Garrett Briaud, Kwang Ho, and many others for their timely help and continued support.

To everyone in the ECE Department, especially Daniela Staiculescu and Tasha Torrence—thank you for being the front line of the department. Your support, patience, and unwavering professionalism made this journey smoother and more manageable. I am incredibly lucky to have had your help along the way.

To my friends and mentors beyond the lab, I owe a deep debt of gratitude. To Dr. Venkatesh Kodukula, thank you for being a sharp critic, a steadfast friend, and an encouraging mentor. Our endless discussions—from technology to the twists and turns of life—kept me intellectually engaged and motivated, both in research and beyond. To Dr. Manish Bhattarai, thank you for inspiring me to reach higher goals and for encouraging me to think outside the box. Your warm friendship, guidance, and support continually fueled my drive for excellence. To Dr. Prithyan Barua, my C++ guru, I certainly would not have survived my first two years without your kindness, support, and generosity. Thank you for being the person I could rely on in times of confusion and dilemma. To Dr. Akash Kota, thank you for being both an inspiration and a rock-solid source of support, always encouraging me to stay focused on my research. From our undergraduate days to your journey as a successful researcher in the United States, your example has been a powerful reminder of how hard work and perseverance can overcome the toughest obstacles. To Dr. Baaladhitya Uppalapati, your unwavering belief and support meant more than I can express. Each time you said “You can do this,” it gave me renewed confidence to move forward. Our friendship and shared struggles are memories I will carry for a lifetime.

I also want to thank my friends, Shreerang Dabade, Vivekananda Dayananda, Prasad Mate, and Jimin Lee. Thank you for always being there, for your constant support, and for making sure I stayed sane during these six years. I am very fortunate to have friends like you who checked in on me and kept me motivated.

My heartfelt thanks go to my friends and neighbors at Georgia Tech—Nitya M. V., Natalie Baborova, Vima Gupta, Ranjani Narayanan, Ajay Mandadi, and Venkat Balla—for your friendship and companionship. A special thanks to Veena Aunty for the wonderful

and delicious food you prepared for us. You all made my stay at Georgia Tech warm, comforting, and memorable.

I am also grateful to Dr. Ananda Samajdar, Dr. Poulami Das, Dr. Gururaj Saileshwar, Dr. Habib Ahmad, Sho Ko, and all my well-wishers and friends who were part of this Ph.D. journey. Your encouragement, advice, and moral support helped sustain me through many difficult phases.

My special thanks go to my brother-in-law, Amaranath Allampati, and my sister, Snigdha Maru, and their children, Sahasra and Suhaas. Your love and support as a family have been invaluable, and you truly became my home away from home. I am deeply indebted to you for standing by me throughout this long journey and for giving me a place of warmth, comfort, and belonging whenever I needed it most.

Finally, I want to thank my parents and grandparents, who endured countless hardships to ensure a bright future for me. My decision to leave a comfortable job and return to school to pursue a Ph.D. was not an easy one, and it affected them more than anyone else. Yet they never wavered in their trust, confidence, and love. In the darkest times, it was their belief in me that kept me moving forward. To my cousins, sisters, brothers, and extended family—this achievement belongs to all of us. I missed many important and beautiful milestones in your lives and could not be there to share them in person, yet you all continued to support and encourage me. I am profoundly grateful and blessed to have you as my family.

As I bring this dissertation to a close, I do not see it as an ending, but as the beginning of a new journey. I carry with me the memories, blessings, support, and love of everyone who has walked alongside me on this path. With that foundation behind me, I feel better prepared to tackle hard problems, seek out meaningful challenges, and strive to find innovative solutions in whatever comes next. Whatever I achieve in the future will always be rooted in the people and experiences that made this Ph.D. possible.

TABLE OF CONTENTS

Acknowledgments	v
List of Tables	xv
List of Figures	xvi
List of Acronyms	xx
Chapter 1: Introduction	1
1.1 The Memory Wall	2
1.2 Thesis Goals	4
1.3 Thesis Contributions	6
1.4 Thesis Organization	8
1.4.1 Bibliographic Notes	9
Chapter 2: Background and Motivation	10
2.1 Opportunities for Memory System Optimizations	10
2.1.1 High Performance Parallel Applications and Hardware Transactional Memory	10
2.1.2 Memory-Bound Server Workloads in Cloud Datacenters	13
2.1.3 Distributed AI Training at Cluster Scale	17

2.2	Emerging interconnect technologies	20
2.2.1	Compute Express Link (CXL)	21
Chapter 3: Enhancing On-Chip HTM Capacity through Safety Hints		23
3.1	Transactional capacity expansion through memory access classification . . .	24
3.2	HinTM design and implementation	26
3.2.1	Defining Safe and Unsafe Memory Accesses	26
3.2.2	HinTM's Classification and Integration Mechanisms	27
3.2.3	Fine-grained static memory access classification	28
3.2.4	ISA Extensions for Safety Information Propagation	29
3.2.5	Coarse-grained dynamic memory access classification	30
3.3	Evaluation methodology	33
3.3.1	Rationale for Capacity-Constrained Evaluation	35
3.4	Capacity Abort Reduction and Speedup with P8 HTM	36
3.4.1	Effect of Page Mode Aborts	38
3.4.2	Transactional Capacity Pressure Study	39
3.4.3	HinTM Effect on Larger HTMs	42
3.4.4	Evaluation Summary	44
3.5	Chapter Summary	45
Chapter 4: Augmenting Server-Level Memory Bandwidth through Idle I/O Resource Harvesting		47
4.1	SURGE Design	49
4.1.1	Architectural Support	50

4.1.2	SURGE Solo	50
4.1.3	SURGE Pod	51
4.1.4	Utility Comparison	52
4.2	SURGE Methodology	55
4.2.1	Workload Placement by Cluster Manager	55
4.2.2	Memory Traffic Split Strategy and Analytical Model	56
4.2.3	Offline Optimization and Generation of Traffic Split Curves	59
4.2.4	Runtime Selection and Application of Optimal Traffic Split	61
4.3	SURGE Implementation	63
4.3.1	Hardware Extensions	63
4.3.2	Software Support	64
4.4	SURGE Evaluation	65
4.4.1	Methodology	65
4.4.2	SURGE under Low I/O Utilization	69
4.4.3	SURGE with Higher I/O Utilization	71
4.4.4	Robustness of SURGE Traffic Split Methodology	74
4.4.5	SURGE in Co-located Workload Scenarios	77
4.4.6	Sensitivity to Salvage Link Latency and Bandwidth	82
4.5	Chapter Summary	83
 Chapter 5: Optimizing Cluster-Scale AI Training through Disaggregated Memory Expansion		85
5.1	The COMET Methodology	87
5.1.1	Workload Modeling	87

5.1.2	Training Strategy Configuration	88
5.1.3	Distributed Training Time Estimation	89
5.1.4	Compute delay estimation	90
5.1.5	Memory traffic estimation	92
5.1.6	Communication delay estimation	93
5.2	COMET Implementation	94
5.2.1	DL Model Analysis	95
5.2.2	Parallelization Strategy	96
5.2.3	Total Training Time Estimation	98
5.2.4	Cluster Parameter Reconfiguration and Iterative Modeling	99
5.3	Evaluation and Case Studies	100
5.3.1	Baseline Evaluation Setup and Workloads	100
5.3.2	Transformer Evaluation	100
5.3.3	Parallelization Strategy Impact on Performance and Memory Re- quirements	102
5.3.4	Effect of Memory System Design	104
5.3.5	Effect of Latency-Bandwidth Trade-Offs in Expanded Memory Sys- tems	105
5.3.6	Effect of Per-node Compute Capability	107
5.3.7	Effect of Networking Capability	108
5.3.8	DRAM Evaluation	110
5.3.9	Comparative DL Training on Different Clusters	112
5.4	COMET takeaways: versatility and speed	116
5.5	Chapter Summary	117

Chapter 6: Related Work	119
6.1 Prior work related to HTM capacity enhancements	119
6.2 Prior work related to memory system optimizations in server CPUs	122
6.3 Prior work related to Distributed DL Training performance and optimizations	123
Chapter 7: Future Research Directions	128
7.1 Rethinking Monolithic Server Architectures	129
7.2 Composable AI Systems: A Path Forward	130
7.3 CXL Shared Memory in Composable AI Systems	132
7.4 Dynamic Page Migration for Disaggregated Memory Systems	136
Chapter 8: Conclusion	138
References	140
Vita	167

LIST OF TABLES

2.1	Transactional capacity of commercial HTM implementations [50].	12
3.1	Simulation parameters.	33
4.1	System parameters used for simulation in ZSim.	66
5.1	Baseline NVIDIA DGX A100 system parameters.	101
5.2	Transformer model layers and dimensions.	101
5.3	Per-node details of various cluster configurations.	113

LIST OF FIGURES

1.1	Focus areas of thesis for memory system optimizations at multiple levels . . .	5
2.1	Percentage of runtime cycles wasted in page mode transitions.	13
2.2	Bandwidth characteristics of modern manycore processors. SKUs sampled from the AMD EPYC and Intel Xeon processor families.	14
2.3	Per-core memory bandwidth demands across our evaluated workloads. Horizontal dashed lines indicate the peak nominal available bandwidth for two modern CPUs, as well as the 50% bandwidth mark, beyond which queuing delays start increasing significantly.	15
3.1	Opportunity study for memory access classification.	25
3.2	State transition diagram for a page’s lifetime. Green memory accesses are dynamically marked safe.	32
3.3	Performance impact of HinTM on the P8 HTM configuration.	37
3.4	Memory access breakdown within transactions.	39
3.5	Transaction size CDFs for P8 configurations: baseline, HinTM-st, and HinTM. HinTM-st and baseline fully overlap in (a) and (b). Note the different y-axis scales.	41
3.6	Performance impact of HinTM on the P8S HTM configuration.	43
3.7	HinTM ’s impact on L1TM. Speedup and cycles wasted on page mode transitions.	45
4.1	SURGE hardware and software components.	49
4.2	Two SURGE architectural embodiments.	51

4.3	Salvage memory utility for SURGE as a function of pod size and P (probability of each individual salvage link having sufficient idle bandwidth to be salvaged by SURGE).	54
4.4	Methodology to analytically determine the optimal traffic split between primary and salvage memories.	57
4.5	Determining traffic split between primary and salvage memory as a function of memory bandwidth demand and salvage memory's latency/bandwidth characteristics.	61
4.6	I/O and memory traffic multiplexing block diagram.	63
4.7	Profiled DDR5-4800 memory load-to-use latency vs. demand (as % of peak) on AMD Bergamo.	67
4.8	Speedup, AMAT, and memory bandwidth usage for baseline vs SURGE in low-networking scenario. Dots indicate the traffic split (% of traffic to primary memory) for each workload, as determined by the SURGE methodology.	70
4.9	Impact of I/O traffic interference. <i>level₁_level₂</i> on the legends indicates ingress_egress path utilization.	72
4.10	Robustness to I/O interference. Traffic split configured expecting <i>low_low</i> I/O traffic in all cases. The plot shows only the subset of workloads that are sensitive to the SURGE-selected traffic split, while the mean refers to all 15 workloads.	74
4.11	AMAT and speedup for the bc workload across the range of possible primary:salvage memory traffic splits. Whiskers show standard deviation.	76
4.12	Deviation of SURGE-derived traffic split from optimal split, and impact on performance.	77
4.13	Speedup of SURGE over the baseline for three workload mixes. Each subplot shows a different workload mix (Mix-A/B/C) with cumulative baseline memory bandwidth demands of approximately 90%, 70%, and 50% of the primary memory bandwidth, respectively.	80
4.14	Impact of harvest memory's bandwidth-latency characteristics on SURGE's effectiveness. SURGE Solo/Pod deployments are more likely to offer characteristics landing toward the bottom left/top right corner, respectively.	82

5.1	COMET methodology overview.	87
5.2	Variation of per-node memory capacity requirements as a function of MP and DP degrees in a fixed-size cluster.	89
5.3	Roofline model. Attainable performance shifts for the same OI, depending on available memory bandwidth.	90
5.4	COMET implementation and workflow.	94
5.5	COMET evaluation space.	95
5.6	Per-node memory footprint for Transformer-1T model with baseline and different ZeRO-DP stages.	97
5.7	Cluster of 1024 A100 GPUs with expanded memories, grouped in 128 8-GPU pods. Link bandwidth is per direction.	100
5.8	Training runtime of Transformer-1T with varying MP/DP degree (norm. to best-performing MP8_DP128 config.).	103
5.9	Effect of memory bandwidth availability to the expanded memory system. The check-mark indicates the baseline configuration to which all the other values are normalized.	105
5.10	Effect of latency vs. memory bandwidth availability to the expanded memory system. All the values are normalized to the baseline MP8_DP128 configuration at 2039 GB/s memory bandwidth.	106
5.11	Effect of node compute capability relative to baseline A100 GPU (MP8_DP128 configuration).	108
5.12	Effect of network capabilities on cluster-scale performance. The check-mark indicates the baseline configuration to which all the other values are normalized. 300/31.25 GB/s is currently a typical NVLink/InfiniBand configuration.	109
5.13	Impact of relative inter-/intra-pod network bandwidth allocation on training runtime. The total bandwidth is kept constant at 331.25 GB/s across all the ratios. The highlighted 1:9.6 ratio represents the baseline cluster's configuration.	110
5.14	DLRM's training performance normalized to 64 nodes with 2TB/s memory bandwidth.. . . .	112

5.15	Evaluated configurations for Dojo and TPU clusters.	114
5.16	Comparison of runtime across different cluster configurations, normalized to cluster A0.	115
7.1	Illustration of compute-communication overlap for fine- vs. coarse-grained PCIe transfers.	133
7.2	Estimated single iteration time with fine- vs. coarse-grained PCIe transfers through analytical modeling over baseline RDMA system.	133

LIST OF ACRONYMS

AI	Artificial Intelligence
AMAT	Average Memory Access Time
BP	Backward Propagation
CAGR	Compound Annual Growth Rate
CPU	Central Processing Unit
CXL	Compute Express Link
DDIO	Data Direct I/O
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
DL	Deep Learning
DLRM	Deep Learning Recommendation Model
DNN	Deep Neural Network
DP	Data Parallelism
DPU	Data Parallel Unit
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
EM	Expanded Memory
EP	Expert Parallelism
FP	Forward Propagation
FSDP	Fully Sharded Data Parallelism
GDDR	Graphics Double Data Rate
GEMM	General Matrix Multiplication

GPU Graphics Processing Unit
HBM High Bandwidth Memory
HinTM Hinted HTM
HPC High Performance Computing
HTM Hardware Transactional Memory
I/O Input/Output
IG Input Gradients
ILP Integer Linear Programming
ISA Instruction Set Architecture
LLC Last Level Cache
LLM Large Language Model
LM Local Memory
ML Machine Learning
MLP Multi Layer Perceptron
MoE Mixture-of-Experts
MP Model Parallelism
NIC Network Interface Card
NLP Natural Language Processing
NVM Non-Volatile Memory
NVMe Non-Volatile Memory express
OI Operational Intensity
OS Operating System
PCIe Peripheral Component Interconnect Express
PP Pipeline Parallelism
PT Page Table
RDMA Remote Direct Memory Access
RTM Restricted Transactional Memory

RX ingress

SLO Service Level Objective

SRAM Static Random Access Memory

STM Software Transactional Memory

TCO Total Cost of Ownership

TLB Translation Lookaside Buffer

TM Transactional Memory

TP Tensor Parallelism

TPU Tensor Processing Unit

TSVs Through Silicon Vias

TX egress

TXN Transaction

VIPS Valid/Invalid – Private/Shared

WG Weight Gradients

SUMMARY

The rapid proliferation of digital services—from web analytics and cloud platforms to social media and generative AI—has driven an unprecedented surge in data generation and processing demands, positioning data centers as the operational core of today’s digital economy. As hyperscale infrastructures expand to meet this exponential growth, the memory subsystem has emerged as a critical performance and cost bottleneck. Modern server nodes face mounting pressure to sustain high-capacity, high-bandwidth, and low-latency memory access as workloads increasingly rely on large in-memory datasets and parallel execution across thousands of cores. However, traditional scaling approaches—such as adding DRAM channels or relying on remote memory through RDMA—face physical, technological, and economic limitations. The resulting “memory wall” manifests as three interdependent challenges: limited capacity, constrained bandwidth, and rising latency, all intensified by the slowdown of Moore’s Law and the end of Dennard scaling.

This thesis addresses these challenges through a holistic, cross-layer co-design approach that enhances memory system performance across three hierarchical levels—chip, server, and cluster. At the chip level, HinTM introduces compiler- and hardware-assisted mechanisms to mitigate capacity aborts in Hardware Transactional Memory systems, thereby improving on-chip cache utilization and parallel execution efficiency. At the server level, SURGE dynamically harvests idle I/O bandwidth over CXL links to augment effective memory bandwidth and reduce memory access latency in bandwidth-bound workloads. Extending to the cluster scale, COMET provides a unified design-space exploration framework that jointly optimizes compute, memory, and interconnect provisioning for distributed AI and HPC workloads. Collectively, these contributions demonstrate that co-optimizing architectural mechanisms with workload and hardware characteristics can overcome the fundamental limitations of memory capacity and bandwidth scaling, enabling sustained performance improvements across modern datacenter systems.

CHAPTER 1

INTRODUCTION

Modern-day online services—spanning web analytics, cloud-based enterprise platforms, social media, digital financial systems, and generative Artificial Intelligence (AI) applications—have become indispensable components of today’s digital infrastructure and everyday life. As of 2025, Google accounts for 91.65% of all global search queries, processing an estimated 9.1 billion searches per day [1] while the social media giant Meta reports 3.07 billion monthly active users across its platforms, reflecting steady user growth worldwide [2]. Meanwhile, propelled by the generative AI boom, OpenAI serves approximately 800 million weekly active users, processing over 2.3 billion messages per week through ChatGPT [3]. With every user continuously generating data and each request probing vast datasets, the demand for rapid, large-scale data access is growing at an unprecedented pace. To meet these escalating requirements, online service providers operate vast datacenters housing hundreds of thousands of multi-processor servers that collectively form the backbone of the global digital economy.

Driven by the explosive expansion of the digital ecosystem, datacenters are scaling at an extraordinary rate. The global datacenter market, valued at USD 242.7 billion in 2024, is projected to grow more than double—surpassing USD 584 billion by 2032—with a Compound Annual Growth Rate (CAGR) of 11.7% [4, 5, 6]. To meet the increasing demands of users and applications, major cloud service providers are deploying hyperscale and even planetary-scale datacenters, each housing tens of thousands of servers [7, 8, 9]. However as the datacenter capacity continues to grow, so do the associated costs. A significant portion of a datacenter’s Total Cost of Ownership (TCO) stems from acquiring and maintaining physical infrastructure—servers, storage, and networking components. Within the server cost breakdown, memory has emerged as a dominant contributor due to its poor scaling

characteristics [10]. *Dynamic Random Access Memory (DRAM)* alone accounts for nearly 50% of server costs in Microsoft Azure and 40% of rack costs at Meta [11]. The ongoing surge in generative AI workloads, compounded by an impending shortage of NAND and DRAM, is expected to further inflate the cost of memory and storage hardware [12].

Most datacenter and cloud applications are inherently parallel and distributed, scaling up across multiple cores and out across multiple servers. Beyond compute performance, many of these applications require their working data to remain memory-resident to meet stringent latency requirements [13]. In addition, the growing dominance of generative AI workloads has intensified the need for specialized accelerator nodes featuring higher memory bandwidth and larger DRAM capacity. Traditionally, scaling memory capacity has relied on either provisioning additional memory channels on processor sockets at design time or leveraging Remote Direct Memory Access (RDMA) technologies to access remote memory distributed across servers via high-speed interconnects such as InfiniBand [14] and Ethernet [15], or even over specialized memory fabrics [16]. However, the scalability of these approaches is limited by technological constraints and the high latency of network communication. For particularly challenging applications—such as graph processing over large, hard-to-partition data structures or AI inference that accesses large, irregularly distributed data structures (e.g., key-value caches storing model activations)—distributed execution often incurs frequent inter-server communication, which can easily dominate overall request latency and lead to violations of Service Level Objective (SLO)s. Therefore, having access to a memory system with large capacity, high bandwidth, and low latency in the datacenter becomes a paramount determinant of application performance.

1.1 The Memory Wall

Memory system performance remains a critical determinant of overall system efficiency in modern datacenter architectures. Over the years, the performance gap between processors and main memory has continued to widen [17, 18]. As the demand for computational

power and parallelism continues to surge, systems are steadfastly approaching towards the three walls of memory system—capacity, bandwidth, and latency. This challenge is further exacerbated by the slowdown of Moore’s Law [19] and the end of Dennard scaling [20], which limit conventional means of improving memory capacity, bandwidth, and latency. Physical constraints in device technology, packaging density, power delivery, and thermal dissipation impose hard limits on how much memory performance can scale [10, 21].

Efforts to address these limitations through alternative technologies, such as 3D XPoint [22], aimed to expand memory capacity using denser Non-Volatile Memory (NVM) were introduced. However, these solutions have fallen short of meeting the growing capacity demands due to practical and economic limitations [23], as well as the substantially higher access latencies that undermine overall performance. Expanding memory beyond a single node—by accessing distributed memory across servers using high-speed datacenter networks via RDMA—offers another path for capacity scaling. Yet, this approach provides limited scalability and incurs significant performance degradation when large or disjoint memory regions are accessed [24], where the much higher latency of remote memory becomes a primary bottleneck.

Increasing memory bandwidth presents a parallel set of challenges, as sustaining higher throughput typically requires either adding more memory channels or operating existing interfaces at higher frequencies. Each memory channel demands hundreds of pins, a resource that is inherently scarce and has been scaling slowly over successive processor generations [25, 26, 27]. Similarly, frequency scaling reduces supported capacity (e.g., limiting to one Dual Inline Memory Module (DIMM) per channel) and introduces stringent signal-integrity constraints [28, 29, 30]. Technologies such as High Bandwidth Memory (HBM) and Graphics Double Data Rate (GDDR) address pin limitations by stacking DRAM dies and using Through Silicon Vias (TSVs) or short, high-speed signaling links to provide substantially higher bandwidth.

Other approaches seek to relieve pressure on off-chip bandwidth by dramatically in-

creasing the amount of on-chip memory. Large-die and wafer-scale integration techniques offer substantial local Static Random Access Memory (SRAM) capacity and reduce the need for frequent off-chip accesses [31]. However, these designs face fundamental manufacturing-yield constraints, thermal and power-delivery challenges, and long on-chip interconnect paths—factors that significantly increase fabrication cost, reduce die yield, and limit their feasibility beyond highly specialized systems [32].

While these high-bandwidth memory architectures offer impressive throughput, they still suffer from high manufacturing cost, limited total capacity, elevated standby power, and complex signal-integrity requirements necessitating robust Error Correcting Code (ECC) support. Compounding these challenges, the continued trend toward many-core Central Processing Unit (CPU)s further reduces the effective memory bandwidth available per core.

Recent advances in interconnect technologies, such as Compute Express Link (CXL), offer promising opportunities for seamlessly expanding memory capacity and bandwidth over high-speed serial interfaces, while enabling memory and Input/Output (I/O) traffic to share the same physical link. These capabilities pave the way for rethinking and redesigning memory systems for greater flexibility, scalability, and efficiency in achieving higher bandwidth and capacity. However, leveraging off-chip memory over serial links introduces additional latency, necessitating workload- and hardware-aware memory system optimizations to fully exploit the potential of these emerging architectures.

1.2 Thesis Goals

The overarching goal of this thesis is to enhance the performance of parallel and memory bandwidth-intensive workloads through memory system optimizations that are co-designed with workload characteristics and the underlying hardware capabilities. Specifically, the objective is to achieve substantial performance improvements for memory-bound workloads across both High Performance Computing (HPC) and cloud environments.

To achieve this goal, we systematically analyze workload behaviors and hardware fea-

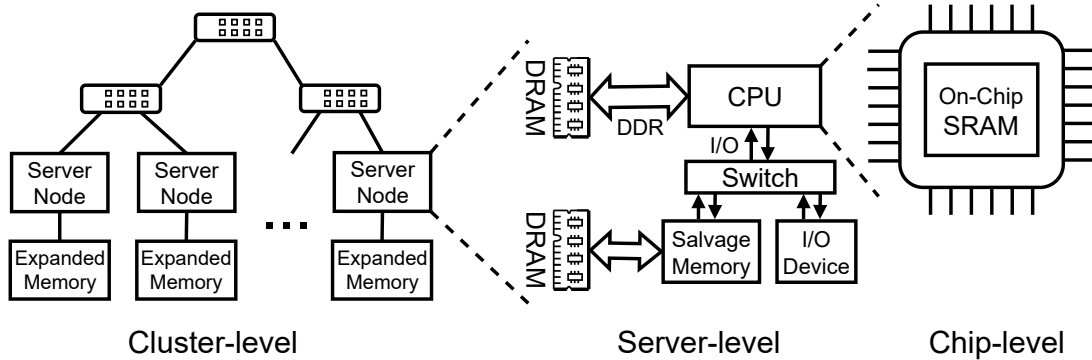


Figure 1.1: Focus areas of thesis for memory system optimizations at multiple levels

tures to uncover inefficiencies and identify optimization opportunities at multiple levels of the system as illustrated in Figure 1.1. The insights derived from this analysis inform the design of co-optimized memory system solutions that integrate architectural mechanisms with system software support to address key performance bottlenecks in the memory system. Building upon this foundation, in this thesis, we demonstrate a set of co-designed memory system optimizations tailored to diverse workload and hardware characteristics across three levels of system hierarchy—chip, server, and cluster to enhance. Collectively, these optimizations aim to enhance effective memory capacity, improve bandwidth utilization, and increase overall system performance for parallel and bandwidth-intensive applications.

We begin by focusing on improving the effective capacity of on-chip SRAM cache memory to enhance the performance of HPC workloads, a class of applications that rely heavily on high-performance parallel programming paradigms. We identify opportunities to enhance the utilization and effective capacity of these limited on-chip caches through software–architecture co-design techniques applied at the processor cache level. Next, at the server level, we illustrate how idle off-chip bandwidth can be harvested to increase available memory bandwidth, thereby reducing the Average Memory Access Time (AMAT) for bandwidth-intensive server and cloud workloads. Finally, at the cluster level, we propose a holistic methodology to model and study the effects of hardware–algorithm co-design in distributed AI training workloads, focusing on how per-node memory expansion

via serial interconnects can be leveraged to improve training performance and scalability in such distributed systems.

Thesis Statement

Holistic optimization across multiple levels of the memory hierarchy, through architectural techniques supported by system software and tailored to workload characteristics and underlying hardware capabilities, can alleviate bandwidth and capacity limitations, improving the performance of parallel and bandwidth-intensive workloads.

In this thesis, we advocate for a co-designed approach to memory system architecture, tailoring architectural mechanisms and supporting system software to the specific characteristics of workloads and hardware platforms. By systematically exploring optimization opportunities across the chip, server, and cluster levels, this work demonstrates how holistic memory system co-design can substantially enhance the performance of parallel and bandwidth-intensive applications, effectively addressing the challenges imposed by the memory wall.

1.3 Thesis Contributions

This thesis introduces a series of memory system optimizations tailored to workload and hardware characteristics, aimed at improving the performance of parallel and bandwidth-intensive applications in cloud and datacenter environments. We identify optimization opportunities at three distinct levels of the memory hierarchy—processor cache, server, and cluster—using different application and hardware scenarios. For each level, we develop and evaluate a proof-of-concept implementation within a microarchitectural/system simulation framework to demonstrate the effectiveness of these optimizations by leveraging architectural mechanisms supported by system software.

First, we focus on optimizing the effective on-chip memory capacity at the processor cache level to enhance the performance of scientific and parallel workloads in the context

of highly parallel programming paradigm of Hardware Transactional Memory (HTM). We propose Hinted HTM (HinTM), a novel hardware-software co-design technique to eliminate costly capacity aborts in the limited-capacity commercial HTM implementations by leveraging architectural and compiler-assisted optimizations. At the heart of HinTM is a *memory access classification* mechanism through (i) a fine-grained, static design-time technique at cacheline granularity, and (ii) a coarse-grained, dynamic runtime mechanism at page granularity. Together, these approaches significantly reduce capacity aborts of hardware-assisted transactions, delivering substantial performance improvements with negligible overhead.

Second, motivated by the vast underutilization of precious off-chip bandwidth in server CPUs, we propose the dynamic harvesting of idle I/O bandwidth to alleviate the memory wall in bandwidth-constrained many-core processors. A static, design-time allocation of off-chip bandwidth between I/O and memory often leads to inefficiencies and *resource stranding*, as I/O bandwidth utilization is highly workload-dependent and typically exhibits asymmetric usage across ingress and egress directions, leaving a substantial portion of the available off-chip bandwidth underutilized in one or both directions. In contrast, dynamically sharing bandwidth between I/O and memory improves overall bandwidth utilization and necessitates judicious memory system optimizations across the hardware and software stack. To this end, we introduce SURGE, a software-assisted architectural mechanism that opportunistically salvages unused I/O bandwidth to augment available memory bandwidth with an additional *salvage memory* accessed over the serial I/O links. We demonstrate two different architectural designs, coupled with a methodology and implementation framework to determine optimal traffic split between main memory and salvage memory under varying workload and hardware conditions, thereby amortizing the latency of serial I/O links.

Finally, at the cluster level, we investigate memory system optimizations for distributed AI training workloads, focusing on expanding per-node memory capacity and bandwidth of

a training node to accelerate the training of large-scale deep learning models such as Large Language Model (LLM)s and Deep Learning Recommendation Model (DLRM)s. To this end, we develop COMET, a holistic cluster design methodology that enables rapid design space co-exploration of model training strategies and key cluster hardware resources, to jointly evaluate the impact on performance of distributed Deep Learning (DL) training applications. Our framework provides a systematic, step-by-step workflow for modeling and co-optimizing compute, memory, and network resources. We examine large-scale model training across diverse cluster configurations to provide actionable insights for cluster designers to improve performance, scalability, and resource efficiency. Furthermore, we demonstrate the effectiveness of COMET methodology through case studies analyzing the impact of memory expansion on distributed deep learning training performance at a cluster scale.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background on the fundamental concepts and technologies pertinent to this research. Chapter 3 presents HinTM, a software–hardware co-design framework that mitigates capacity limitations in Hardware Transactional Memory through compiler and architectural mechanisms utilizing memory safety hints. Chapter 4 introduces SURGE, a software-assisted architectural technique that enhances effective memory bandwidth by dynamically harvesting idle I/O bandwidth resources. Chapter 5 describes COMET, a comprehensive design-space exploration methodology for large-scale distributed deep learning clusters, jointly analyzing the effects of parallelization strategies and resource provisioning on training performance. Chapter 6 discusses related work, Chapter 7 outlines potential avenues for future research, and Chapter 8 concludes the thesis.

1.4.1 Bibliographic Notes

This thesis was conducted under the supervision of Prof. Alexandros Daglis. Chapter 3 is partially based on a paper published in the *29th IEEE International Symposium on High-Performance Computer Architecture* (HPCA) in 2023 [33], with portions developed in collaboration with Anirudh Jain. Chapter 4 expands on work previously released as an arXiv pre-print [34], incorporating additional methodology and evaluation developed during the course of this dissertation. Chapter 5 builds upon a preprint published on arXiv in 2022 [35] and a poster presented at the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) in 2024 [36], and includes contributions from Saeed Rashidi, Taekyung Heo, and Abhimanyu Rajeshkumar Bambhaniya, under the advisement of Prof. Tushar Krishna. Finally, the ideas presented in Chapter 7 are the result of collaborative efforts with Dr. Lianjie Cao and Dr. Puneet Sharma at Hewlett Packard Enterprise Labs.

CHAPTER 2

BACKGROUND AND MOTIVATION

This chapter introduces the motivation and foundational context for optimizing memory systems across diverse workload families and hardware platforms. It highlights key opportunities for co-design and optimization at multiple levels of the memory hierarchy—ranging from on-chip caches to server- and cluster-scale memory systems. For each category, we outline the underlying challenges, present workload-driven insights, and discuss how targeted memory system optimizations can enhance performance, scalability, and efficiency. Finally, we examine emerging trends in interconnect technologies, emphasizing their growing role in enabling new opportunities for holistic memory system optimization.

2.1 Opportunities for Memory System Optimizations

2.1.1 High Performance Parallel Applications and Hardware Transactional Memory

High-performance scientific and parallel applications rely on fine-grained synchronization mechanisms to achieve scalability on many-core processors. In most cases, synchronization is implemented through software-based locking mechanisms (e.g., POSIX `pthread` mutexes), which must be explicitly inserted and managed by the programmer during application development.

In contrast, Transactional Memory (TM) [37] provides a high-performance programming abstraction that simplifies concurrency control in shared-memory parallel applications. By delegating the coordination of concurrent memory accesses to the underlying system, TM relieves programmers from the complexity of manual synchronization. As parallelism became fundamental to modern computing, the promise of improved programmability and scalability inspired extensive research into TM systems [38, 39, 40, 41, 42, 43,

43, 44, 45, 46, 47, 48].

The fundamental concept underlying TM is the concurrent and optimistic execution of critical code sections, referred to as Transaction (TXN)s, without acquiring explicit locks. During execution, if a thread encounters contention due to a concurrent modification of a shared data structure by another thread, the conflicting TXN is aborted, all speculative updates are rolled back, and the aborting thread retries the transaction at a later time. Otherwise, if no conflicts are detected, the TXN commits successfully, and its modifications are atomically made visible to the rest of the system.

TM implementations generally fall into two broad categories: Software Transactional Memory (STM) and Hardware Transactional Memory (HTM). STM systems rely solely on software mechanisms, maintaining transaction metadata in main memory to detect conflicts among concurrently executing transactions. This approach, however, incurs significant off-chip memory accesses and associated latency overheads. In contrast, HTM systems leverage on-chip SRAM-based hardware buffers to track a transaction's read and write sets, using the cache coherence protocol to detect conflicts among concurrent transactions. By embedding transactional support into the hardware, HTM enables fine-grained concurrency control and low-latency conflict detection, offering a compelling alternative to traditional locking-based synchronization.

Over the past decade, major CPU vendors such as Intel and IBM have released commercial HTM implementations. In these systems, transactional state buffering is integrated into the memory hierarchy—either within cache structures (e.g., Intel's Restricted Transactional Memory (RTM) [44], IBM zEC12 [49]) or within dedicated hardware buffers coupled to the cache hierarchy (e.g., IBM POWER8 [42]).

A key limitation of existing HTM systems is their restricted transactional capacity, bounded by the size of underlying hardware structures—whether dedicated buffers or cache resources [51]. While numerous academic proposals have explored more flexible or scalable HTM designs, commercial implementations impose strict constraints on transaction

Table 2.1: Transactional capacity of commercial HTM implementations [50].

CPU name	TXN ld capacity	TXN st capacity
Blue Gene/Q	20MB	20MB
zEC12	1MB	8KB
POWER8	8KB	8KB
Intel Core i7	4MB	22KB

size and system behavior, limiting their practical applicability. Regardless of architectural differences, all commercial HTMs enforce an implicit upper bound on supported transaction size that directly correlates with the capacity of the underlying hardware buffers [52]. Table 2.1 summarizes the transactional capacities of widely used commercial HTM implementations.

When a transaction’s buffered state exceeds the hardware’s capacity, the transaction aborts and typically falls back to a software handler, which employs coarse-grained locking to guarantee atomicity. This fallback path eliminates parallelism during its execution and can severely degrade performance. Figure 2.1 illustrates the fraction of runtime spent on capacity aborts for a set of scientific and parallel benchmarks executed with eight threads (four for `genome` and `yada`) on a POWER8-like HTM configuration. While some applications such as `kmeans` and `ssca2` use small transactions that never exceed hardware buffer limits, others suffer substantial performance degradation, with capacity aborts accounting for up to 89% of total runtime and 22% on average.

On the other hand, larger-capacity HTM implementations are essential to fully realize the benefits of the TM programming paradigm. However, the limited capacity of current HTMs constrains transaction sizes, preventing many applications from exploiting the full potential of hardware transactional execution. Consequently, it is imperative to reduce or eliminate capacity aborts to improve application performance. Since physically expanding the on-chip SRAM buffers is not feasible, alternative mechanisms are required to *selectively track only the most critical metadata* of a transaction’s memory accesses within the

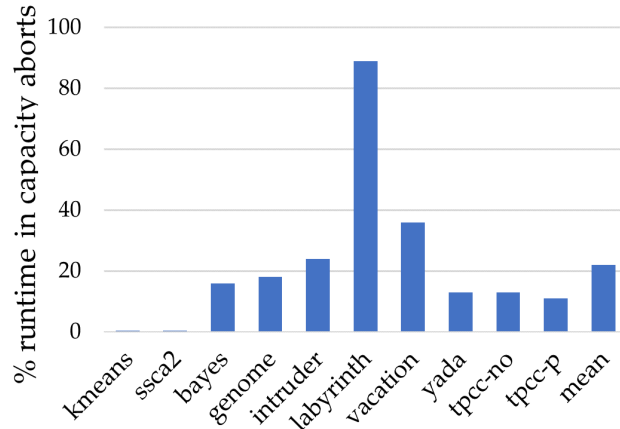


Figure 2.1: Percentage of runtime cycles wasted in page mode transitions.

available hardware capacity of on-chip SRAM buffers, thereby enabling larger effective transaction sizes.

To address this challenge, we employ memory access classification mechanisms, introduced in Chapter 3, guided by detailed characterization of memory accesses of each TXN observed across a range of parallel and HPC workloads. Building on these insights, we propose HinTM, a novel compiler-assisted architectural optimization that provides *safety hints* to the HTM controller through a combination of static and dynamic mechanisms. These hints enable the controller to track only the critical subset of memory accesses within the limited on-chip SRAM buffers, thereby effectively expanding the transactional tracking capacity for supporting larger transaction sizes in HTM systems.

2.1.2 Memory-Bound Server Workloads in Cloud Datacenters

The continuous growth of core counts in modern server-grade CPUs has enabled higher levels of workload consolidation and improved datacenter efficiency. However, these many-core designs have increasingly exposed a critical limitation: the stagnation of per-core memory bandwidth scaling. While compute density continues to rise with every processor generation, the available off-chip memory bandwidth per core has steadily declined, reinforcing the long-standing *memory bandwidth wall* problem [53]. This imbalance stems

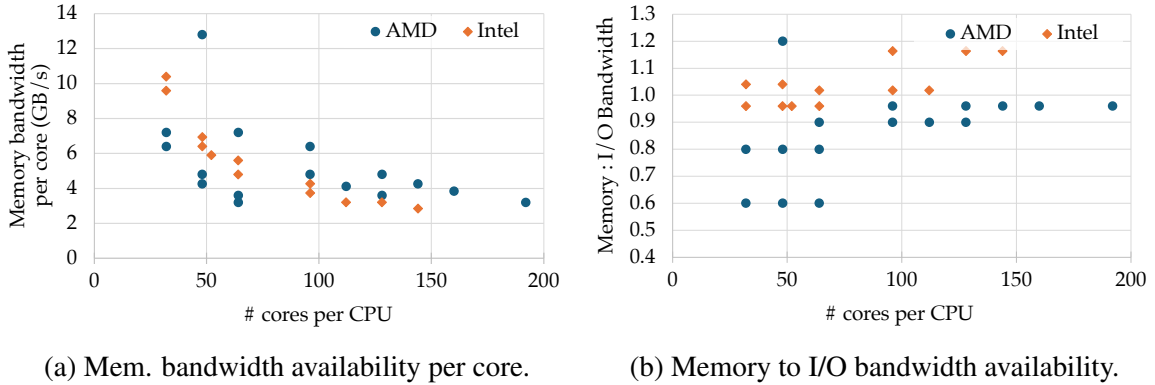


Figure 2.2: Bandwidth characteristics of modern manycore processors. SKUs sampled from the AMD EPYC and Intel Xeon processor families.

primarily from fundamental architectural constraints in Double Data Rate (DDR)-based memory systems. As core counts increase, memory bandwidth does not scale proportionally, primarily due to limits on the number of DDR channels a processor can support. Each DDR channel requires over 160 signal pins on the processor package, while total pin count remains constrained by package area, power delivery, and signal integrity considerations [54]. Consequently, the number of DDR channels cannot grow proportionally with compute cores. Although successive DDR generations have achieved higher data rates, the improvement rate has slowed due to increasing signal integrity and power delivery challenges [28, 55, 56]. The combined effect is a sharp decline in available memory bandwidth per core in modern many-core processors.

Figure 2.2a illustrates this trend, showing the per-core memory bandwidth of recent high-end server CPUs as a function of core count. For example, Intel’s Sierra Forest processor (144 cores) provides approximately 2.8 GB/s of bandwidth per core, while AMD’s Bergamo (128 cores) offers about 3.6 GB/s per core. Many cloud and datacenter workloads now operate near or beyond these limits, creating substantial contention for memory bandwidth. Once memory bandwidth utilization exceeds roughly 50%, queuing delays and contention cause average memory access time (AMAT) to rise sharply resulting in substantially longer memory access times [54, 57, 58].

Figure 2.3 shows the per-core memory bandwidth demands of a representative set of

cloud and datacenter workloads. These workloads frequently operate at or above the per-core bandwidth limits of modern CPUs, resulting in substantial queuing delays and insufficient bandwidth availability. Consequently, memory bandwidth has emerged as a primary bottleneck in modern many-core server processors.

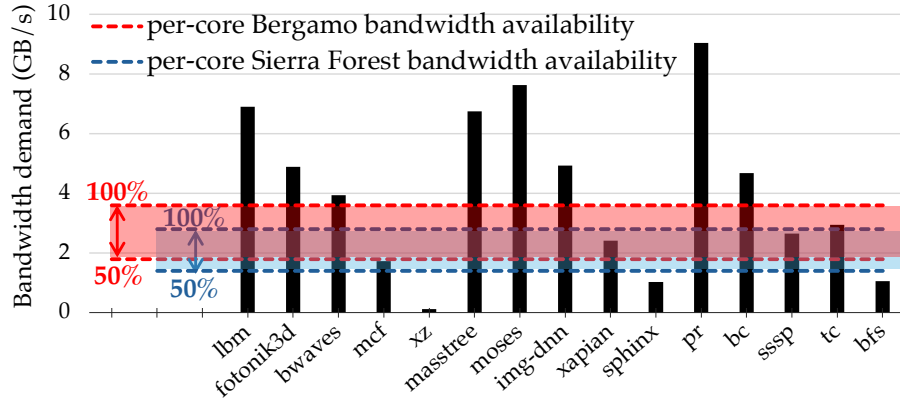


Figure 2.3: Per-core memory bandwidth demands across our evaluated workloads. Horizontal dashed lines indicate the peak nominal available bandwidth for two modern CPUs, as well as the 50% bandwidth mark, beyond which queuing delays start increasing significantly.

Static Off-Chip Bandwidth Partitioning. Beyond intrinsic DDR limitations, another major constraint lies in how off-chip bandwidth resources are statically partitioned at design time between memory and I/O subsystems. A processor’s off-chip memory bandwidth availability is capped by the limited number of pins available per socket. Each processor socket has a fixed number of pins that must serve both memory channels (DDR) and I/O interfaces (typically Peripheral Component Interconnect Express (PCIe)). Physical and mechanical constraints on pin density scaling further limit available off-chip bandwidth [59]. The absolute off-chip data movement rate a processor can achieve over its limited pins also depends on the signaling and protocol employed—for example, serial interfaces are considerably more pin-efficient than DDR [54].

Across modern processors, the memory-to-I/O bandwidth allocation ratio typically ranges between 0.6 and 1.2 (0.94, on average, among the processors we studied), as shown

in Figure 2.2b. While this roughly balanced provisioning simplifies design, it introduces resource stranding: one subsystem’s unused bandwidth cannot be repurposed to alleviate pressure on the other. For instance, when a workload saturates memory bandwidth but utilizes little I/O, a large portion of the off-chip bandwidth remains idle, despite the system being memory-bound. Conversely, network- or storage-intensive workloads may leave the memory channels underused.

Eliminating stranding requires treating the collective off-chip bandwidth as interchangeable between memory and I/O. Prior datacenter workload studies confirm that this inefficiency is pervasive. Our key insight is that datacenter servers often leave much of their I/O bandwidth under-utilized. For instance, PCIe-attached SSDs are often used only as boot volumes [60, 61], so they see little activity during normal operation. For example, Benson et al. report that (i) 70% of server network links use less than 1% of capacity [62], and (ii) the 95th-percentile utilization over 10 days is under 25% [63]. Similarly, Zhang et al. [64] show that, despite significant inter-rack traffic, host Network Interface Card (NIC) utilization within a rack remains low for both ingress and egress, leaving high-bandwidth NICs largely idle. This under-utilization arises from several factors: the absence of network traffic during job computation phases, highly skewed traffic volumes between hosts during communication, and heterogeneous resource requirements across jobs that exacerbate resource fragmentation. As a result, a substantial fraction of the I/O bandwidth provisioned at CPU design time remains unused across datacenter servers

Dynamic I/O Bandwidth Harvesting. To address bandwidth underutilization and improve off-chip resource efficiency, we introduce SURGE, a software-assisted architectural mechanism that dynamically converts idle I/O bandwidth into usable memory bandwidth. Leveraging the fine-grained multiplexing and full-duplex capabilities of modern serial interconnects such as CXL, SURGE enables on-demand sharing of off-chip bandwidth between memory and I/O domains. This dynamic bandwidth reallocation mitigates memory

bottlenecks without adding physical channels or compromising I/O performance, enhancing overall bandwidth efficiency for memory-bound server workloads.

In Chapter 4, we present the detailed architectural design, methodology, software interface, implementation, and evaluation of SURGE, demonstrating its effectiveness in significantly enhancing the performance and bandwidth efficiency for memory-bound workloads while maintaining efficient utilization of I/O resources in datacenter servers.

2.1.3 Distributed AI Training at Cluster Scale

Recent advancements in artificial intelligence (AI) have been driven by increasingly large and complex deep learning (DL) models, particularly large language models LLMs containing hundreds of billions to trillions of parameters [65, 66, 67, 68, 69, 70, 71, 72, 73]. Recent multi-modal models such as GPT-4 [71], Gemini Ultra [70], and Llama 4 Behemoth [68] employ Mixture-of-Experts (MoE) architectures [74, 65] that scale to unprecedented sizes. However, the immense memory footprint of these models—combined with the limited capacity of on-node memory—necessitates distributed training across large clusters of compute nodes interconnected through high-bandwidth networks to sustain acceptable throughput.

As models continue to scale, both training and inference increasingly strain available compute, memory, and interconnect resources. For instance, training a 1-trillion parameter model using the mixed-precision Adam optimizer [75] requires approximately 16 terabytes of model memory, far exceeding the memory capacity of any single node. Consequently, the training of such large-scale models is done through hybrid parallelization strategies that integrate multiple forms of data and model partitioning, enabling efficient distribution of vast datasets and model parameters across thousands of interconnected nodes.

A typical training iteration involves two main computational phases: the Forward Propagation (FP) pass and the Backward Propagation (BP) pass. During the FP pass, the model processes input data to compute intermediate activations at each layer, while the BP pass

computes gradients of the loss function with respect to model parameters for weight updates. To distribute these computations efficiently, various parallelization strategies—often used in hybrid combinations—partition model parameters, activations, and gradients across nodes in distinct ways:

- **Data Parallelism (DP)** [76]: Each node maintains a complete replica of the model while processing a unique shard of the training dataset. After each iteration, model updates from all nodes are aggregated using collective communication primitives such as *AllReduce*, ensuring global synchronization for the next iteration.
- **Tensor Parallelism (TP)** [77, 78]: The model is partitioned across multiple nodes, with each node responsible for a subset of the model’s parameters. The input data is replicated across nodes, and collective communication is required to exchange intermediate activations and gradients during both FP and BP passes. Depending on how the model is partitioned, TP manifest as either Model Parallelism (MP) or Pipeline Parallelism (PP) [79, 80, 81], wherein model layers are divided into pipeline stages and distributed across nodes.
- **Fully Sharded Data Parallelism (FSDP)** [82, 83, 84]: A hybrid strategy that shards both model parameters and optimizer states across nodes, reducing per-node memory usage while maintaining communication efficiency via high-speed collective synchronization.
- **Expert Parallelism (EP)** [85]: A strategy used in MoE-based architectures, where distinct expert modules are distributed across nodes. For each input, only a subset of experts is activated, enabling efficient scaling by avoiding full model replication across all nodes.

Beyond parallelization, offloading techniques such as ZeRO-Offload [86] and ZeRO-Infinity [87] extend memory capacity by spilling model states, activations, and optimizer

buffers into slower tiers of the memory hierarchy, including non-volatile devices such as Non-Volatile Memory express (NVMe). While these techniques alleviate per-node memory pressure and reduce the scale-out requirement, they introduce substantial data transfer overhead due to the limited bandwidth of slower memory tiers, leading to degraded training throughput [87].

Looking forward, with model sizes projected to exceed 10 trillion parameters [88, 66], distributed training at this scale will demand unprecedented levels of aggregate memory capacity and bandwidth. However, because the bandwidth of slower memory tiers remains significantly lower than PCIe or network interconnects, excessive reliance on offloading causes severe performance degradation and suboptimal resource utilization.

Therefore, *scalable and efficient memory system design* plays a pivotal role in sustaining performance and energy efficiency in distributed AI training. A memory system capable of providing higher effective capacity and bandwidth—combined with system-level optimizations attuned to workload behavior and hardware characteristics—can substantially influence cluster performance and cost efficiency. In particular, per-node memory expansion through technologies such as CXL enables larger in-memory model footprints, reducing inter-node communication and synchronization overheads. Depending on the training configuration, such expansion can be leveraged to either maximize training throughput or minimize total cluster size required to train a target model.

The performance and efficiency of a distributed deep learning cluster ultimately depend on a combination of factors: per-node computational capability, available memory capacity and bandwidth, and the inter-node network bandwidth. Beyond raw hardware specifications, performance is also shaped by the training structure—that is, the selected parallelization strategy—which dictates workload characteristics, memory footprint per node, and the extent to which each resource is utilized.

Accordingly, optimizing distributed training requires holistic co-design of the parallelization strategy alongside the cluster’s compute, memory, and interconnect architecture.

In Chapter 5, we introduce COMET, our cluster-level co-design methodology that systematically evaluates these interactions and identifies optimal configurations where per-node memory expansion and bandwidth provisioning yield measurable improvements in training performance and cost efficiency across diverse distributed learning workloads.

2.2 Emerging interconnect technologies

The rapid growth of data-intensive and AI-driven workloads has intensified the need for scalable, high-bandwidth, and low-latency memory access across compute systems. Traditional DIMM-based DRAM architectures, though performant, are limited by physical capacity, signal integrity constraints, and socket-level pin bandwidth. To address these challenges, the industry has increasingly turned to high-speed serial interconnect technologies that extend memory capacity and bandwidth beyond conventional boundaries.

Early standards such as OpenCAPI [89], CCIX [90], and Gen-Z [91] introduced mechanisms to expose remote or shared memory across heterogeneous processors via high-speed links. While each offered unique design trade-offs—ranging from protocol flexibility to cache coherence semantics—their concepts converged into the now-dominant Compute Express Link (CXL) standard [92]. CXL has since emerged as a unifying fabric for coherent communication between CPUs, accelerators, and memory devices, enabling fine-grained memory sharing and composable system architectures. Its industry-wide adoption reflects a growing consensus toward coherent, unified memory and I/O fabrics that can flexibly integrate diverse compute and memory resources [93, 94, 95].

In parallel, major vendors have developed proprietary interconnects tailored to their hardware ecosystems. Examples include AMD’s Infinity Fabric (IF) [96, 97, 98, 99, 100], Intel’s Ultra Path Interconnect (UPI) [101], and NVIDIA’s NVLink and NVSwitch [102], all of which provide low-latency, high-bandwidth communication within multi-socket systems or between GPUs and memory devices. While these interconnects deliver exceptional intra-system performance, they are generally closed, architecture-specific, and limited in

scalability across vendor boundaries. CXL, in contrast, provides an *open and standardized foundation* for composable architectures, bridging the gap between proprietary fabrics and data center-level interconnect standards.

2.2.1 Compute Express Link (CXL)

Among emerging interconnects, CXL is particularly transformative for its ability to provide a unified, cache-coherent interface for both memory and I/O devices. Its layered protocol structure—comprising CXL.io, CXL.cache, and CXL.mem—enables flexible sharing and efficient multiplexing of communication across the same physical link. Its flexible design allows heterogeneous components to share a common memory fabric, thereby enabling composable and extensible memory systems. CXL devices are broadly categorized into three types:

- **Type-1** devices provide accelerators with coherent access to host memory.
- **Type-2** devices integrate local memory that remains coherent with the host memory.
- **Type-3** devices function as passive memory expanders, exposing additional memory capacity directly to the host processor.

Among these, Type-3 devices are particularly relevant for large-scale AI and memory-bound workloads, as they allow DRAM expansion through the CXL.mem protocol, exposing cacheline-granular access to remote memory over a high-speed serial link such as PCIe.

CXL offers substantially higher per-pin bandwidth compared to traditional DDR interfaces. Leveraging efficient serial signaling, CXL links can achieve up to four times higher per-pin bandwidth than DDR [54]. Moreover, CXL supports full-duplex communication, allowing simultaneous bidirectional data transfers. For instance, while a DDR5-4800 channel provides approximately 38.4 GB/s of aggregate read/write bandwidth, a CXL 3.0 ×8

link can concurrently transmit and receive up to 32 GB/s in each direction. This bidirectional capability enables dynamic repurposing of underutilized bandwidth, a feature particularly beneficial for asymmetric workloads where read and write intensities vary over time. While the serial interface enables high bandwidth, it also introduces additional latency due to multiple SerDes conversions along the critical path. Reported round-trip times for CXL-attached memory range from 50 ns to over 100 ns [11, 54]. Although this is higher than local DRAM access (~ 50 ns), it is often comparable to end-to-end access latencies on many-core processors (e.g., ~ 130 ns on a 128-core AMD Bergamo) once queuing delays are considered. Consequently, CXL-attached memory is best utilized as a secondary or parallel memory tier rather than as a direct DRAM replacement.

To fully leverage CXL as a parallel extension to main memory, workload- and hardware-aware co-design is essential. Effective utilization requires coordinated strategies such as:

- Intelligent page placement and migration to minimize high-latency remote accesses.
- Caching and prefetching mechanisms that overlap computation with data movement.
- System software support for dynamic memory allocation, bandwidth scheduling, and topology-aware orchestration to avoid contention.
- Opportunistic reuse of idle I/O bandwidth to improve effective memory throughput.

When combined, these strategies enable memory systems that deliver the capacity and bandwidth scalability required by large-scale, memory-intensive workloads while mitigating latency overheads.

From a broader perspective, CXL represents a fundamental architectural inflection point—transforming rigid, socket-bound memory subsystems into flexible, composable memory fabrics. This evolution paves the way for holistic memory-hierarchy co-design, where architectural mechanisms and system software are jointly optimized to align with workload characteristics and hardware capabilities across chip, server, and cluster scales.

CHAPTER 3

ENHANCING ON-CHIP HTM CAPACITY THROUGH SAFETY HINTS

Hardware Transactional Memory (HTM) represents a high-performance realization of the transactional memory abstraction, offering programmers an intuitive mechanism to express atomic regions of code while delegating synchronization complexity to hardware. By enabling optimistic concurrency control with hardware-enforced atomicity, HTM simplifies parallel programming and reduces the reliance on coarse-grained locking.

Despite its promise, the practical adoption of HTM in commercial processors remains constrained by strict hardware limitations and rigid transactional semantics. Most notable among these limitations is the restricted size of supported transactions, which is implicitly bounded by the hardware’s buffering capacity. Once a transaction’s read or write set exceeds this capacity, the transaction must abort and fall back to a software-based locking mechanism. This fallback path—often accompanied by thread serialization and synchronization overhead—can significantly degrade performance, especially in workloads with large or data-intensive critical sections.

As discussed in Chapter 2.1.1, such capacity limitations constitute a fundamental barrier to scaling transactional execution on modern processors. The source of this pressure lies partly in the programming model itself: the conventional HTM interface treats every memory access within a programmer-defined transactional region (delimited by `begin` and `end` instructions) as equally important to transactional correctness. In reality, many of these accesses are benign—operating on thread-private or read-only data that cannot cause conflicts—yet they still consume transactional tracking capacity. Consequently, hardware resources are often overutilized, leading to premature capacity aborts and reduced throughput.

This chapter presents `HinTM`, a lightweight compiler–runtime–hardware co-design that

mitigates transactional capacity pressure by identifying and excluding non-conflicting memory accesses from transactional tracking. By leveraging safety hints derived through static and dynamic memory access classification, HinTM effectively expands the usable on-chip transactional capacity without modifying cache hierarchies or coherence mechanisms. The subsequent sections describe the design principles, implementation methodology, and detailed evaluation of HinTM, demonstrating how this approach enhances HTM performance across diverse workloads while maintaining architectural simplicity and correctness.

3.1 Transactional capacity expansion through memory access classification

The limited transactional capacity of HTMs largely stems from their conservative tracking semantics. In conventional HTMs, once a transactional region is delimited by the programmer, *every* memory access within it is tracked by hardware structures that enforce atomicity. This implicit, all-inclusive tracking consumes scarce on-chip metadata resources—even for memory locations that cannot participate in data races—leading to frequent capacity overflows and transaction aborts.

We observe that a large fraction of memory accesses within transactions operate on data that are either thread-private or shared read-only across threads. Such accesses are safe from a concurrency-control standpoint, as they cannot violate transactional atomicity. Distinguishing these safe memory locations from unsafe ones allows the hardware to track only those accesses that may lead to conflicts, thereby reducing transactional state pressure and effectively expanding the usable capacity of existing HTM designs.

To evaluate the prevalence of such safe accesses in real workloads, we perform an empirical analysis across a range of scientific and parallel transactional benchmarks. Specifically, we identify the fraction of memory regions that exhibit no read–write sharing across threads throughout execution. A region (page or cache block) is deemed safe if no inter-thread read–write or write–write sharing occurs on it during the program’s runtime. We further measure the proportion of transactional read accesses targeting these safe regions to

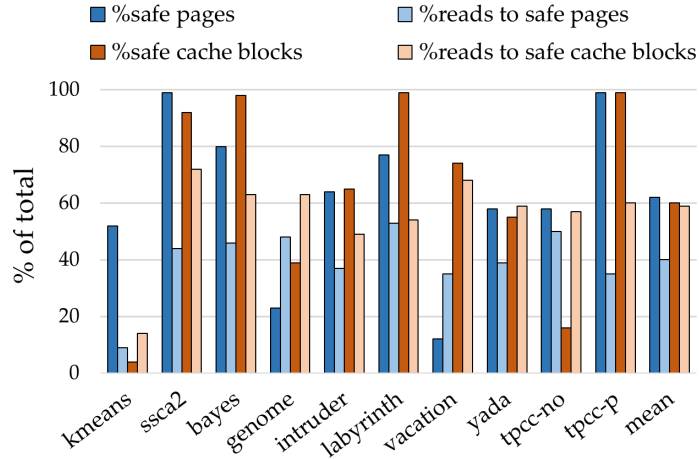


Figure 3.1: Opportunity study for memory access classification.

understand how much of the transactional footprint can be safely excluded from tracking.

Figure 3.1 presents this analysis for representative benchmarks running with eight threads (four for `genome` and `yada` due to limited scalability) on a POWER8-like HTM configuration (methodology in Chapter 3.3). Across applications, a significant subset of accessed pages—on average 62%—are safe, while 40% of transactional reads occur on these pages. When examined at cache-block granularity, the proportion of safe read accesses rises to nearly 60% on average.

These findings highlight substantial potential for capacity optimization through memory access classification. While the reduction in tracked accesses may not directly translate to a proportional decrease in aborts, it strongly suggests that selective tracking can alleviate capacity bottlenecks and improve transactional throughput. Importantly, the results show that even coarse-grained (page-level) classification captures most of this opportunity, indicating that practical implementations with modest hardware and software support can achieve meaningful benefits.

In summary, distinguishing *safe* from *unsafe* memory accesses provides a low-overhead means of tailoring transactional tracking to actual concurrency needs, reducing on-chip resource pressure without altering fundamental HTM semantics. The next sections introduce HinTM, a lightweight compiler–runtime–hardware co-design that operationalizes this prin-

principle through compiler-generated safety hints and runtime-assisted adaptive tracking mechanisms.

3.2 HinTM design and implementation

We propose HinTM (Hinted HTM), a lightweight enhancement to conventional hardware transactional memory systems that augments existing designs with compiler- and runtime-assisted safety hints. These hints enable the hardware to distinguish between *safe* and *unsafe* memory accesses within a transaction (TXN), thereby reducing unnecessary tracking and alleviating transactional capacity pressure.

HinTM builds upon a baseline HTM architecture that maintains transactional state either through dedicated hardware buffers or metadata stored within the data cache. With minimal hardware extensions, HinTM relaxes the capacity constraints of bounded HTMs by tailoring tracking behavior to actual concurrency requirements. The design comprises three key components: (i) a static compile-time mechanism for fine-grained memory access classification, (ii) a dynamic runtime mechanism for adaptive, coarse-grained classification, and (iii) a safety hint propagation interface that conveys compiler and runtime annotations to the underlying HTM controller.

3.2.1 Defining Safe and Unsafe Memory Accesses

From a transactional standpoint, a *memory location* is considered *safe* if it satisfies one of the following conditions:

- (i) it is accessed exclusively by a single thread during the transaction’s lifetime; or
- (ii) it is shared across threads but remains read-only throughout the transaction.

A *memory access* is safe if it targets a safe memory location and does not introduce side effects that could leave the program’s state inconsistent upon a transaction abort.

Under these definitions, all load operations to safe memory locations are inherently safe, as they do not modify program state and cannot cause conflicts. The safety of store operations, however, requires more careful consideration. Even writes to thread-private memory can leave the system in an inconsistent state if a transaction aborts without restoring the pre-transaction value. Consequently, a store is deemed safe only if it is *initializing*—that is, it represents the first access to a memory location within a transaction and does not depend on prior transactional state.

Common examples of such initializing stores include:

- stack-allocated local variables initialized within a transaction for private use,
- newly created objects not yet inserted into shared data structures, and
- arguments written to the stack prior to a function call.

These initializing stores maintain transactional correctness even under abort and retry conditions, as their effects are confined to thread-private state until explicitly published to shared memory.

3.2.2 HinTM’s Classification and Integration Mechanisms

By differentiating between safe and unsafe memory accesses, HinTM enables selective transactional tracking that aligns hardware resource utilization with concurrency behavior. This selective tracking reduces redundant metadata entries in transactional buffers and effectively expands the usable capacity of on-chip HTM structures—achieving capacity gains without altering cache organization or coherence protocols.

To operationalize this concept, HinTM employs two complementary memory access classification mechanisms:

1. **Fine-grained static classification**¹: a compiler-based analysis that identifies provably safe loads and stores at compile time using data-flow and alias analysis, and

¹The static classification component of HinTM was implemented by **Anirudh Jain** using the LLVM compiler framework. For a detailed description of the compiler passes, analysis techniques, and instruction encoding mechanisms, refer to our published work [33].

2. **Coarse-grained dynamic classification:** a runtime mechanism that adaptively identifies and annotates safe memory regions based on access patterns observed during program execution.

Together, these mechanisms provide the underlying hardware with *safety hints* that guide the transactional controller to exclude safe accesses from tracking. The following sections describe each mechanism in detail, illustrating how the cooperation between compiler annotations and runtime feedback enables HinTM to expand effective transactional capacity and improve performance across diverse workloads.

3.2.3 Fine-grained static memory access classification

The static component of HinTM’s memory access classification framework performs compile-time analysis to identify and annotate memory operations that are provably safe. This analysis operates at the instruction level, marking loads and stores that access memory locations determined to be thread-private or shared read-only. Typical examples include stack-allocated local variables, function parameters passed by value, and loads to global constants or read-only data regions.

A memory operation is classified as safe only when the compiler can conclusively determine that all memory locations accessed by the instruction are safe. If a memory instruction references locations whose safety cannot be statically inferred, or potentially spans both safe and unsafe regions, it is conservatively classified as unsafe. This conservative design guarantees correctness while maximizing the subset of instructions that can be safely excluded from transactional tracking.

HinTM implements this functionality through a series of compiler passes built on the LLVM framework [103]. These passes perform static data-flow and escape analysis to determine thread-private regions and read-only data, marking the corresponding memory operations as safe. Specifically, a memory operation within a transaction is annotated as safe if it satisfies one of the following criteria:

- **Safe loads:** The instruction performs a load from either a thread-private memory location or a shared region that remains read-only during program execution. Thread-privacy is verified through escape analysis, which ensures that no other thread can reference the accessed object.
- **Safe stores:** The instruction performs an *initializing store* to a thread-private memory location—that is, the store defines the memory location before it is subsequently used within the transaction. Such stores do not affect shared state and thus maintain correctness even if the transaction aborts.

The compiler annotates the identified safe memory operations directly in the generated binary using lightweight Instruction Set Architecture (ISA) extensions. Each load and store instruction is augmented with a *safety flag* that encodes whether the access is safe or unsafe. During execution, the processor decodes this flag and conveys the information to the HTM controller, which then tracks only the memory accesses performed by unsafe instructions. This selective tracking significantly reduces metadata pressure in transactional buffers, effectively expanding the usable transactional capacity of on-chip hardware structures without modifying cache or coherence organization.

3.2.4 ISA Extensions for Safety Information Propagation

To communicate compiler-generated safety annotations to hardware, HinTM introduces minimal ISA-level support. In our MIPS-based implementation, two unused opcodes were repurposed to define the instructions `load_word_safe` and `store_word_safe`. These instructions are functionally identical to their standard counterparts in terms of pipeline behavior, memory access latency, and microarchitectural optimizations such as load coalescing and merging. The only difference lies in their interaction with the HTM controller: memory locations accessed by safe instructions are not recorded in the transactional tracking structures. The compiler automatically transforms qualifying instructions into their “safe” variants, making the process entirely transparent to the programmer.

Although introducing new opcodes might appear costly, modern ISAs already support mechanisms for direct communication between the CPU and hardware co-processors (e.g., ARM’s co-processor interface [104]). By conceptually treating the HTM subsystem as a co-processor, these existing communication pathways can be repurposed to convey safety hints efficiently, avoiding major hardware modifications.

3.2.5 Coarse-grained dynamic memory access classification

While static memory access classification is accurate, it is inherently conservative, as the compiler lacks visibility into the dynamic sequence of memory accesses issued by concurrently executing threads. For instance, a heap-allocated dataset that is globally visible but partitioned across threads may appear shared at compile time, even though no true inter-thread sharing occurs in practice. To complement the conservative static classification, HinTM introduces a secondary, runtime-based mechanism that dynamically identifies safe memory regions by observing access patterns during execution.

Rationale and Overview. The dynamic memory access classification mechanism extends the address translation process to monitor inter-thread page-level sharing patterns at runtime. By doing so, it enables the hardware to classify certain memory reads as safe, thereby excluding them from transactional tracking within the HTM controller. A page is deemed safe if it satisfies one of the following criteria:

- (i) It is exclusively accessed by a single thread (i.e., thread-private); or
- (ii) It is shared across threads but remains read-only.

Unlike static classification, the dynamic mechanism never marks write operations as safe, as determining whether a write is truly *initializing* at runtime is difficult. Moreover, dynamic classification does not apply to instructions already marked as safe by the compiler (Chapter 3.2.3). Instead, it focuses on capturing runtime sharing behavior that the compiler cannot infer statically.

Mechanism and Implementation. To detect inter-thread sharing, we extend conventional address translation structures—specifically, the page table (Page Table (PT)) and the data Translation Lookaside Buffer (TLB)—to maintain per-page sharing metadata. Each page table entry is augmented with three additional fields:

- (i) a **Thread ID** (*tid*) field, which records the first thread to access the page;
- (ii) a **Read-only** (*ro*) bit, indicating whether the page has been accessed only by reads;
and
- (iii) a **Shared** (*shared*) bit, indicating whether multiple threads have accessed the page.

These bits are also replicated in the data TLB entries, allowing fast access to a page’s safety status without repeated page table walks. The dynamic classification logic consults these bits during address translation to determine whether a page qualifies as safe.

If a translated address belongs to a page marked $\langle private, * \rangle$ or $\langle shared, ro \rangle$, the memory read is considered safe. The processor then marks the access accordingly and informs the HTM controller to skip tracking the address, similar to compiler-annotated safe instructions. This selective exclusion of reads from safe pages alleviates transactional metadata pressure and extends the effective capacity of the underlying HTM structures.

Page State Transitions. Figure 3.2 illustrates the page state transition diagram as multiple threads access a page over time. When a thread first accesses a page, the page table entry is initialized with that thread’s ID and access mode (*ro* or *rw*). All reads from a $\langle private, * \rangle$ or $\langle shared, ro \rangle$ page are safe.

When a second thread accesses the same page:

- If its access is a read and the page is in $\langle private, ro \rangle$ state, the page transitions to $\langle shared, ro \rangle$, and all subsequent reads to this page remain safe.
- If a thread later attempts to write to a $\langle shared, ro \rangle$ page, the page transitions to $\langle shared, rw \rangle$, invalidating its prior safety status.

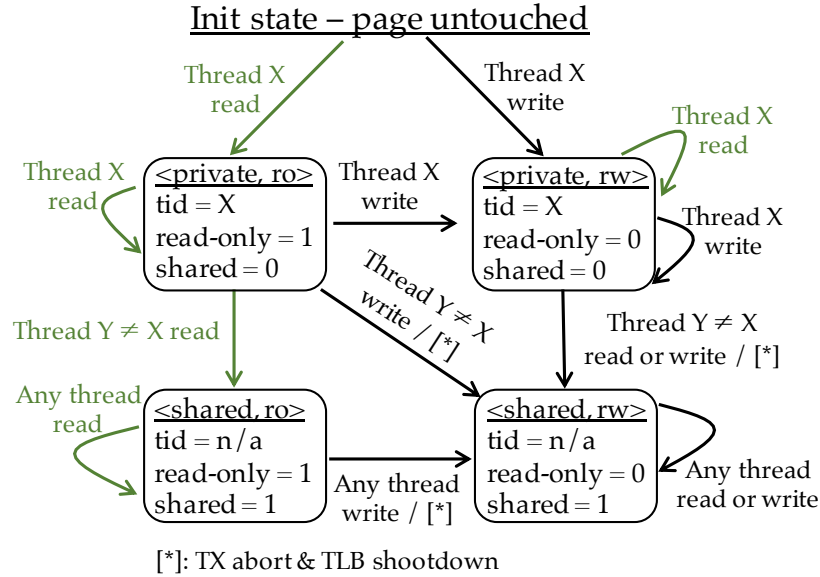


Figure 3.2: State transition diagram for a page’s lifetime. **Green** memory accesses are dynamically marked safe.

Because HinTM’s runtime mechanism does not track memory within pages at finer granularity, such a state transition requires conservatively aborting any active transactions that might have accessed the page assuming it was safe. These aborts, termed *page-mode aborts*, prevent potential atomicity violations that could otherwise occur when a previously safe page becomes writable.

To maintain coherence of page-state metadata, the initiating thread triggers a TLB shutdown [105] across all cores that have cached translations for the transitioning page. The initiator core (i) aborts its own transaction if necessary, (ii) sends inter-processor interrupts (IPIs) to the affected slave cores, (iii) waits for acknowledgments after TLB invalidation, and (iv) updates the global page table entry to reflect the new $\langle shared, rw \rangle$ state. This ensures that the updated unsafe state is visible to all threads in subsequent accesses

Distinction from Page-Based TMs. It is important to distinguish HinTM’s dynamic classification from page-based transactional memory (TM) systems such as IBM’s 801 [106], PTM [40], and XTM [107], which track transactional state directly at page granularity. In contrast, HinTM does not use page-level tracking to manage transactional state. Instead, it

Table 3.1: Simulation parameters.

CPU	8 OoO cores, 2GHz, 176-entry ROB 4-wide dispatch/retirement, MIPS ISA
L1 Cache	32KB 8-way L1d/L1i (split) 64B blocks, 3-cycle latency
L2 Cache	shared non-inclusive, 8MB, 16-way 64B blocks, 12-cycle latency
Coherence	Snoopy MESI
Memory	100-cycle latency

leverages page-level metadata purely as a filtering mechanism—informing the cache-block-granular HTM controller whether specific pages can be safely excluded from tracking. This design avoids the complexity and scalability challenges of coarse-grained page-based TMs while retaining the precision of fine-grained hardware tracking.

3.3 Evaluation methodology

This section details the simulation framework, hardware configurations, and workloads used to evaluate HinTM’s impact on transactional capacity and performance

System Organization. We employ the SESC cycle-accurate simulator [108] to model conventional HTMs with eager conflict detection [43]. The simulated system is an 8-core SMP configured as shown in Table 3.1. Cache latencies are derived from CACTI 7 [109] at 22nm technology.

Compiler. We extend LLVM 9.0.1 [110] with static analysis passes targeting the MIPS backend to align with the simulator’s ISA.

We evaluated four baseline HTM configurations:

- **Dedicated transactional buffering:** Modeled after IBM’s POWER8 HTM implementation [42], this configuration provisions an external 64-entry fully associative

buffer coupled with the L1 data cache to track cache blocks that belong to the running transaction's readset and writeset. We refer to this HTM configuration as `P8`.

- **Hardware signatures:** We extend the `P8` baseline with hardware signatures, modeled after the state-of-the-art PBX hashing with a 1kb bitvector [111]. Signatures prevent capacity aborts when a TXN's readset spills from the transactional buffer. We refer to this configuration as `P8S`. `P8S` increases readset but not writeset capacity, and introduces the possibility of false conflict aborts due to aliasing.
- **In-L1-cache transactional buffering:** Instead of provisioning dedicated buffers, a transaction's state is tracked in the L1 data cache, offering larger transactional capacity than `P8`. We refer to this configuration as `L1TM`.
- **Infinite buffering:** This ideal—from a capacity perspective—HTM configuration never aborts due to capacity overflows, but otherwise remains functionally identical to the previous two HTMs in terms of detecting and reacting to conflict aborts, etc. We refer to this configuration as `InfCap` and use it as an upper bound for the maximum gains achievable by completely eliminating capacity aborts. Figure 3.1's fraction of runtime wasted on capacity aborts is derived as a comparison between `InfCap` and `P8`.

We extend the `P8`, `P8S`, and `L1TM` baselines with `HinTM`'s memory classification techniques and hardware extensions, introducing three additional configurations for each of them:

- **`HinTM-st`** employs static memory classification (Chapter 3.2.3).
- **`HinTM-dyn`** employs dynamic memory classification (Chapter 3.2.5). Safe to unsafe page mode transitions incur a TXN abort and TLB shutdown, which involves Operating System (OS) handler executions and IPI latencies. We model a cost of 6600 and 1450 cycles for the initiator and involved slave cores, respectively, derived

from an extensive TLB shutdown cost study [105]. $\langle private, ro \rangle$ to $\langle private, rw \rangle$ page mode transitions incur a 1450-cycle minor page fault cost [112].

- **HinTM** employs both static and dynamic memory classification (HinTM-st + HinTM-dyn).

Evaluated Workloads. Similar to recent HTM work [113, 50, 114, 43], we evaluate HinTM using the STAMP transactional benchmark suite [115], as well as TPCC’s two most prevalent queries: `new_order` (named *tpcc-no*) and `payment` (named *tpcc-p*) [116], [117]. We deploy `genome` and `yada` on four threads because we observed poor scalability for higher thread counts. Every other application runs on eight threads.

3.3.1 Rationale for Capacity-Constrained Evaluation

We base our study on HTM implementations with relatively small transactional capacity for practical reasons. Evaluating systems with larger transactional capacity requires proportionally larger application inputs to create meaningful capacity pressure, leading to impractically long simulation times. This constraint is well recognized in prior work, as transactional benchmark suites often provide reduced input sizes for simulation-based studies [115].

While HTMs with greater buffering capacity experience fewer capacity aborts, even commercial implementations with larger transactional windows — such as Intel’s HTM — continue to exhibit non-trivial performance overheads due to capacity limitations [51]. Evaluating these limitations using existing transactional benchmarks presents a fundamental “chicken-and-egg” problem: benchmark designers tune transactional regions to fit within the capacity of current hardware, which masks potential scalability bottlenecks. Increasing HTM capacity would undoubtedly reduce abort rates for these workloads, but it would also enable new applications or larger critical sections that are not represented in current benchmark suites.

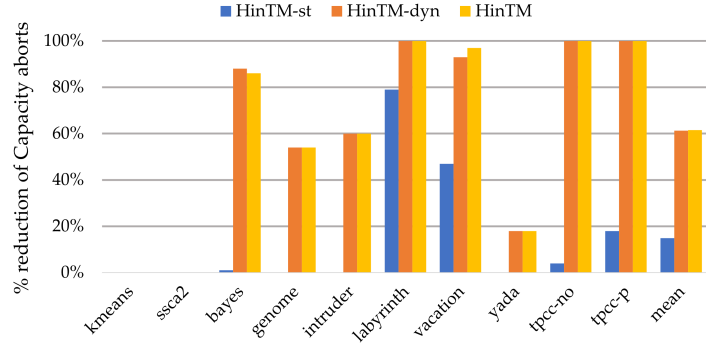
Importantly, the workloads we evaluate are not artificially dominated by capacity constraints. For example, in the `TPCC-p` benchmark, approximately 85% of all transaction aborts are conflict-driven—both with and without HinTM—yet mitigating the remaining capacity-related aborts still results in a 16% performance improvement. This demonstrates that HinTM can deliver tangible benefits even when capacity aborts are not the dominant source of transactional failure.

In summary, merely increasing the hardware’s transactional buffering capacity only postpones the onset of capacity limitations; it does not eliminate them. HinTM provides a complementary path to achieving similar benefits by reducing unnecessary tracking pressure through software-guided safety hints. Our evaluation quantifies these benefits across several representative HTM designs with varying capacity characteristics. The `P8S` configuration highlights HinTM’s impact on HTMs with asymmetric readset and writeset capacities—reflecting the design traits of some commercial systems, such as Intel’s—while `L1TM` illustrates its effectiveness on configurations with larger transactional capacity than `P8`.

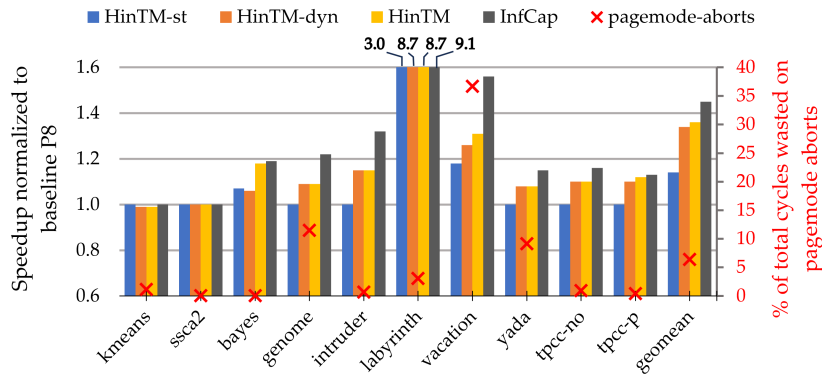
3.4 Capacity Abort Reduction and Speedup with `P8` HTM

Figure 3.3a illustrates the achieved capacity abort reduction for each of HinTM-st, HinTM-dyn, and HinTM as compared to the baseline `P8` HTM configuration. Figure 3.3b shows how Figure 3.3a’s capacity abort reductions reflect into performance improvement, and also includes the performance effect of the hypothetical `InfCap`, which eliminates *all* capacity aborts.

As evidenced in Figure 3.3a, static classification (HinTM-st) alone in most cases is not sufficient to reduce capacity aborts and, by extension, improve performance. As we later demonstrate, while static classification does identify safe accesses, they are not enough to reduce a TXN’s size enough to prevent a capacity abort from occurring. There are two notable exceptions: `labyrinth` and `vacation`, for which HinTM-st prevents $\sim 80\%$ and



(a) Capacity abort reduction.



(b) Speedup over baseline P8.

Figure 3.3: Performance impact of HinTM on the P8 HTM configuration.

~48% of capacity aborts, which results in $2.98\times$ and $1.18\times$ speedup, respectively. The resulting speedup is not a direct function of the achieved reduction in capacity aborts: while HinTM-st prevents some TXNs from capacity-aborting by implicitly increasing the underlying P8’s effective transactional capacity, these same TXNs may end up aborting later due to other reasons (e.g., a conflict). InfCap’s speedup in Figure 3.3b is indicative of each application’s potential benefit from capacity abort reduction. For instance, InfCap demonstrates that labyrinth’s improvement potential is significantly higher than vacation’s ($9.1\times$ versus $1.6\times$).

HinTM-dyn is more effective than HinTM-st, eliminating 61% of capacity aborts on average. This drastic reduction yields an average speedup of $1.34\times$ ($1.45\times$ if applications with zero capacity aborts—kmeans and tcca2—are disregarded), with labyrinth achieving

the highest speedup of $8.7\times$. Because HinTM-dyn as a standalone mechanism is much more effective than HinTM-st, the achieved capacity abort reductions of HinTM (which is a combination of dyn and st) are mostly subsumed by HinTM-dyn. As a result, HinTM's speedup is marginally higher than HinTM-dyn's (2% higher on average). Overall, HinTM achieves an average speedup of $1.36\times$ over baseline P8 ($1.14\times$ excluding the extreme cases of labyrinth, kmeans, and scca2), and is within 6% of InfCap's performance. Importantly, for applications like scca2 and kmeans that exclusively employ very small TXNs and thus never trigger a capacity abort, HinTM has a performance-neutral effect: naturally, HinTM cannot improve performance, but is not detrimental either. HinTM-st has no effect, while the page mode transition aborts introduced by HinTM-dyn do not incur a noticeable slowdown. This marginal slowdown is avoidable by proactively disabling HinTM's mechanisms for applications that only use tiny TXNs.

3.4.1 Effect of Page Mode Aborts

Figure 3.3b's secondary y axis shows the cost incurred by our newly introduced page mode aborts, quantified as the aggregate cycle count spent across cores on page mode abort actions (including initiator/slave core overheads as per Chapter 3.3) divided by the total number of cycles spent for the application's execution. The net cost of page mode aborts depends on two factors: the frequency of such aborts and the resulting cost per abort. The former is a function of the fraction of safe pages and the application's total runtime; the longer the runtime, the better page mode transition costs are amortized, as each page may transition at most once in our HinTM implementation. The latter is a function of the number of threads affected by the ensuing TLB shutdown, and the amount of TXN work that ends up being lost upon such shutdown.

Figure 3.3b shows that the fraction of cycles spent on page mode transitions is modest, with vacation as the only outlier because of a combination of three factors: vacation exhibits a high fraction of read-write pages ($>85\%$), the highest frequency of page-mode

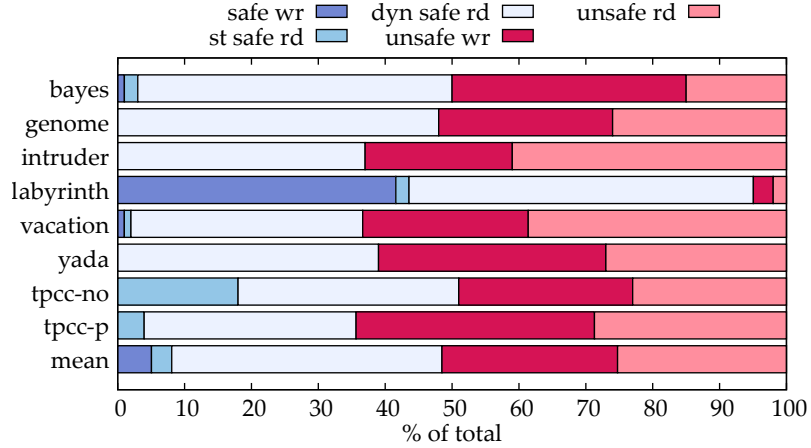


Figure 3.4: Memory access breakdown within transactions.

aborts, and the highest cost (number of execution cycles lost) per page-mode-transition abort. To further study vacation’s outlier behavior and investigate opportunities for further optimization, we conduct an additional experiment with an ideal zero-cost TLB shutdown. Surprisingly, we find that vacation’s achieved speedup only increases by 4%. Despite that vacation on HinTM spends almost 40% of total cycles in page mode aborts, eliminating these costs yields minimal gains, as the increased effective concurrency converts into more conflict aborts that consume most of the achieved overhead savings. Mechanisms for page safety transition prediction or for faster TLB shutdowns (like DiDi [105]) can alleviate page mode transition costs.

3.4.2 Transactional Capacity Pressure Study

We henceforth omit *ssca2* and *kmeans*, which exhibit no capacity aborts, for brevity. To identify HinTM’s source of benefits, Figure 3.4 shows the dynamic breakdown of memory accesses performed within each application’s TXNs by type, distinguishing between compiler- and runtime-annotated safe accesses (collected using HinTM + P8).

The two memory access classification mechanisms combined identify $\sim 50\%$ of the total memory accesses on average as safe. In labyrinth’s extreme case where most memory accesses are to thread-private buffers, our combined classification mechanisms identify

95% of the accesses as safe. The vast majority of safe accesses are identified by the dynamic classification mechanism, indicating why HinTM-dyn is notably more effective than HinTM-st in Figure 3.3.

The fraction of compiler-annotated safe accesses indicates why HinTM-st only achieves small performance benefits. Our static analysis identifies no safe accesses for genome, intruder, and yada, but classifies 18% of tpcc-no’s loads as safe. The best case for static classification is labyrinth, marking 44% of the memory accesses performed within TXNs as safe. This high percentage of statically identified safety is inherent to the application’s structure, where every TXN starts by making a thread-private copy of the grid it operates on to perform optimistic source-destination routing.

Finally, vacation and bayes have a small fraction (3% and 2% respectively) of their transactional memory accesses statically identified as safe.

The fraction of safe accesses alone does not explain the observed capacity abort reduction and performance gains. For example, while HinTM-st identifies just 2% of transactional memory accesses as safe in vacation, it significantly reduces capacity aborts and achieves a sizeable speedup of 18% (Figure 3.3). In contrast, the 18% of loads statically marked as safe in tpcc-no only reduce capacity aborts by 4%, resulting in virtually no speedup. Interestingly, the opposite trend holds for tpcc-p: only 4% of loads are statically marked as safe, allowing HinTM-st to reduce capacity aborts by 18%. This trend is attributed to tpcc-no’s safe loads exhibiting higher spatiotemporal locality. To shed more light into these behaviors, we next analyze each application’s TXN size distribution when HinTM-st and HinTM are used.

Figure 3.5 shows the readset+writeset Cumulative Distribution Function (CDF) of each application’s TXNs, omitting yada and intruder for brevity. We cap the x-axis at 64 cache blocks, matching our modeled P8 configuration’s transactional capacity. TXN with sizes beyond the x-axis range are TXNs that surely abort due to capacity constraints. We collect TXN sizes by running `InfCap` and recording each committed TXN’s readset+writeset size

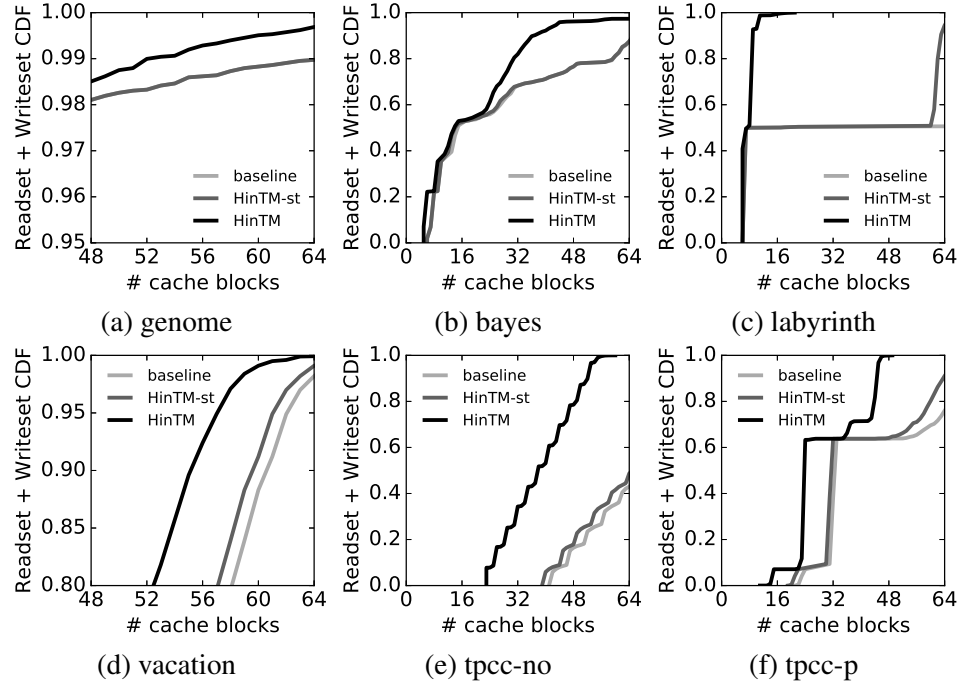


Figure 3.5: Transaction size CDFs for P8 configurations: baseline, HinTM-st, and HinTM. HinTM-st and baseline fully overlap in (a) and (b). Note the different y-axis scales.

as recorded by:

1. **baseline HTM**—i.e., every cache block touched in a TXN.
2. **HinTM-st**—i.e., every cache block touched by a memory operation that is not statically marked as safe.
3. **HinTM**—i.e., every cache block touched by a memory operation that is not marked as safe by either of our two classification mechanisms.

The gap between HinTM’s and baseline’s CDFs shows how memory access classification (primarily the dynamic mechanism) shrinks effective TXN sizes. Specifically the gap between different HTM configurations on the graphs’ far right end indicates the reduced fraction of TXNs exceeding P8’s transactional capacity and explains the capacity abort reductions reported in Figure 3.3a.

To illustrate, Figure 3.5d shows that 2% of vacation’s TXNs exceed P8’s transactional capacity, causing baseline P8 to perform 56% worse than InfCap (Figure 3.3b). HinTM-

st’s safety marking enables half of those TXNs to fit in P8’s transactional buffer, leading to a 47% reduction in capacity aborts, and recouping about half of the performance gap between baseline and InfCap. We hypothesize that the effect is so pronounced, despite only 2% of vacation’s runtime memory accesses being statically identified as safe, because these safe accesses are to unique cache blocks, while unsafe accesses have high spatio-temporal locality in the cache blocks they touch.

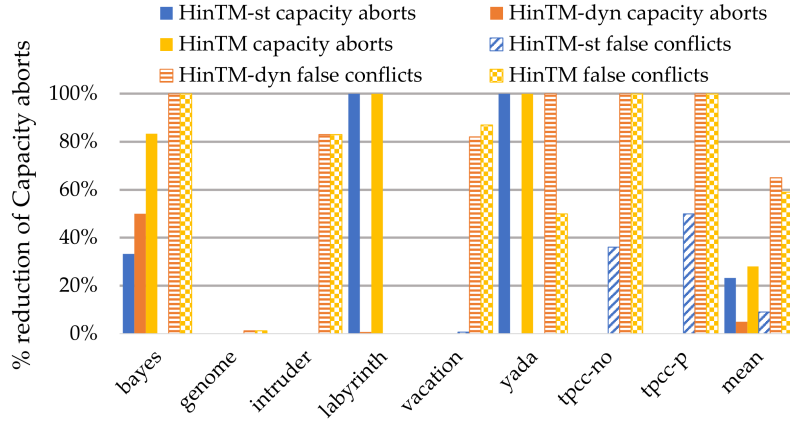
3.4.3 HinTM Effect on Larger HTMs

We now evaluate HinTM’s impact combined with larger HTM baselines. We use Chapter 3.4’s applications with larger inputs to generate sufficient transactional capacity pressure.

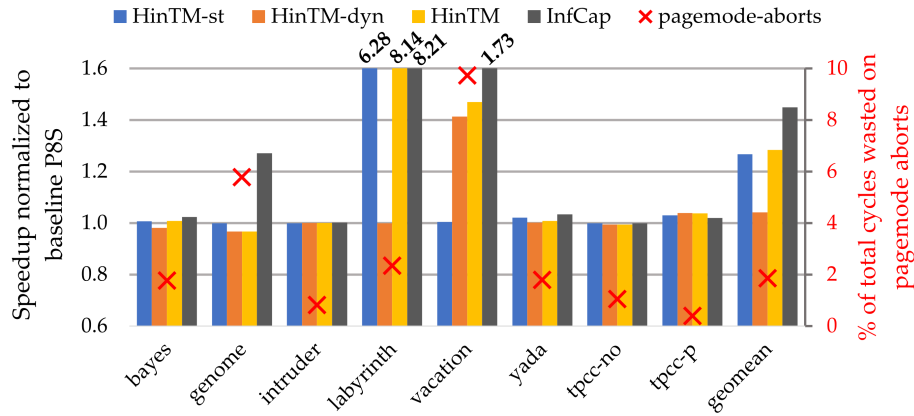
1. P8 with Signatures (P8S). P8S alleviates P8’s capacity pressure by employing signatures to support an unbounded readset. In this case, the opportunity for HinTM’s memory access classification is narrower, with benefits stemming only from false conflict and modest writeset size reduction. Figure 3.6a illustrates the achieved capacity and false conflict abort reduction for HinTM-st, HinTM-dyn, and HinTM as compared to the P8S baseline for a subset of our benchmark applications. The effect of these reductions on performance is shown in Figure 3.6b.

Compared to the P8 configuration, HinTM’s opportunity for capacity abort reduction is limited, because any such reduction must stem from static classification’s identification of safe writes for writeset size reduction, as P8S’s readset is effectively unbounded. HinTM only reduces capacity aborts for bayes, labyrinth and yada.

Due to the application’s nature, HinTM-st identifies a large fraction of labyrinth’s writes as safe, eliminating all capacity aborts. For bayes and yada, the small fraction of writes HinTM identifies as safe results in large relative capacity abort reduction, yet performance gains are minimal, because the absolute number of capacity aborts for both applications is low, accounting for a negligible fraction of their runtime.



(a) Capacity and false conflict abort reduction.



(b) Speedup over baseline P8S.

Figure 3.6: Performance impact of HinTM on the P8S HTM configuration.

Safe read identification does not reduce capacity abortions in P8S, but does mitigate false conflict abortions, which sometimes constitute a significant fraction of the application’s total conflict abortions. For instance, false conflicts cause 39% and 37% of total conflict abortions in vacation and genome, respectively. However, reducing these false conflicts does not always result in performance improvements. For vacation, HinTM’s reduction of false conflicts by 87% improves performance by $1.47\times$ over baseline P8S, while genome’s performance remains unaffected. In tpcc-no’s case, $\sim 2\%$ of TXNs result in false conflicts, which HinTM completely eliminates. However, HinTM’s overhead due to page mode transitions offsets this benefit, resulting in a small net performance loss.

Overall, when combined with P8S, HinTM’s benefit wanes, as signatures eliminate readset capacity constraints. However, HinTM remains beneficial, benefiting most applications modestly and some (like labyrinth and vacation) significantly, for an average speedup of 1.28×.

2. Intel TSX-Style (L1TM). L1TM reduces capacity pressure by tracking read and write sets in the larger (32KB 8-way) private L1 cache. This tracking style can suffer from capacity aborts due to both capacity and set-conflict misses. In order to generate capacity pressure in L1TM while using practical workload sizes in our simulation environment, we employ 2-way SMT on each core. Figure 3.7 shows the performance results. Despite L1TM’s increased transactional capacity, HinTM yields significant performance gains—1.7× on average and up to 7.1×—by reducing capacity aborts by 29–100%. Due to SMT-incurred capacity pressure, HinTM delivers the best performance improvements of all baseline HTM configurations evaluated.

The best gains are achieved for labyrinth, followed by genome. Page mode aborts are, on average, of no concern, with one significant outlier. As indicated by InfCap, transactional capacity expansion holds significant promise for vacation, which however is not fulfilled due to exorbitant page mode abort costs. Vacation’s case motivates investigating improved mechanisms for page-mode classification with reduced page mode transition penalties, as previously mentioned in Chapter 3.2.5.

3.4.4 Evaluation Summary

Our evaluation shows HinTM’s effectively extends a baseline HTM’s limited transactional capacity, improving performance by alleviating capacity aborts. Achieving the same effect solely with hardware requires larger buffering capacity or increased complexity for sophisticated overflow mechanisms. Even when such overflow mechanisms exist (e.g., signatures), HinTM is a beneficial auxiliary mechanism.

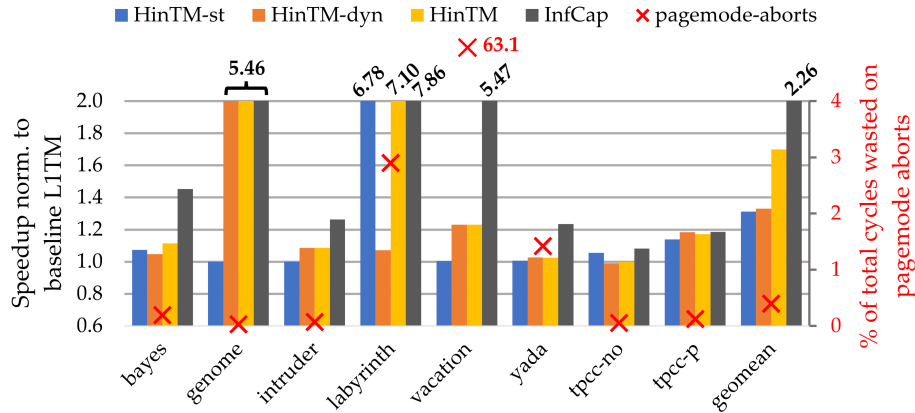


Figure 3.7: HinTM’s impact on L1TM. Speedup and cycles wasted on page mode transitions.

HinTM-st’s average benefits are underwhelming. Although only noticeably boosting few workloads, HinTM-st is worth considering, as its implementation cost is predominantly on the compiler, with minimal hardware requirements. Binary size increase is also modest; our function replication (Chapter 3.2.3) leads to an average and maximum increase of 3% and 5.8%, respectively. We reiterate that advanced compiler techniques can potentially be used to identify more safe accesses than our simple static analysis techniques and improve HinTM-st’s potential. Furthermore, all our evaluated workloads are written in C, resulting in very conservative compiler safety classifications. We expect increased static classification opportunity when using languages with stronger in-built memory safety guarantees (e.g., Rust).

3.5 Chapter Summary

This chapter presented the design, implementation, and detailed evaluation of HinTM, a lightweight extension to conventional hardware transactional memory systems. By combining static and dynamic memory access classification, HinTM reduces the tracking of non-conflicting accesses and effectively expands the usable on-chip transactional capacity. Through systematic evaluation across multiple HTM architectures, we demonstrated that HinTM consistently mitigates capacity-induced aborts and improves transactional perfor-

mance, while maintaining architectural simplicity.

Beyond its quantitative benefits, HinTM illustrates the broader principle of *tailoring memory system behavior to workload and hardware characteristics*. Its co-designed compiler–runtime mechanism exemplifies how semantic information from the software layer can inform hardware-level optimizations, thereby improving concurrency without increasing hardware complexity. The lightweight nature of HinTM makes it applicable to both existing commercial architectures and emerging designs, including future RISC-V processors with planned HTM extensions [118].

More broadly, the insights from this work reinforce the potential of memory system co-design as a pathway to overcome on-chip scalability bottlenecks. The principles demonstrated through HinTM—workload-guided optimization, selective state tracking, and capacity-efficient design—serve as a foundation for higher-level system optimizations explored in the subsequent chapters, which extend these ideas to server- and cluster-scale memory systems.

CHAPTER 4

AUGMENTING SERVER-LEVEL MEMORY BANDWIDTH THROUGH IDLE I/O RESOURCE HARVESTING

As modern datacenter workloads continue to scale in volume and complexity, server-grade processors have evolved toward increasingly many-core architectures to sustain higher computational throughput and greater workload consolidation. However, while compute density has improved steadily with each generation, the growth of off-chip DDR memory bandwidth has not kept pace. Physical pin limitations, packaging constraints, and signal integrity challenges have hindered further expansion of memory channels, leading to a progressive decline in available bandwidth per core. This widening gap between core count and memory provisioning has intensified the long-standing *memory bandwidth wall*, posing a fundamental obstacle to the performance and scalability of bandwidth-intensive server workloads.

Beyond the intrinsic limitations of DDR scaling, the effective utilization of off-chip bandwidth is further constrained by *static resource partitioning*. Each processor socket is provisioned with a fixed number of physical pins that must concurrently support DDR memory channels and high-speed I/O interfaces such as PCIe. At design time, this finite off-chip bandwidth is statically divided between the memory and I/O subsystems based on projected peak utilization ratios. However, such rigid partitioning precludes dynamic reallocation at runtime, often resulting in *bandwidth stranding*—a condition in which the memory subsystem becomes bandwidth-saturated while the I/O subsystem’s allocated capacity remains idle—thereby diminishing overall system efficiency.

As discussed in Chapter 2.1.2, empirical studies of production-scale datacenter infrastructures indicate that a substantial portion of the I/O bandwidth provisioned in modern server-grade processors consistently goes underutilized. Network links and PCIe-attached

storage devices frequently operate far below their designed throughput limits, leaving a significant share of I/O bandwidth idle. Yet, due to the static partitioning of memory and I/O bandwidth, these underutilized I/O resources cannot be repurposed to relieve pressure on the memory subsystem. This persistent mismatch between provisioned and utilized bandwidth exposes an architectural inefficiency and highlights the need for a dynamic, workload-aware resource-sharing mechanism.

To overcome this inefficiency, we introduce SURGE, a software-assisted architectural mechanism that dynamically harvests unused I/O bandwidth to augment available memory bandwidth at runtime. SURGE takes advantage of emerging unified interconnect standards such as Compute Express Link (CXL), which consolidate memory and I/O traffic into a cache-coherent, full-duplex, high-speed serial interface. Leveraging this unification, SURGE opportunistically reallocates idle I/O bandwidth to serve memory transactions, thereby increasing effective memory throughput for workloads constrained by bandwidth contention and queuing delays.

Conceptually, SURGE transforms underutilized I/O bandwidth into a performance-enabling resource, mitigating the memory bandwidth wall while preserving full I/O functionality. It bridges the gap between static hardware provisioning and dynamic workload behavior by aligning bandwidth allocation with the asymmetric utilization patterns typical of cloud and datacenter environments. Through this adaptive approach, SURGE enables a new class of memory system optimizations that enhance overall efficiency, scalability, and performance in modern many-core server architectures.

The remainder of this chapter describes the architectural design and implementation of SURGE in detail, outlining the coordinated hardware–software mechanisms that enable efficient bandwidth harvesting and demonstrating the resulting improvements in memory performance across representative datacenter workloads.

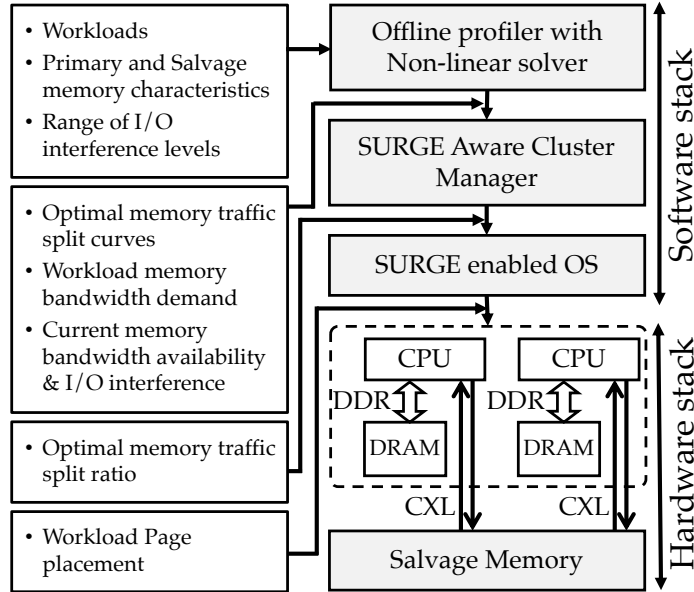


Figure 4.1: SURGE hardware and software components.

4.1 SURGE Design

Allowing dynamic multiplexing of memory and I/O bandwidth resources mitigates the inefficiencies introduced by static design-time partitioning of a processor’s off-chip bandwidth. To this end, we propose SURGE, a software-assisted architectural mechanism that targets memory-bound workloads by dynamically salvaging idle I/O bandwidth to enhance effective memory bandwidth availability.

Figure 4.1 illustrates the high-level organization of SURGE. The mechanism combines lightweight architectural extensions with software coordination to maximize the hardware technique’s utility. On the architectural front (Chapter 4.1.1), the processor is provisioned with additional salvage memory accessible via the I/O interfaces, which are enhanced to support concurrent multiplexing of memory and I/O traffic. On the software side (Chapter 4.2), the operating system and cluster management software coordinate when and how the additional bandwidth resources are activated, ensuring that SURGE’s benefits are realized only when the I/O subsystem is underutilized.

SURGE is not intricately tied to any specific protocol or off-chip interface fabric; its

only fundamental requirement is the capability to dynamically multiplex memory and I/O traffic over the same physical interface. Among current standards, Compute Express Link (CXL) represents a particularly promising substrate for realizing this capability. Accordingly, this work adopts CXL as the reference implementation to provide a quantitative and architectural basis for evaluating SURGE ’s potential.

4.1.1 Architectural Support

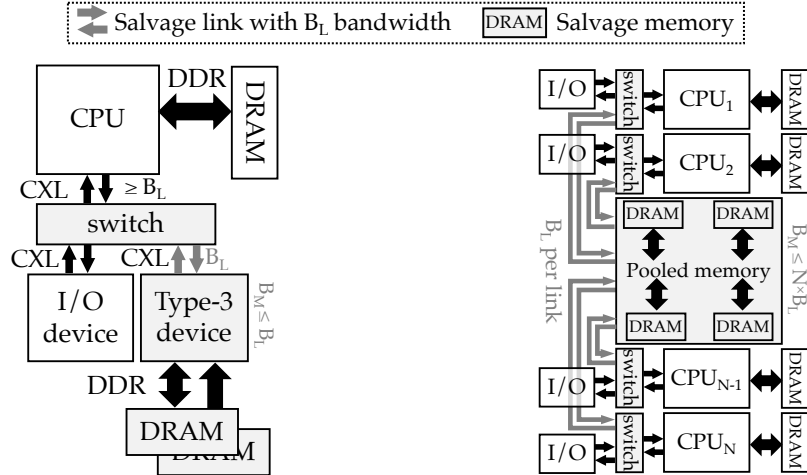
We explore two architectural design instances that realize the SURGE concept: one at the individual server level and another at the cluster (or pod) level. The first design, SURGE Solo, focuses on localized I/O bandwidth salvaging within a single server, while the second, SURGE Pod, extends this concept to enable resource pooling across multiple servers in a tightly coupled cluster. The two approaches are detailed in Chapters 4.1.2 and 4.1.3, followed by a comparative analysis of their resource efficiency in Chapter 4.1.4.

4.1.2 SURGE Solo

Figure 4.2a illustrates the simplest form of SURGE, which targets per-server bandwidth augmentation. Consider a CXL-capable PCIe interface attached to an I/O device, such as a network interface card (NIC). SURGE Solo bifurcates this interface to support both I/O and memory functions: the primary link remains connected to the I/O device, while a secondary link connects to a CXL-attached memory device (a Type-3 device). We refer to these as the *salvage link* and *salvage memory*, respectively.

An on-chip arbiter dynamically multiplexes the traffic between the two devices, steering memory requests to the salvage memory and I/O packets to the connected peripheral device. The arbitration logic ensures that memory traffic is injected only when the I/O interface is underutilized. The operational details of this hardware mechanism are discussed further in Chapter 4.3.1.

The key performance parameters governing SURGE Solo are the bandwidth of the



(a) Solo SURGE. Sharing a CXL interface channel between an I/O device and a CXL Type-3 device as salvage memory. (b) SURGE Pod. Sharing a CXL Type-3 multi-headed memory device as salvage memory across multiple servers.

Figure 4.2: Two SURGE architectural embodiments.

salvage link (B_L), the bandwidth of the salvage memory ($B_M \leq B_L$), and the latency overhead introduced by the I/O interface. These parameters are determined by the underlying interconnect technology. For instance, a 16-lane PCIe 5.0 or CXL link provides approximately 63 GB/s of bandwidth per direction with an added latency of 50–100 ns relative to direct DDR access [54]. The salvage memory’s bandwidth B_M depends on the number and speed of the DDR channels provisioned on the CXL Type 3 device—for example, a single DDR5-4800 channel (38.4 GB/s) or two DDR4-3200 channels can be comfortably supported over a 16-lane CXL link.

4.1.3 SURGE Pod

The SURGE Pod configuration, shown in Figure 4.2b, is an alternative design point leveraging the same principle of opportunistic I/O bandwidth salvaging to a multi-server environment. In this design, the salvage memory is pooled across multiple servers grouped into a logical unit called a pod, as in prior architectures proposed to mitigate memory capacity stranding [119, 11]. The pooled memory is a CXL Type 3 multi-headed device, featuring multiple CXL ports to support direct connections to every server in the pod.

In contrast to SURGE Solo, the pooled configuration enables aggregate provisioning of memory bandwidth (B_M) that exceeds the per-link bandwidth (B_L), allowing multiple servers to simultaneously access the shared salvage memory. This design significantly improves resource utilization, as the probability that at least one server has idle I/O bandwidth increases with pod size. The implications of B_M provisioning and trade-offs in resource efficiency are discussed in Chapter 4.1.4.

4.1.4 Utility Comparison

The two SURGE design variants—SURGE Solo and SURGE Pod—represent distinct points in the design space, differing in complexity, scalability, and overall bandwidth utilization efficiency. While both leverage idle I/O bandwidth to augment memory bandwidth, their effectiveness depends on how consistently the provisioned salvage memory can be used without being stranded behind busy I/O links.

In the SURGE Solo configuration, the salvage memory is directly attached to a single server via one salvage link. This link serves as both the conduit for memory traffic and the arbiter of opportunity: when the I/O subsystem is actively transmitting data, the link is occupied, and the salvage memory becomes inaccessible. Consequently, whenever the I/O interface operates near capacity, the salvage memory remains underutilized. This phenomenon, termed *bandwidth stranding*, directly limits the practical utility of the provisioned salvage memory.

To quantify this behavior, we define utility (U) as the probability that a salvage memory device’s provisioned bandwidth is actively used to serve memory requests. Conversely, the stranding probability ($1 - U$) reflects the likelihood that the salvage memory remains idle due to insufficient available I/O bandwidth. For SURGE Solo, the utility U_{Solo} can be expressed as:

$$U_{\text{Solo}} = P \tag{4.1}$$

where P denotes the probability that a server has sufficient idle I/O bandwidth available for

salvage.

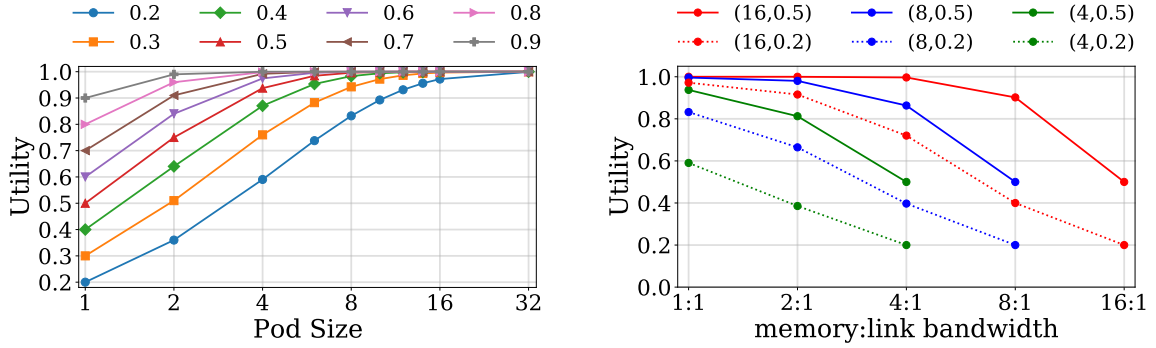
The SURGE Pod design improves upon Solo by pooling salvage memory across multiple servers interconnected within a pod. Each server in the pod independently monitors its I/O utilization and can access the shared salvage memory when its local I/O link is idle. This collaborative model significantly reduces the overall probability of memory stranding, since the likelihood that all servers in the pod simultaneously experience full I/O utilization is much smaller. For a pod comprising N servers, the aggregate utility U_{Pod} is given by:

$$U_{\text{Pod}} = 1 - (1 - P)^N \quad (4.2)$$

This formulation captures the benefit of statistical multiplexing: as N increases, the probability that at least one server can salvage idle I/O bandwidth rises rapidly. For example, with $P = 0.2$ (i.e., each server having idle I/O bandwidth with probability 20%) and $N = 16$, the aggregate utility increases from 20% in SURGE Solo to approximately 97% in SURGE Pod. Figure 4.3a illustrates this trend, demonstrating that larger pods yield substantially higher utilization of provisioned salvage memory.

However, this improvement in utility comes with associated trade-offs. Increasing the pod size introduces (i) greater interconnect complexity and cost, as more servers must maintain coherent connectivity to the shared salvage memory, and (ii) higher access latency, as memory requests may traverse additional network or fabric hops. Prior evaluations of CXL-based memory pooling report an additional 75–100 ns access latency for groups of 8–16 servers [120, 11], which bounds the practical pod size for latency-sensitive workloads.

Designing a cost-effective SURGE system therefore requires balancing the salvage memory’s provisioned bandwidth (B_M) and the salvage link bandwidth (B_L) against expected utility and performance requirements. For a pod with N servers, each connected via a salvage link of bandwidth B_L , the aggregate salvage memory bandwidth B_M can range from B_L (no overprovisioning) up to $N \times B_L$ (fully matched to aggregate link capacity).



(a) Utility for various P values, as a function of pod size. Pod size of 1 is SURGE Solo.

(b) Utility across (pod size, P) combinations as a function of provisioned $B_M:B_L$ ratio.

Figure 4.3: Salvage memory utility for SURGE as a function of pod size and P (probability of each individual salvage link having sufficient idle bandwidth to be salvaged by SURGE).

The optimal provisioning point depends on both P and N , as excessive overprovisioning yields diminishing returns once stranding probability becomes low.

Figure 4.3b plots utility as a function of B_M for different pod sizes and idle-bandwidth probabilities. On the x-axis, a ratio of $X:1$ indicates that $B_M = X \times B_L$. The results show that utility declines more sharply for smaller pods and higher I/O activity (larger P). For example, a 16-node pod with $B_M = 4 \times B_L$ sustains over 80% utility even when $P = 0.2$. In general, maintaining high utility favors moderate salvage memory provisioning—sufficient to absorb temporary bandwidth slack but not so excessive as to leave capacity idle during sustained I/O activity.

In summary, the SURGE architecture demonstrates how dynamic multiplexing of memory and I/O bandwidth can recover otherwise stranded off-chip capacity to alleviate memory bandwidth limitations. The two design variants, SURGE Solo and SURGE Pod, highlight a trade-off between simplicity and scalability. SURGE Solo offers minimal implementation complexity and server-level deployability but achieves limited utility under high I/O activity. In contrast, SURGE Pod leverages memory pooling across multiple servers to improve utilization and amortize cost, at the expense of higher implementation complexity and modest latency overheads. Regardless of configuration, realizing the full benefit of SURGE requires effective runtime coordination to identify idle I/O bandwidth, manage

salvage memory allocation, and balance resource sharing across workloads. These coordination tasks are enabled by software mechanisms, described next in Chapter 4.2.

4.2 SURGE Methodology

4.2.1 Workload Placement by Cluster Manager

The fundamental premise of SURGE is that modern servers frequently exhibit idle I/O bandwidth that can be opportunistically harvested to augment memory bandwidth. However, this condition is not guaranteed across all workloads — it depends on each server’s workload composition and the degree of I/O activity at runtime. Consequently, in a cluster environment, the cluster manager plays a crucial role in orchestrating workload placement to maximize SURGE’s utility and overall system efficiency.

By being SURGE-aware, the cluster manager can infer when a server’s I/O subsystem is likely to remain underutilized and communicate this information to the server’s local OS. The OS can then initiate bandwidth salvaging operations as described in Chapter 4.2.2. In effect, the cluster manager serves as a global coordinator, enabling workload-aware resource allocation and ensuring that SURGE activates memory bandwidth augmentation only when sufficient idle I/O capacity is available.

A SURGE-aware cluster manager also avoids detrimental workload colocations. For example, consider a server currently running workload A , which is known to minimally utilize I/O resources. The cluster manager can colocate a memory-intensive workload B on the same server, anticipating that sufficient I/O bandwidth will remain idle for SURGE to exploit. Once this placement is made, the cluster manager notifies the server’s OS that I/O bandwidth is available for salvaging. If workload A later completes or is replaced by a more I/O-intensive task, the cluster manager must either (i) schedule a new workload with similar I/O characteristics, or (ii) notify the OS about an impending change in available idle I/O bandwidth. Upon receiving such notification, the OS adapts by migrating workload B ’s pages from salvage memory back to primary memory, thereby preserving performance sta-

bility. This synergy between the cluster manager and the OS requires the cluster manager to maintain awareness of the performance characteristics of all active workloads, particularly their memory bandwidth and I/O utilization profiles. Such workload characterization is already a key feature of modern cluster management frameworks, which use profiling or declarative metadata to inform scheduling decisions [121, 122, 123, 124]. SURGE extends this principle to off-chip bandwidth management, allowing system software to make coordinated decisions that align workload behavior with available hardware resources.

4.2.2 Memory Traffic Split Strategy and Analytical Model

A key challenge in a SURGE-enabled server is determining the optimal distribution of memory traffic between the server’s *primary memory* (directly DDR-attached) and the *salvage memory* accessible via the I/O interface. To address this, we develop an analytical model, summarized in Figure 4.4, that determines the traffic split ratio minimizing the overall Average Memory Access Time (AMAT) experienced by a workload.

The model’s objective is to compute the optimal fraction of traffic directed to the primary memory, denoted as:

$$R^* = \arg \min_R \text{AMAT}(R) \quad (4.3)$$

where R represents the fraction of total memory traffic sent to the primary memory, and $(1 - R)$ corresponds to the fraction directed to the salvage memory. The AMAT is expressed as a function of the access latencies of both memory types, the ingress and egress latencies of the salvage link, and the queuing delays that increase with overall bandwidth utilization.

Model Inputs and Profiling. As shown in step ① of Figure 4.4, the model takes as input the *load–latency curves* of the server’s primary and salvage memories, along with the load–latency characteristics of the ingress and egress paths of the I/O interface (e.g., CXL), all profiled directly on real hardware. Each server configuration is characterized by discrete latency measurements collected across a range of utilization levels for the DDR-attached

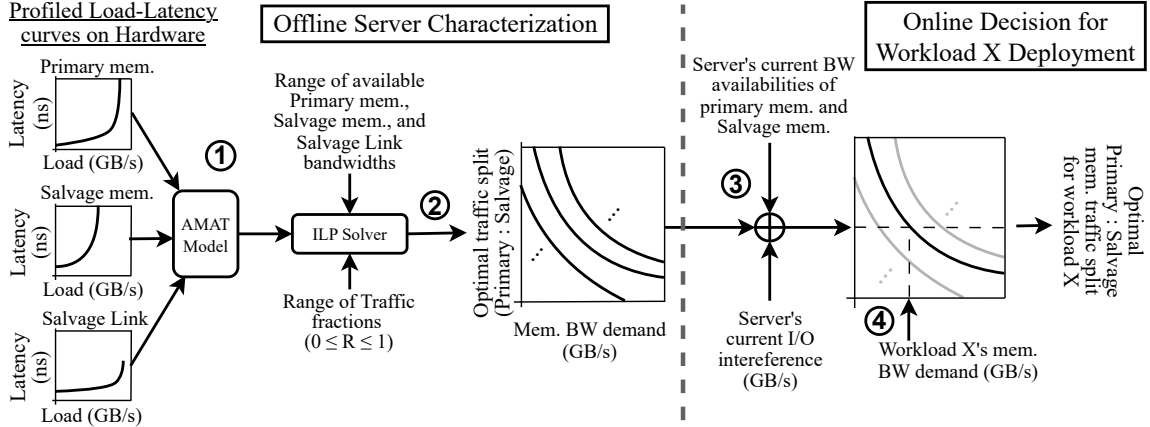


Figure 4.4: Methodology to analytically determine the optimal traffic split between primary and salvage memories.

primary memory, the I/O-based salvage memory, and the I/O interface. These measurements provide a comprehensive view of subsystem behavior under varying bandwidth utilization and serve as empirical input to the analytical model. Prior work has demonstrated that such profiling accurately captures the performance and queuing characteristics of real memory systems [57].

To generalize these discrete measurements into a form suitable for continuous optimization, the data are interpolated using cubic splines to construct smooth functions representing the load-dependent access latencies of the primary and salvage memories. Each interpolated curve preserves the fidelity of the hardware-observed behavior while providing the smooth continuity required for analytical evaluation. Using these interpolated latency curves, the analytical model computes the optimal traffic split ratio R^* , which specifies, for any given workload bandwidth demand D , the proportion of traffic that should be steered to primary memory to minimize the overall AMAT.

Modeling of SURGE’s AMAT. Using the interpolated latency functions derived from offline profiling, we construct a mathematical model that expresses the total AMAT, experienced by a workload using SURGE, as a function of the memory traffic distribution between the primary and salvage memories. This formulation serves as the foundation for

the subsequent Integer Linear Programming (ILP)–based optimization, which determines the traffic split ratio that minimizes the overall access latency for a given workload bandwidth demand D . The model captures the interdependence between bandwidth utilization and access latency across both the memory subsystems and the I/O interface.

For a workload with total bandwidth demand D , a fraction R of the traffic is directed to the primary (DDR-attached) memory, while the remaining $(1 - R)$ is directed to the salvage memory accessed via the I/O interface. The effective utilization of each memory subsystem is therefore defined as:

$$U_P = \frac{R \cdot D}{B_P}, \quad U_S = \frac{(1 - R) \cdot D}{B_S} \quad (4.4)$$

where U_P and U_S denote the effective utilization levels of the primary and salvage memories, respectively, and B_P and B_S denote their corresponding peak sustainable bandwidths.

The total AMAT experienced by the workload for a given traffic split R is expressed as:

$$\text{AMAT}(R) = R \cdot L_P(U_P) + (1 - R) \cdot \left[L_S(U_S) + L_{\text{ing}}(\lambda_{\text{ing}}) + L_{\text{egr}}(\lambda_{\text{egr}}) \right] \quad (4.5)$$

where $L_P(U_p)$ and $L_S(U_s)$ represent the interpolated load-latency functions for the primary and salvage memories, while L_{ing} and L_{egr} capture the latency characteristics of the ingress and egress paths of the I/O interface (e.g., CXL), as a function of the link utilization in each direction (λ_{ing} and λ_{egr}). These latency functions model the queuing behavior observed in the profiled load–latency curves, thereby reflecting realistic contention effects under varying traffic conditions.

The ingress and egress link loads, which determine the corresponding interface latencies, vary with the directions of traffic traversing the link, and are evaluated as:

$$\lambda_{\text{ing}} = \frac{(1 - R) \cdot D \cdot \rho_{\text{rd}}}{\eta} + IO_{\text{ing}}, \quad \lambda_{\text{egr}} = \frac{(1 - R) \cdot D \cdot \rho_{\text{wr}}}{\eta} + IO_{\text{egr}} \quad (4.6)$$

where where IO_{ing} and IO_{egr} represent the ingress/egress I/O activity, $\rho_{rd} = 0.75$ and $\rho_{wr} = 0.25$ correspond to a typical 3 : 1 read-to-write traffic ratio assumed in the system, and $\eta = 0.94$ denotes the effective link efficiency of the CXL I/O interface. Thus, as the fraction of traffic directed to the salvage memory increases, the ingress and egress latencies also rise, capturing the queuing effects and dynamic contention behavior of the interface under higher utilization.

4.2.3 Offline Optimization and Generation of Traffic Split Curves

In the step ② of Figure 4.4, an integer linear programming (ILP)–based optimization is performed to determine the traffic distribution that minimizes the overall AMAT (Equation (4.5)) for a given workload bandwidth demand D . The optimization leverages the interpolated latency functions obtained from hardware profiling to accurately capture the non-linear interactions between bandwidth utilization and access latency across the memory subsystems and the I/O interface.

The ILP solver evaluates a discrete set of candidate traffic split ratios of R (typically ranging from 0.05 to 1.0 in uniform increments) and computes the corresponding AMAT for each configuration using the interpolated latency functions. For each value of R , the solver interpolates the load-dependent latencies for $L_P(U_P)$, $L_S(U_S)$, $L_{ing}(\lambda_{ing})$, and $L_{egr}(\lambda_{egr})$, and then aggregates them according to Equation (4.5). The optimization objective is therefore defined as:

$$R^* = \arg \min_R \text{AMAT}(R) \text{ subject to } 0 \leq R \leq 1 \quad (4.7)$$

where the solver identifies R^* as the *optimal traffic fraction* directed to the primary memory (and correspondingly $(1 - R^*)$ to the salvage memory) that minimizes aggregate access latency for a workload with bandwidth demand D . This formulation captures the latency–bandwidth trade-off between the DDR-attached and I/O-attached memories, along

with the dynamic queuing effects that emerge on the interface under higher utilization.

While the above formulation produces the theoretically optimal traffic split for a single set of system parameters, real datacenter environments exhibit dynamic variations in available memory and interface bandwidths due to workload co-location and fluctuating I/O activity. To ensure SURGE remains effective under such dynamic conditions, the optimization is extended offline across a multidimensional design space that spans:

- available bandwidth fractions of the primary and salvage memories (B_P^{avail} , B_S^{avail}),
- instantaneous utilization levels of the ingress and egress interface paths, and
- representative workload bandwidth demands D .

By repeating this optimization across multiple combinations of primary memory, salvage memory, and interface bandwidth availabilities, the model produces a comprehensive *family of optimal traffic split curves* that collectively capture the best traffic distribution policies under diverse system states. For each configuration within this multidimensional design space, the ILP solver evaluates discrete values of R , computes the corresponding AMAT using Equation (4.5), and identifies the optimal split R^* as function of the workload demand (D), the available bandwidths of the primary and salvage memories (B_P^{avail} , B_S^{avail}), and the load-dependent interface latencies (L_{ing} , L_{egr}).

Finally, the resulting family of optimal traffic split curves is stored as a lookup table indexed by the current availability ratios of primary, salvage, and interface bandwidth. Steps ① and ② are executed offline once per unique server configuration. This precomputation eliminates the need for costly runtime optimization, allowing SURGE to determine near-instantaneous traffic distribution decisions through lightweight lookup operations performed by the operating system and the cluster manager.

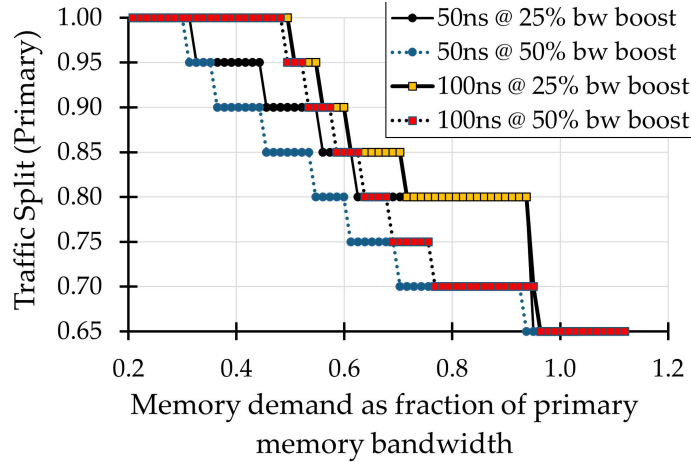


Figure 4.5: Determining traffic split between primary and salvage memory as a function of memory bandwidth demand and salvage memory’s latency/bandwidth characteristics.

4.2.4 Runtime Selection and Application of Optimal Traffic Split

Upon the deployment of a new workload, SURGE determines the optimal primary-to-salvage traffic ratio in a two-stage process. In step ③, the system evaluates the target server’s current utilization of primary and salvage memory bandwidths, along with the prevailing I/O traffic conditions, to select the most appropriate precomputed optimal traffic split curve from the offline-generated lookup table. Each entry in this table maps a workload’s expected bandwidth demand D to its corresponding optimal split fraction R^* , allowing the system to quickly identify the best traffic allocation policy without re-running the optimization computations. Accurately accounting for concurrent I/O activity during this stage is critical, as active interface utilization can constrain the salvage memory’s effective bandwidth and, in turn, influence the optimal split decision. In step ④, the workload’s estimated memory bandwidth demand D is used to probe the selected curve, yielding the precise primary-to-salvage traffic ratio to be applied for that workload under the current system state.

Figure 4.5 illustrates the use of these optimal traffic split curves. Each curve corresponds to a distinct salvage memory configuration, characterized by unique bandwidth and latency parameters. In the plots, salvage bandwidth is expressed relative to primary mem-

ory bandwidth, while salvage latency represents the additional delay beyond the baseline DRAM access latency. For instance, consider a system equipped with DDR5-4800 primary memory offering 38.4 GB/s peak bandwidth and approximately 50 ns DRAM access latency. A salvage memory configuration labeled “50 ns @ 50% boost” represents a device providing an additional 19.2 GB/s of bandwidth (i.e., 50% of the primary memory’s capacity) with an effective total access latency of 100 ns—comprising 50 ns DRAM latency and 50 ns link overhead—when accessed over a serial interconnect such as CXL.

At lower memory bandwidth demands, most traffic is directed to the low-latency primary memory, which can efficiently serve requests without queuing. As demand increases and the primary memory approaches saturation, a progressively larger fraction of traffic is offloaded to the salvage memory, alleviating contention and minimizing the total AMAT. The effect of unloaded latency is most pronounced at lower demand levels, where queuing is minimal. Under high bandwidth demand, however, the system becomes bandwidth-bound, and queuing delays dominate the overall access latency—making the salvage memory’s additional latency relatively less significant. This trend is visible in Figure 4.5: while configurations with higher salvage bandwidth shift more traffic from the primary memory, the optimal split ratios converge at high demand levels regardless of the salvage memory’s latency overhead.

Finally, although the analytical model computes the optimal traffic distribution assuming that traffic can be statelessly distributed between the two memories, the effective realization of this split is governed by data placement policies. In practice, SURGE enforces the computed split through a page-based placement strategy, wherein the operating system distributes physical pages between the primary and salvage memories according to the model-predicted ratio. The detailed implementation of this page placement mechanism is discussed in Chapter 4.3.2.

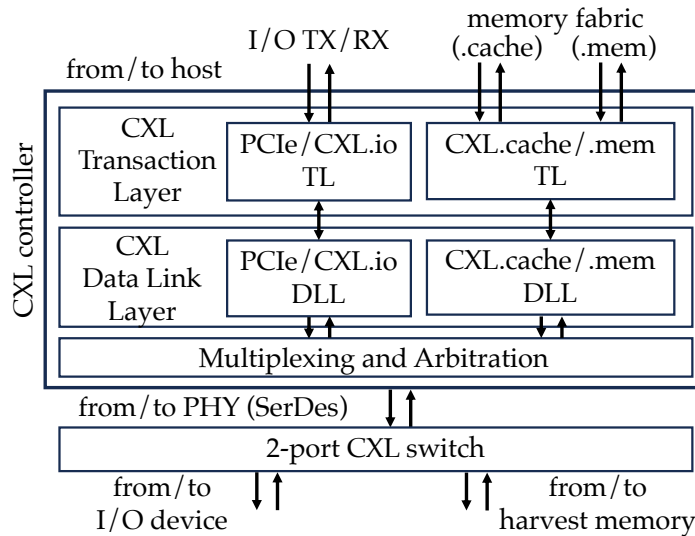


Figure 4.6: I/O and memory traffic multiplexing block diagram.

4.3 SURGE Implementation

4.3.1 Hardware Extensions

As discussed in Chapter 4.1.1, SURGE requires a fabric that supports dynamic multiplexing of I/O and memory traffic. Our implementation focuses on the CXL protocol, as its specification supports such dynamic multiplexing of CXL.io, .mem, and .cache traffic. While existing CXL controller IP blocks (like the one from Rambus [125]) support all three CXL protocols, they are currently designed to operate in a single mode at system initialization time. However, the CXL specification includes the “Flex Bus” feature, which supports dynamic multiplexing of traffic types and even provisions for programmable arbitration policy between them. Flex Bus provides the basic functionality required by SURGE and, to the best of our knowledge, we are the first to leverage this feature for new architectural designs that push the memory bandwidth wall.

Figure 4.6 shows a block diagram of a CXL controller that performs .io and .mem traffic multiplexing, along with a 2-port CXL switch to support traffic bifurcation between the primary I/O device and our added salvage memory device. Given that, in SURGE, the interface “belongs” to the I/O device, we configure the arbiter to always prioritize I/O traffic

over memory. At high I/O activity, such policy can penalize the—generally more latency-sensitive—memory traffic. However, prolonged high I/O activity is outside SURGE’s target use cases.

4.3.2 Software Support

Memory traffic split via page placement. Data placement to achieve the traffic split derived via our analysis in Chapter 4.2.2 can be performed in several ways. For example, a mechanism could perform fine-grained (e.g., per cache block) data distribution between primary and salvage memory. We opt for a coarser-grained, but also much less intrusive, page-based approach that can be seamlessly integrated in any modern OS.

In our SURGE implementation, the OS implicitly achieves memory traffic distribution between primary and salvage memory by allocating pages in the respective memory type at the desired ratio. We assume a first-touch policy; every time the OS allocates a new page for a workload, it selects the primary or salvage memory with a probability R^* or $(1 - R^*)$, respectively, defined by Chapter 4.2.2’s traffic split analysis. We find that such allocation results in the intended traffic split, confirming statistical expectations.

Role of cluster manager. We assume a cluster manager aware of the fleet’s SURGE-enabled servers and at a minimum:

1. Is aware of the expected average memory and I/O bandwidth use of each workload that is about to be deployed, as derived by prior workload profiling.
2. Given the target server’s state and workload to be deployed, determines the best traffic split between primary and salvage memory as per Chapter 4.2.2’s methodology, and indicates that split to the server’s OS.
3. Does not deploy a new I/O-intensive workload on a server where a previously deployed memory-intensive workload with permission to salvage I/O bandwidth is still active.

Advanced software support can further improve SURGE’s utility. For example, more sophisticated page placement policies than first-touch may also explicitly take the CXL salvage link’s directionality into account and place read-/write-heavy pages in salvage memory when the I/O interface has more ingress/egress activity, respectively. Additionally, advanced synergies between the cluster manager and the OS can be developed, to allow dynamic memory allocation changes during a workload’s runtime—for example, the cluster manager can instruct the OS to free up I/O resources by scaling back on salvaging via page migration from salvage to primary memory. We focus on the demonstration of SURGE’s benefits as a novel proof-of-concept architectural technique, leaving more advanced software mechanisms to future work, further discussed in Chapter 7.4.

4.4 SURGE Evaluation

4.4.1 Methodology

Modeled system. We simulate a scaled-down version of a modern manycore server CPU. We consider two exemplary systems of such a processor family. The AMD EPYC 9754 CPU (Bergamo) [126] features 128 cores, 12 DDR5-4800 channels, and 128 PCIe 5 lanes per socket. The Intel Xeon 6780E CPU (Sierra Forest) [127] features 144 cores, 8 DDR5-6400, and 88 PCIe 5 lanes per socket. We model a scaled down CPU with 12 cores that preserves core-to-memory-bandwidth and core-to-LLC-capacity ratios representative of such systems.

Table 4.1 details the parameters used for cycle-level simulation in ZSim [128]. We adopt SURGE Pod as the primary instance of SURGE in most of our evaluation due to its higher utility, as discussed in Chapter 4.1. We assume a SURGE Pod deployment offering 50% additional memory bandwidth over a single server’s primary DDR memory bandwidth at a 100 ns latency premium (at zero load). We evaluate other bandwidth and latency values, including configurations representative of SURGE Solo deployments, as a sensitivity study (Chapter 4.4.6).

Table 4.1: System parameters used for simulation in ZSim.

CPU	12 OoO cores, 2.4GHz, 4-wide, 256-entry ROB
L1	32KB L1-I & L1-D, 8-way, 64B blocks, 4-cycle hit, inclusive
L2	512 KB, 8-way, 8-cycle hit
LLC	2MB/core, shared & inclusive, 16-way, 20-cycle hit
Memory	DDR-based (Primary): 1x DDR5-4800 channel (baseline)
	CXL-attached (Salvage): 50% bandwidth of primary
CXL	1x16 PCIe 5.0, 100 ns latency overhead
NIC	1x400 Gbps interface

Memory modeling. We model memory behavior using the Mess framework [57]. We profile the memory system of several AMD and Intel server-grade CPUs and observe similar load-latency characteristics. For our evaluation, we use the load-latency curve derived from the AMD EPYC 9754 CPU (Bergamo), as SURGE is most suitable for manycore CPUs with lower bandwidth-per-core availability. Our simulation model uses Figure 4.7’s curve to determine the memory component’s latency as a function of its utilization, assessed at 1000-cycle execution intervals. We apply the same modeling method for both the primary (direct DDR) and the salvage (CXL-attached) memory.

CXL interface modeling. We extend ZSim’s performance models with a new CXL interface component, which models the interface’s latency, utilization, and the impact of the latter on the former. We modeled an x16 CXL link operating at 32.0 GT/s, which corresponds to a 64 GB/s raw transfer rate per direction. At the Physical and Link Layers, we assume a 68-byte flit format. Each link-layer flit consists of 64 bytes of payload and 2 bytes of overhead, resulting in an effective link efficiency of 0.94 after accounting for protocol and formatting overheads [129]. Beyond link-layer efficiency, we also incorporate metadata transfer overheads introduced by the transport protocol for both request and response packets exchanged between the CPU host and the CXL device, following prior work [130]. These metadata overheads reduce the effective peak bandwidth of the CXL interface to $\sim 80\%$ of nominal in the ingress (device-to-CPU) direction and $\sim 40\%$ in the egress (CPU-

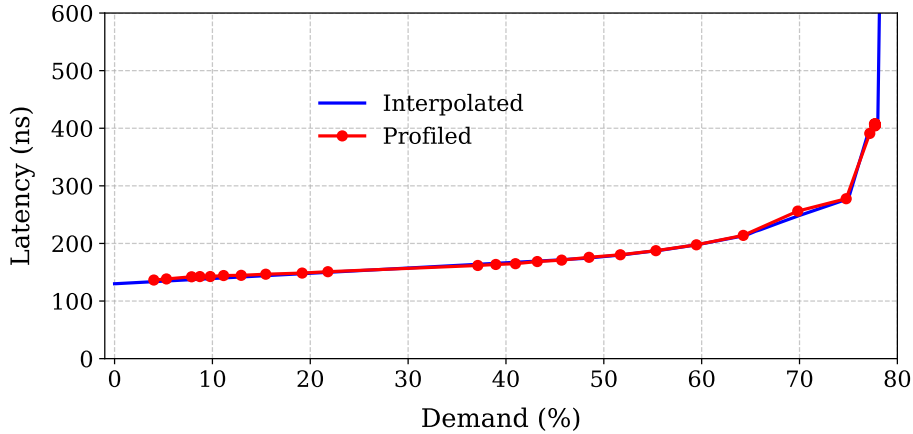


Figure 4.7: Profiled DDR5-4800 memory load-to-use latency vs. demand (as % of peak) on AMD Bergamo.

to-device) direction. The precise fractions depend on each workload’s read/write ratio.

Evaluated Workloads. We evaluate five workloads from each of the Tailbench [131], SPEC [132], and GAP [133] benchmark suites representing the typical cloud and data center environments, with a combination of online services, throughput workloads, and Graph Analytics.

- **Online services:** We select five workloads from Tailbench, including an open-source search engine (xapian), an in-memory key-value store (masstree), a statistical machine translation system (moses), a speech recognition engine (sphinx), and an image recognition model (imgdnn), using their default datasets. We exclude shore, silo, and specjbb as they fail to execute reliably on ZSim.
- **Throughput workloads:** We evaluate five workloads from the SPEC CPU 2017 rate benchmark suite, in ref mode. These include throughput workloads from various domains, such as data compression (xz), scientific computing (bwaves, lbm, fotonik3d), and graph-based optimization (mcf).
- **Graph analytics:** We evaluate five kernels from the GAP benchmark suite using the Kronecker dataset, including Breadth-First Search (bfs), Single-Source Shortest

Paths (*sssp*), PageRank (*pr*), Betweenness Centrality (*bc*), and Triangle Counting (*tc*).

Our evaluation primarily focuses on homogeneous scenarios, where all cores execute the same workload, but also perform a study using a few mixes of co-located workloads in Chapter 4.4.5. We simulate 200 million instructions per core after fast-forwarding to a representative region of interest and warming up the caches. Figure 2.3 shows each workload’s per-core memory bandwidth demand, derived by using an ideal memory with unconstrained bandwidth and fixed zero-load latency.

I/O traffic modeling. ZSim does not simulate any I/O devices by default. To evaluate I/O interference, and without loss of generality, we focus on NICs, as they are typically the highest-bandwidth I/O devices on modern servers. We emulate network traffic with a ZSim-integrated traffic generator from prior work [134] that injects and ejects packets over the simulated CXL link at configurable Poisson intervals. We also capture Data Direct I/O (DDIO) behavior: all ingress network traffic is written directly into designated ways of the Last Level Cache (LLC). Finally, we evaluate three levels of NIC utilization—*high* (80%), *medium* (50%), and *low* (10% of peak bandwidth)—capturing a range of I/O interference scenarios.

I/O utilization scenarios. We use the combination of our workloads and network traffic modeling to investigate common scenarios encountered in cloud deployments. We use a *level₁_level₂* notation to indicate the level of I/O activity on the ingress (RX) and egress (TX) path, respectively.

1. **Low I/O scenario:** This is the most intuitive use case for SURGE, where the CPU executes workloads with little I/O activity, leaving ample opportunity for idle I/O bandwidth harvesting. In such scenario, I/O activity is *low_low*.

2. **Media streaming (egress-heavy) scenario:** Media streaming (e.g., Netflix [135]) is a common workload in datacenters and features heavy egress (TX) path utilization. However, the ingress (RX) path is underutilized, leaving room for effective I/O bandwidth harvesting, as most workloads are read-heavy. In such scenario, I/O activity is *low_high*.
3. **Ingress-heavy scenario:** Although less common than egress-heavy scenarios, ingress-heavy deployments are encountered in presence of workloads with high data ingestion rate from peripheral I/O devices, while leaving the egress path underutilized. In such scenario, I/O activity is *high_low*.

The egress- and ingress-heavy scenarios demonstrate the effect of the CXL interface’s directionality, allowing independent operation of the two data movement directions.

Evaluation. We now evaluate SURGE with the purpose of illustrating the benefits of our proposed idle I/O bandwidth harvesting technique, as well as demonstrating our devised mechanism’s effectiveness and robustness. We aim to answer the following questions:

1. What is SURGE’s performance gain headroom in a variety of workload deployment scenarios? (Chapters 4.4.2 and 4.4.3)
2. How effective is our heuristic in deriving the optimal traffic split for a given workload, and how robust is SURGE’s performance to uncertainty in bandwidth utilization and I/O interference? (Chapter 4.4.4)
3. How significantly do salvage link latency and bandwidth characteristics impact SURGE’s effectiveness? (Chapter 4.4.6)

4.4.2 SURGE under Low I/O Utilization

Figure 4.8 (top) shows the speedup achieved by SURGE over the baseline across our evaluated workloads, under the low I/O scenario. Figure 4.8 (middle) presents the breakdown

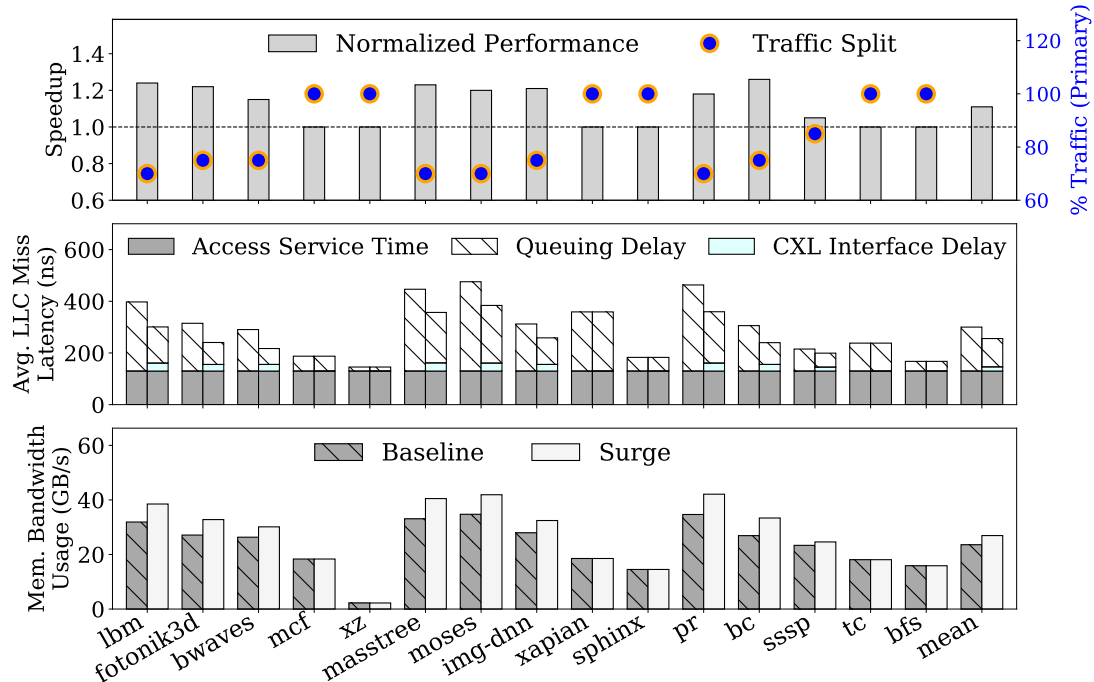


Figure 4.8: Speedup, AMAT, and memory bandwidth usage for baseline vs SURGE in low-networking scenario. Dots indicate the traffic split (% of traffic to primary memory) for each workload, as determined by the SURGE methodology.

of average LLC miss latency (henceforth AMAT for brevity), decomposed into “Access Service Time”—i.e., the unloaded latency of retrieving data from the DRAM device—and “Queuing Delay”. In the baseline, queuing delay occurs exclusively at the primary memory, whereas in SURGE the metric represents the average queuing latency experienced across both the primary and salvage memories. “CXL Interface Delay” quantifies the time spent on the CXL interface *on average*; for example, an access to primary memory experiences zero CXL interface delay. Finally, Figure 4.8 (bottom) shows each workload’s aggregate memory bandwidth demand.

Memory-intensive workloads encountering high queuing delay due to memory bandwidth pressure in the baseline are relieved by leveraging the additional memory bandwidth afforded by the salvage memory, which is accessible over the salvage link. By splitting memory traffic between the primary and salvage memories, the system’s overall memory bandwidth increases and average memory access latency decreases. Across all workloads

with substantial queuing delay in the baseline, SURGE reduces AMAT For this subset of workloads, SURGE directs 25–30% of the traffic to the salvage memory, resulting in an average speedup of $1.2\times$ (up to $1.3\times$). It is noteworthy that while AMAT and performance are qualitatively correlated, the same reduction in AMAT across different workloads does not deliver the same speedup. Individual workload characteristics, such as sensitivity to memory latency, available memory-level parallelism, and temporal burstiness in memory traffic, determine the resulting speedup.

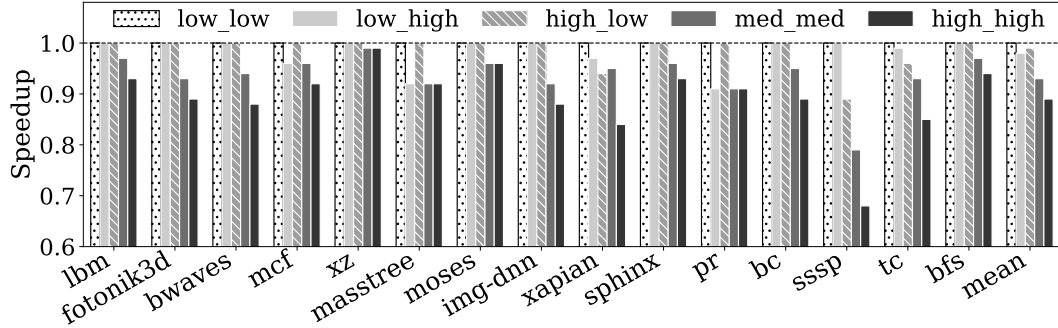
The memory bandwidth demand of several workloads (mcf, xz, xapian, sphinx, tc, bfs) is $\leq 50\%$ of the primary memory’s peak theoretical availability. With such low demands, these workloads encounter minimal memory queuing in the baseline, thus SURGE’s traffic split methodology directs all (or most) of the memory traffic to the primary memory. Hence, such workloads show no or marginal benefit with SURGE.

Considering all workloads, in scenarios with low I/O activity, SURGE achieves an average queuing delay reduction of 36% and delivers an average speedup of $1.1\times$, and up to $1.3\times$, over the baseline.

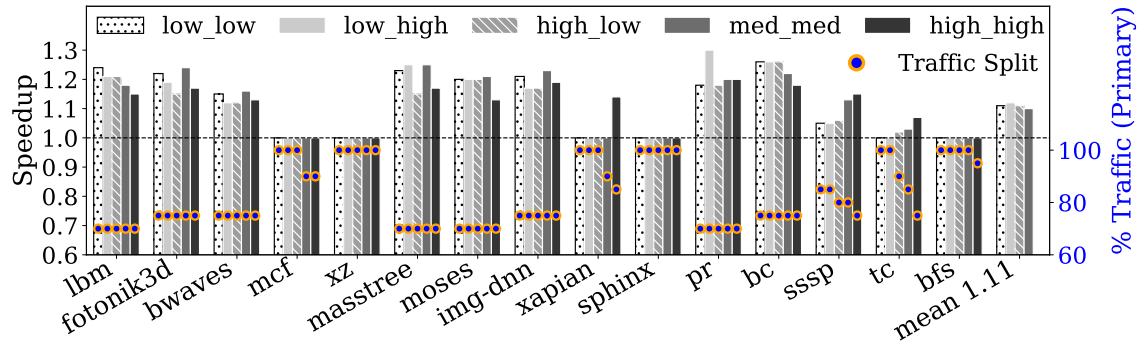
4.4.3 SURGE with Higher I/O Utilization

While the low I/O scenarios are the most intuitive use case for idle I/O bandwidth salvaging, SURGE is applicable and effective even with higher I/O activity. To highlight SURGE’s breadth of applicability, we evaluate scenarios with a variety of I/O utilization levels, as outlined in Chapter 4.4.1. We use different *ingress* and *egress* traffic level combinations to evaluate *RX_TX* I/O traffic interference of *low_high*, *high_low*, *med_med*, and *high_high*.

We first focus on the baseline to study each workload’s behavior with increasing I/O traffic. The results are shown in Figure 4.9a. Even though we use DDIO, increasing I/O traffic exacerbates memory bandwidth pressure, as both application and I/O traffic data move between the LLC and the primary memory. Such I/O-induced traffic introduces interference in the memory hierarchy that degrades performance.



(a) Baseline workload sensitivity to I/O interference. Speedup values norm. to performance with *low_low* interference.



(b) SURGE benefit under different levels of I/O utilization.

Figure 4.9: Impact of I/O traffic interference. $level_1_level_2$ on the legends indicates ingress_egress path utilization.

We make two important observations. First, workloads exhibit different sensitivity to increasing I/O traffic interference. For example, *xz*, which has the lowest bandwidth requirements, remains virtually unaffected; *xapian*, *sssp*, and *tc* exhibit sharp performance drops as I/O utilization increases; and the remaining workloads experience different degrees of more gradual degradation. Second, workloads are affected differently by RX and TX interference, with *tc* and *sssp* more sensitive to the former, and others (like *mcf*, *masstree*, *pr*) more sensitive to the latter. Across workloads, a medium or high interference on both ingress and egress is detrimental to performance.

Figure 4.9b shows the impact of I/O utilization on SURGE’s effectiveness. The *low_low* configuration represents the scenario studied in Chapter 4.4.2, *low_high* represents egress-heavy scenarios, such as media streaming, while *high_low* represents ingress-heavy scenarios. SURGE yields an average speedup of $\sim 1.1\times$ over the baseline, across all I/O

interference configurations. The results indicate that SURGE effectively adjusts the traffic split between primary and salvage memory, balancing bandwidth pressure on primary memory with I/O interface and salvage memory bandwidth availability to minimize the contribution of queuing delays on AMAT.

As I/O activity increases, SURGE’s methodology reevaluates the optimal traffic split between primary and salvage memory. Higher I/O activity means less available bandwidth to salvage on the I/O link, but also increased bandwidth pressure on the primary memory due to network-induced memory traffic, pushing queuing delays higher. With few exceptions, the SURGE-determined optimal traffic split remains stable across I/O utilization levels because

- (i) even under higher I/O utilization, the I/O interface offers considerable bandwidth availability to complement the primary memory’s bandwidth, and
- (ii) the increase in the I/O interface’s contributed queuing delay at higher utilization is still smaller than the primary memory’s high queuing delays under high memory bandwidth demand.

Even in the small subset of workloads where higher I/O interference shifts the preferred traffic split (mcf, xapian, sssp, tc, bfs), SURGE places *less* traffic to primary memory, indicating that queuing at the primary memory bandwidth is more detrimental than queuing on the I/O interface, thereby determining that shifting more traffic to the latter is overall beneficial. As a result, even with *high-high* I/O activity, SURGE yields speedups of up to $1.2\times$. Importantly, even when SURGE does not benefit a workload, it has a performance-neutral effect, delivering performance on par with the baseline.

Finally, in a few workload cases, like xapian, sssp, and tc, SURGE delivers a higher speedup under higher I/O utilization. The increased speedup is because the baseline’s performance degrades fast, while SURGE offers better performance robustness, as its enforced traffic split manages to absorb the detrimental bandwidth interference experienced by the

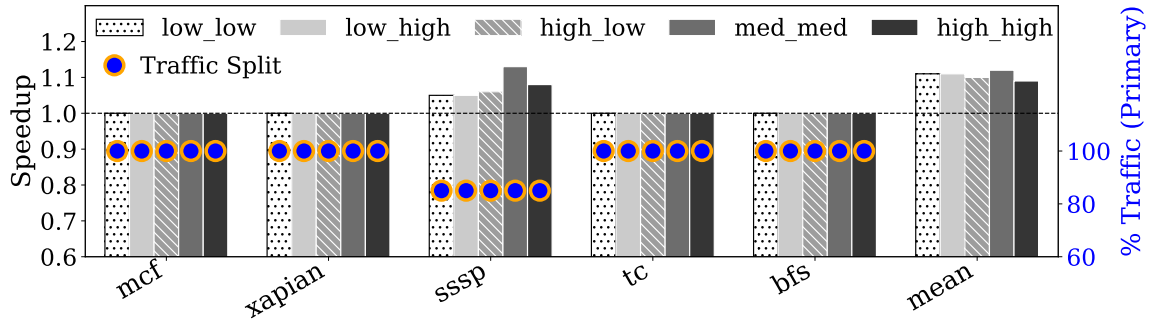


Figure 4.10: Robustness to I/O interference. Traffic split configured expecting *low_low* I/O traffic in all cases. The plot shows only the subset of workloads that are sensitive to the SURGE-selected traffic split, while the mean refers to all 15 workloads.

baseline at the primary memory.

4.4.4 Robustness of SURGE Traffic Split Methodology

SURGE’s traffic split methodology relies on workload knowledge, specifically their memory bandwidth demand and I/O activity levels, derived from profiling. In this section, we study our traffic split approach’s robustness to imprecision in such knowledge, due to inaccurate profiling or fluctuations in memory bandwidth utilization and I/O interference.

Performance sensitivity to I/O interference. We first focus on the impact of I/O interference uncertainty; namely, when I/O utilization ends up being higher than expected at the time of a workload’s deployment—for example, expecting *low* utilization but encountering *medium* or *high* instead, which represents a substantial spike of $5\times/8\times$, respectively. Figure 4.9b already showed that the optimal traffic split for a given workload is largely insensitive to the I/O utilization level. In other words, for most workloads, the SURGE traffic split methodology would determine the same traffic split value across all the I/O utilization levels we consider.

Figure 4.10 focuses on SURGE’s performance for the five workloads that benefit from adjusting the traffic split according to the I/O activity level. The remaining ten workloads’ performance is omitted, because it matches Figure 4.9b’s results. Of those five workloads,

sssp still benefits from SURGE. While the remaining four workloads lose any performance gains when I/O utilization is higher than the expected *low_low*, *none* of them experiences a slowdown, performing on par with the baseline. For some cases, like *mcf* and *xpian* under *med_med* and *mcf* under *high_high*, a comparison between Figure 4.9b and Figure 4.10 reveals that SURGE performs on par with the baseline, despite a difference in the traffic split used in the two experiments. This behavior is attributed to our observation that the optimal traffic split is a *range* rather than a single point, as explained next in the following section on performance sensitivity to traffic split.

Overall, SURGE delivers robust performance results even under unexpected I/O traffic interference (whether sustained or instantaneous in the form of temporal spikes). No workload experiences a slowdown, and average speedup remains at $\sim 1.1\times$ across I/O interference levels, with some workloads still experiencing a performance gain of up to $1.25\times$.

Performance sensitivity to traffic split. For our next robustness analysis, we use the workloads *without any I/O interference* to exclusively focus on memory bandwidth utilization uncertainty. Figure 4.11 shows the resulting AMAT and speedup as a function of the applied traffic split. In the interest of space, we only indicatively show data for the *bc* workload; similar trends hold for all workloads. Evidently, the selected traffic split critically affects performance: the best choice can yield a speedup of $1.2\times$, but a poor choice can incur a 67% slowdown. For *bc*, the optimal primary:salvage split in a deployment scenario without I/O interference falls in the 80:20 to 65:35 range. However, there is a wide traffic split range from 85:15 to 60:40 where SURGE yields substantial performance gains of $1.15\times$ or higher. Given the breadth of this range, SURGE is robust to uncertainty or fluctuations in memory bandwidth usage.

As previously discussed, SURGE’s performance gains primarily stem from reducing the memory system’s AMAT. Figure 4.11 confirms that the highest speedup is achieved by

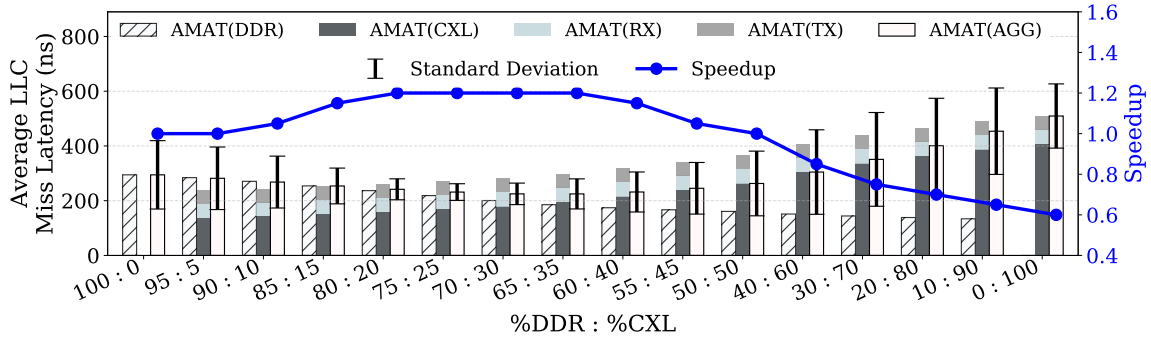


Figure 4.11: AMAT and speedup for the bc workload across the range of possible primary:salvage memory traffic splits. Whiskers show standard deviation.

the configuration with the lowest AMAT. The whiskers on Figure 4.11’s aggregate AMAT bars highlight an additional significant benefit of SURGE. In addition to reducing AMAT, SURGE also reduces memory access latency variance, because queuing delay—which significantly fluctuates over time due to bursty workload behavior—becomes a smaller contributor to overall latency. Higher memory access time predictability additionally contributes to performance improvement, as observed in prior work [54].

SURGE’s methodology determines a traffic split based on average bandwidth utilization derived from profiling. We now evaluate how close the derived split is to the optimal (i.e., best-performing split). We derive the latter experimentally, using a primary:salvage split sweep in 5% increments.

Figure 4.12 (top) shows that, across all workloads, the SURGE methodology’s theoretically derived traffic split matches or is very close to the practically optimal. Importantly, there is a significant margin of deviation from the optimal split within which SURGE performs close to its maximum potential: the error bars indicate the traffic split range that still delivers performance within 10% of the optimal split. The reason for such robustness is that there is a wide plateau around the performance peak, as previously demonstrated in Figure 4.11.

Figure 4.12 (bottom) shows each workload’s memory demand, and the speedup achieved with the SURGE-derived and the optimal split, respectively. Indeed, SURGE’s achieved

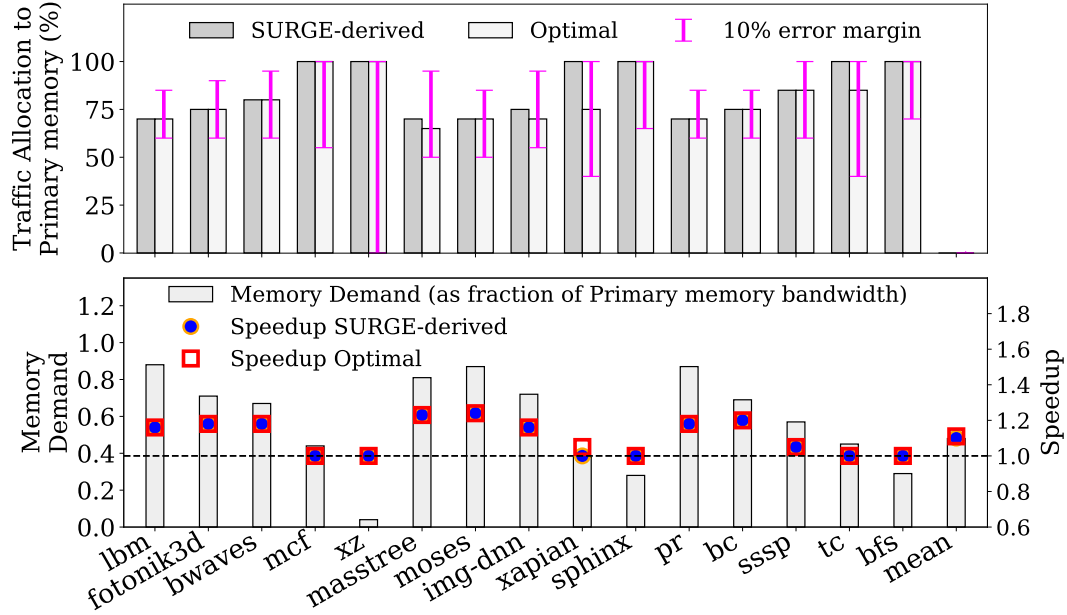


Figure 4.12: Deviation of SURGE-derived traffic split from optimal split, and impact on performance.

speedup is very close to the one achieved with an optimal split—within 1% on average and at most within 5%. As discussed in Chapter 4.4.2, high bandwidth utilization in the baseline is essential to derive substantial gains from SURGE, but that average utilization alone is not predictive of the achievable speedup.

4.4.5 SURGE in Co-located Workload Scenarios

The preceding analyses focused on homogeneous deployments, where all cores execute the same workload and thus observe similar memory and I/O interference patterns. It is common in production datacenters to co-locate heterogeneous workloads on the same server to maximize hardware utilization and amortize operational costs [123]. In such scenarios, SURGE must allocate traffic between primary and salvage memory under diverse and potentially imbalanced bandwidth demands across concurrently running workloads. To study such deployment scenarios, we evaluate SURGE under heterogeneous workload co-location in the low-I/O interference operational regime, where the CXL interface offers ample headroom for bandwidth salvaging.

For this study, we construct three workload mixes with decreasing aggregate memory bandwidth demands, denoted Mix-A, Mix-B, and Mix-C. Each mix comprises four workloads, with each workload running on three cores of our 12-core system. The aggregate memory bandwidth demand of the three mixes ranges from severely bandwidth-constrained to comfortably provisioned, corresponding to approximately 90%, 70%, and 50% of the available primary memory bandwidth, respectively.

- **Mix-A:** lbm, masstree, mooses, pagerank.
- **Mix-B:** lbm, mcf, mooses, sssp.
- **Mix-C:** lbm, fotonik3d, mcf, xz.

For each workload mix, we evaluate two traffic-split policies:

- ***incremental:*** Workloads are deployed incrementally. When a new workload is deployed, SURGE computes its traffic split using the analytical model, based on that workload’s expected memory bandwidth demand, the current primary and salvage memory bandwidth availability (determined by the already deployed workloads on the system), and I/O interface availability. Previously deployed workloads retain their originally assigned traffic splits.
- ***uniform:*** This case assumes that all four workloads are deployed at the same time, hence the aggregate bandwidth demand on the server is known in advance. Given that knowledge, the SURGE methodology determines a single optimal traffic split, which is applied uniformly across all workloads in the mix.

In the case of the incremental workload deployment, each workload’s determined primary:salvage traffic split, and, by extension, its performance, is affected by its deployment order. To demonstrate this effect, we evaluate two different deployment orders for the same workload mix with the incremental traffic split policy, resulting in two evaluated configurations, as follows:

- ***incr-forward***: Workloads are deployed in the order indicated by the set comprising the mix (left to right).
- ***incr-reverse***: Workloads are deployed in the reverse order.

Figure 4.13 shows the results for the three workload mixes, two deployment orders, and two traffic split policies. Figure 4.13 (top) shows the speedup for Mix-A, the evaluated mix with the highest memory bandwidth demand ($> 80\%$ of the primary memory’s peak bandwidth). Each of the mix’s workloads individually exhibits high memory bandwidth demand. As observed by the comparison of the *incr-forward* and *incr-reverse* policies, workloads deployed early in the sequence see greater primary memory bandwidth availability and thus direct most of their traffic to primary memory, with little need to use salvage memory. As additional high-demand workloads are deployed, SURGE progressively shifts more of their traffic to salvage memory to alleviate queuing encountered at the primary memory. Consequently, the order of deployment directly influences which workloads benefit most from the additional bandwidth and which ones rely more heavily on salvage memory. *Uniform* determines that a global 70:30 primary:salvage traffic split is the best single value for the aggregate workload. Overall, for the strongly bandwidth-constrained workload Mix-A, all three evaluated scenarios, *incr-** and *uniform*, deliver a speedup of 1.2–1.3 \times . While *incr-forward* achieves the best mean speedup of 1.3 \times , *uniform* results in a more fair distribution of performance gains across the mix’s workloads.

Figure 4.13 (middle) shows the speedup for Mix-B, which comprises workloads with cumulative demand around 70% of the primary memory’s peak bandwidth. In this mix, *lbm* and *moses* are highly bandwidth-intensive, whereas *mcf* and *sssp* have more moderate bandwidth demands. Under *incr-forward*, the early placement of high-demand workloads causes the primary memory to operate close to saturation by the time later workloads are deployed. To avoid further exacerbating primary memory queuing, SURGE assigns a larger fraction of the late-arriving workload’s traffic (in this case, *mcf*) to the higher-latency salvage memory. For *mcf*, this shift does not yield sufficient queuing reduction to offset the

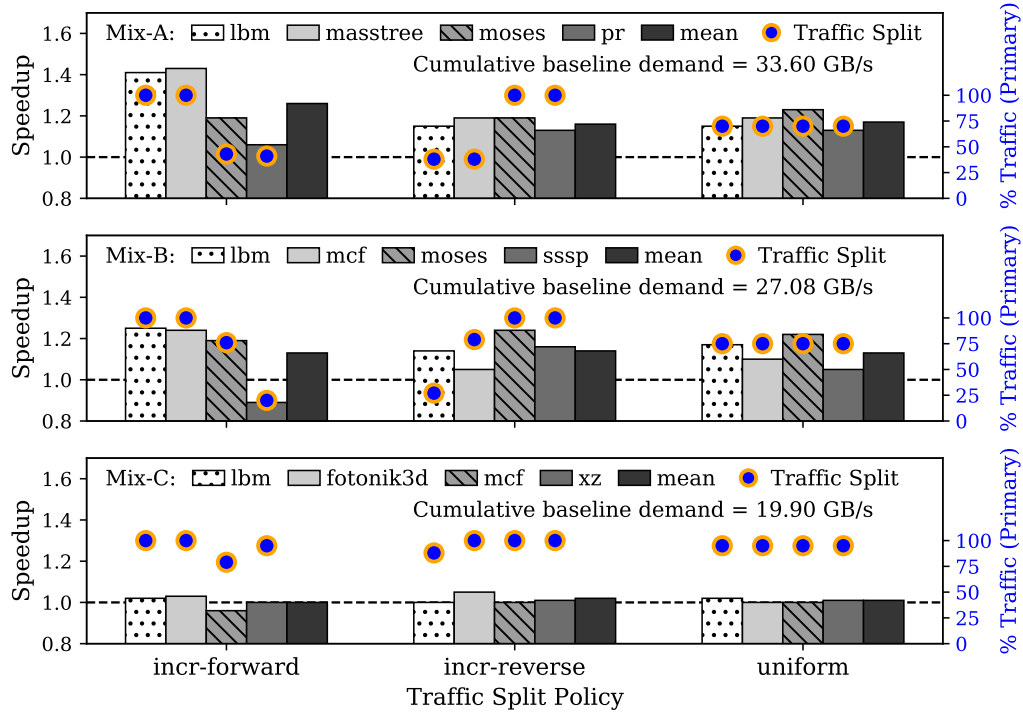


Figure 4.13: Speedup of SURGE over the baseline for three workload mixes. Each sub-plot shows a different workload mix (Mix-A/B/C) with cumulative baseline memory bandwidth demands of approximately 90%, 70%, and 50% of the primary memory bandwidth, respectively.

salvage latency penalty, resulting in a 12% slowdown relative to the baseline. This behavior reflects SURGE’s global objective to prioritize mitigating queuing and protecting the performance of already-deployed, highly memory-bound workloads, even if this occasionally reduces performance for a later-arriving, moderately demanding workload. Overall, SURGE delivers an average speedup of $1.13\times$ for Mix-B.

Finally, Figure 4.13 (bottom) shows the speedup for Mix-C, where the cumulative demand is only about 50% of the primary memory’s peak bandwidth. In this regime, the system is less bandwidth-constrained, and SURGE has limited incentive to divert traffic away from primary memory, so the net effect of salvaging is modest. However, even in this relatively unconstrained setting, deployment order matters with the incremental traffic split policy. Under *incr-forward*, the early placement of lbm and fotonik3d consumes a large share of the available primary bandwidth headroom. As a result, later workloads

such as *mcf* are assigned a higher fraction of their traffic to salvage memory than would be necessary in a more balanced placement, causing avoidable performance degradation. In contrast, in *incr-reverse*, where the less demanding workloads are deployed first, results in a more balanced performance across the mix. Overall, for Mix-C, SURGE has a performance-neutral effect.

Taken together, the Mix-A, Mix-B, and Mix-C results illustrate that both aggregate bandwidth demand and deployment order shape SURGE’s effectiveness under co-location. In addition to these per-mix trends, a cross-mix comparison of a single workload further clarifies the role of overall memory system contention. For example, focusing on *lbm*, the first workload deployed in all three workload mixes, *incr-forward* determines an optimal 100% primary memory traffic split in all cases. Although the traffic split for *lbm* is the same across the three mixes, its speedup with SURGE differs, with the highest gains achieved in Mix-A compared to Mix-B or Mix-C. In Mix-A, the baseline performance of *lbm* degrades more rapidly due to interference from other co-located high-bandwidth workloads (*masstree*, *moses*, and *pr*), which impose greater contention on the primary memory than the co-located workloads in Mix-B or Mix-C. This increased contention exacerbates *lbm*’s queuing delay in the baseline, so even with *lbm*’s traffic fully resident in primary memory under SURGE, the resulting speedup in Mix-A is higher than in the other mixes.

With respect to workload deployment order, it is preferable to deploy less bandwidth-intensive workloads earlier, allowing them to retain primary memory residency, and to admit highly bandwidth-bound, high-queuing workloads later, steering a larger fraction of their traffic to salvage memory. The underlying intuition is that heavily queued workloads derive substantial benefit from any additional bandwidth, and are therefore more tolerant of the salvage memory’s higher unloaded latency, whereas lightly queued workloads can suffer when diverted unnecessarily away from the low-latency primary memory. A SURGE-aware cluster manager can exploit this asymmetry to improve both overall memory-system efficiency and per-workload performance under heterogeneous co-location. Importantly,

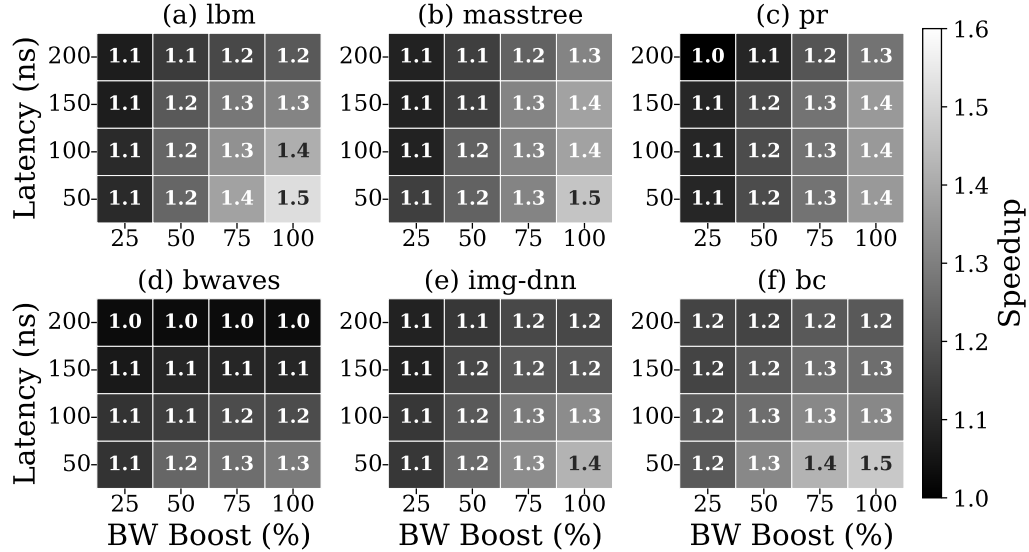


Figure 4.14: Impact of harvest memory’s bandwidth-latency characteristics on SURGE’s effectiveness. SURGE Solo/Pod deployments are more likely to offer characteristics landing toward the bottom left/top right corner, respectively.

although individual workload performance is affected by deployment order, every workload mix we evaluated always benefits from SURGE on average.

4.4.6 Sensitivity to Salvage Link Latency and Bandwidth

We conclude our evaluation with a sensitivity study on the latency and bandwidth characteristics of the CXL-attached salvage memory. We express salvage memory bandwidth in terms relative to the primary memory’s bandwidth. Figure 4.14 shows SURGE’s attainable speedup for 16 bandwidth-latency combinations and six of our workloads with *low_low* I/O utilization. 50% bandwidth boost at 100 ns latency premium is the configuration used throughout the evaluation, which we consider a reasonable design point for a SURGE Pod deployment. Configurations with lower bandwidth boost and potentially lower latency (e.g., 25% boost, 50 ns) are more likely for SURGE Solo deployments, as discussed in Chapter 4.1.4.

While the sensitivity of workloads to the salvage memory’s bandwidth-latency characteristics differs, the same key observation holds across the board: for the bandwidth-

strapped scenarios SURGE targets, bandwidth is more important than unloaded latency penalty. The reason is that SURGE is a technique that shines under bandwidth scarcity, where the primary memory experiences latency spikes due to queuing. In such cases, even a 200 ns unloaded latency premium is preferable, if the memory system can be brought to an operational point prior to the region of exponential queuing delays. Workloads with lower bandwidth demands that don't benefit from SURGE, like mcf (not shown), are insensitive to salvage memory latency premium, because the SURGE traffic split methodology diverts less (or no) traffic to the salvage memory. The effect of latency is more pronounced at higher bandwidth availability, where the contribution of queuing to AMAT is less pronounced. Our sensitivity study highlights that the higher bandwidth that SURGE Pod is likely to be able to afford can more than offset is higher latency premium.

4.5 Chapter Summary

Modern manycore processors increasingly face severe memory bandwidth constraints, as off-chip bandwidth has not kept pace with the growth in core counts. This work presented SURGE, a software-assisted architectural mechanism that dynamically salvages idle I/O bandwidth and converts it to a memory bandwidth boost. SURGE exploits the full-duplex, unified nature of modern serial interfaces such as CXL, which support both memory and I/O traffic over shared physical links.

At the hardware level, SURGE enables dynamic multiplexing of memory and I/O traffic across the same off-chip interface, converting periods of underutilized I/O bandwidth into additional memory bandwidth capacity. Complementing this architectural mechanism, SURGE incorporates a software layer—comprising the operating system and cluster manager—that jointly determines the optimal traffic split between primary (DDR-attached) and salvage (I/O-attached) memory. This decision is guided by an analytical model that minimizes the average memory access time (AMAT) for each workload, based on profiled memory and interface characteristics.

Our comprehensive evaluation demonstrates that SURGE substantially improves system performance across diverse workloads by efficiently repurposing unused I/O resources. When the I/O subsystem is largely underutilized, SURGE achieves up to $1.3\times$ performance improvement, and maintains up to $1.2\times$ even under substantial I/O activity.

In summary, SURGE provides a novel and practical approach to alleviate the off-chip bandwidth bottleneck in modern manycore servers. By unifying architectural and software-level coordination, it transforms otherwise stranded I/O bandwidth into usable memory bandwidth to derive a tangible performance benefit for bandwidth-starved workloads, paving the way for more balanced and bandwidth-efficient server architectures.

CHAPTER 5

OPTIMIZING CLUSTER-SCALE AI TRAINING THROUGH DISAGGREGATED MEMORY EXPANSION

The unprecedented growth in the size and complexity of modern AI/Machine Learning (ML) models has fundamentally reshaped the design priorities of large-scale computing infrastructures. Models powering contemporary applications—such as natural language processing [136, 137, 138], drug discovery [139, 140], text-to-speech systems [141, 142, 143], and large-scale recommendation engines [144, 145]—now require tens to hundreds of terabytes of memory to train efficiently. To accommodate these extreme memory demands, training is typically conducted in a distributed fashion across clusters of high-end accelerator nodes (e.g., GPUs, TPUs, and custom ASICs) interconnected via high-bandwidth, low-latency such as InfiniBand, NVLink, and Ethernet [146, 147].

As discussed in Chapter 2.1.3, a variety of training and parallelization strategies are employed to partition model parameters, activations, and datasets across compute nodes, enabling large-scale distributed training. However, the scalability of such scale-out approaches is inherently limited. As the number of participating nodes increases, communication and synchronization overheads begin to dominate, ultimately constraining overall performance and efficiency. An alternative strategy to reduce reliance on large cluster sizes is to offload model states to host DRAM and slower memory tiers such as NVMe storage. While this tiered offloading temporarily extends the effective memory capacity, it can introduce substantial performance penalties when a larger fraction of model states is placed in slower tiers, owing to their higher latency and limited bandwidth. Sustaining high training throughput therefore requires a *scalable and efficient memory system* capable of providing larger DRAM capacity and bandwidth—without excessive dependence on costly high-bandwidth memory or performance-limiting storage devices.

To sustain continued progress in large-scale AI training, future systems require memory architectures that combine commodity DRAM-based expansion with workload-aware system optimizations, achieving both scalability and cost efficiency. A particularly promising development in this space is the emergence of *disaggregated memory* expansion through high-speed serial interconnects such as Compute Express Link (CXL), which introduces a cache-coherent, high-bandwidth interface enabling processors and accelerators to expand memory capacity seamlessly beyond traditional DIMM-based limits. Such per-node CXL-based memory expansion increases the in-memory footprint available to each training node, thereby reducing dependence on slower storage tiers and alleviating inter-node communication overheads. However, as discussed in Chapter 2.2, realizing these benefits requires *carefully tuned memory system optimizations* that account for both workload behavior and underlying hardware characteristics to mitigate the latency penalties introduced by serial interconnects.

Therefore, understanding and evaluating the benefits of per-node memory expansion requires a comprehensive cross-layer analysis that jointly considers the model architecture, training algorithm, compute capability, memory hierarchy, and network topology. Achieving optimal training performance, in turn, demands a holistic co-design approach where the model’s parallelization strategy is tuned in concert with the cluster’s compute, memory, and interconnect resources. Capturing these complex interactions necessitates a systematic methodology that quantifies how design decisions across the hardware–software stack influence cluster-scale training performance.

To this end, this chapter introduces COMET—a Cluster-scale Optimization Methodology for Efficient Training—which enables rapid and iterative algorithm–hardware co-design for large-scale DL training. COMET provides a unified framework to jointly explore model parallelization strategies and cluster resource configurations, offering quantitative insights into how compute, memory, and interconnect parameters collectively shape distributed training efficiency. The methodology supports composite design-space explo-

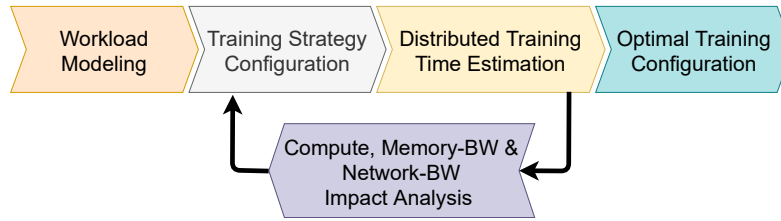


Figure 5.1: COMET methodology overview.

ration, where *cluster resource provisioning strategies* and *parallelization schemes* for a given training task are jointly covaried to uncover optimal configurations that balance performance and scalability.

By modeling layer-wise training behavior, evaluating diverse parallelization strategies, and analyzing the system-level impact of architectural design choices—such as per-node capacity scaling—COMET enables system architects to identify design points that maximize training performance, scalability, and cost efficiency across emerging AI workloads.

5.1 The COMET Methodology

Figure 5.1 shows a high-level overview of our methodology and this section details each of its steps.

5.1.1 Workload Modeling

The first step involves decomposing the model of interest into its layers, along with the number of operations and data movement requirements in each layer. We express each layer as a General Matrix Multiplication (GEMM) between input activations ($M \times K$) and weights ($K \times N$), producing an output matrix ($M \times N$). After decomposing the model into layers, we compute the number of parameters for the operand matrices of each layer. The total size of a model, in terms of number of parameters, is given by the sum of the weight matrices’ (i.e., $K \times N$) elements [148]. Layers that cannot be encoded as GEMMs (e.g., embedding-lookups) are represented by their input/output operand sizes, total number of operations and data moved between memory and the compute unit.

5.1.2 Training Strategy Configuration

In this step, we determine the parallelization strategy for the selected model based on the model type and per-node memory capacity. Different strategies focus on optimizing training for different metrics such as throughput, compute utilization, inter-node communication, etc.

The current version of COMET focuses on Data and Model parallelism (MP and DP— as described in Chapter 2.1.3). Given a target cluster size, we compute the per-node memory capacity requirement as a function of the selected degree of MP and DP in the cluster. We compute the per-node memory footprint to hold all the data (model, input/output matrices) required for the distributed DL training task. The model’s memory footprint is dictated by model states and activations. We compute the memory footprint required for each operand matrix based on its type (i.e., model weights or input/output activations), size of parameters, and the memory optimizations (e.g., ZeRO-DP, as explained later in Section Chapter 5.2). For a cluster of size N , we start with the initial condition where all the nodes are within the MP dimension (i.e., $MP = N, DP = 1$) and, collectively, hold a single copy of the entire model. Then we sweep the (MP, DP) degree to the other extreme (i.e., $MP = 1, DP = N$), considering all power-of-two combinations, with $MP \times DP = N$.

Doubling the DP dimension implies halving the MP dimension, so half the number of nodes must hold an entire copy of the model. Consequently, the per-node memory capacity requirement doubles. To illustrate, Figure 5.2 shows a cluster configured as two data-parallel node groups ($DP = 2$) and each DP node group consisting of m nodes performing m -way model parallelism ($MP = m$), for a total node count $N = 2m$. To contain a copy of the entire model of size C across each DP group’s m nodes, each node requires a minimum memory capacity $P = \frac{C}{m}$. Moving from a $DP = 2, MP = m$ to a $DP = 4, MP = \frac{m}{2}$ configuration doubles the per-node memory capacity requirement to $2P$, as each DP group of $\frac{m}{2}$ nodes must now hold the entire model C . As a result, the cluster’s total memory capacity also doubles from $2mP$ to $4mP$.

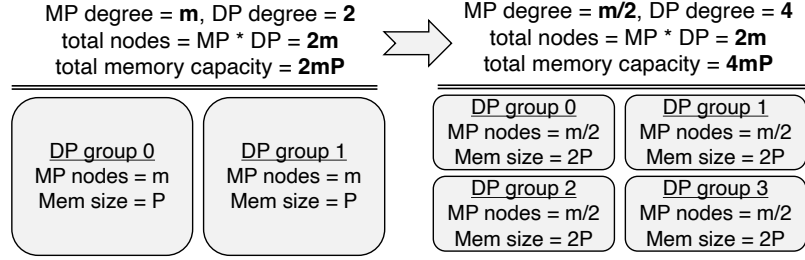


Figure 5.2: Variation of per-node memory capacity requirements as a function of MP and DP degrees in a fixed-size cluster.

The chosen (MP, DP) degree not only affects the memory requirement per node, but also results in different computation requirements and communication behavior. Therefore, for each (MP, DP) combination, we also derive the volume of per-node computation and inter-node communication.

5.1.3 Distributed Training Time Estimation

The computation and communication requirements per layer for each (MP, DP) combination are fed into a performance model that estimates the model’s training time. The performance model estimates end-to-end training time as a function of the modeled cluster’s per-node compute capability, memory capacity and bandwidth, network bandwidth and topology.

A key design decision in COMET is opting for generality and breadth rather than detailed modeling of individual components, as our methodology is intended to enable rapid and scalable exploration of a vast design space, rather than highly accurate performance estimations. COMET’s goal is to enable users to glean *performance trends* as cluster parameters are varied both jointly and separately. Therefore, we chose to go with detailed analytical models for compute, memory, and network rather than be tied to any specific technology or component instance. Performance estimation for DL training models lends itself well to analytical modeling (as opposed to cycle-level simulation), as their computation, memory access, and communication patterns exhibit regularity typically absent

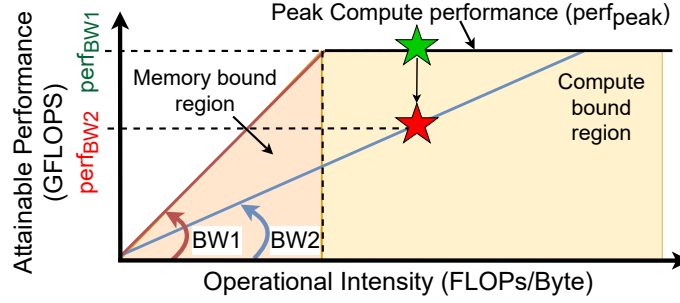


Figure 5.3: Roofline model. Attainable performance shifts for the same OI, depending on available memory bandwidth.

from general workloads [149, 150]. By sweeping generic characteristics like TOPS, bandwidth, and latency, COMET’s users may easily create proxies for specific components or technologies of interest, such as GPUs with different computational capabilities, memories with different capacity/bandwidth characteristics (e.g., HBM vs. DDR), or networks with different bandwidth/latency characteristics (e.g., InfiniBand vs. NVLink). Next, we describe how individual performance models are constructed and tied together in COMET.

5.1.4 Compute delay estimation

To produce compute-delay estimations independent of any compute node’s microarchitectural details, we employ a roofline model [151, 152]. In each case, the compute node of interest is represented by its peak performance ($perf_{peak}$ in *GFLOPS*) and memory bandwidth.

Figure 5.3 shows a roofline model, which consists of a compute-bound region (under the flat line) and a memory-bound region (under the slanted line). The flat line is dictated by the target node’s $perf_{peak}$. The slanted line’s slope denotes the compute node’s available memory bandwidth $BW_{mem}(GB/s)$. In COMET’s design space exploration, a change in the available memory bandwidth changes the roofline’s slope and, correspondingly, the intersection point with the $perf_{peak}$ horizontal line shifts.

For each training phase, a workload layer’s Operational Intensity (OI) is calculated as

$$OI (FLOPs/byte) = \frac{\# \text{ of floating-point operations}}{\text{memory traffic}} \quad (5.1)$$

where *memory traffic* is the total number of bytes moved between the memory and processor for operand and result matrices of the layer. Chapter 5.1.5 elaborates on our estimation of memory traffic generated by each layer. Based on the target compute node’s characteristics, a workload layer’s OI may place it under the roofline’s compute-bound or memory-bound region. The maximum compute performance achieved for a layer i with OI_i is $perf_{max} (GFLOPS) = \min \{perf_{peak}, OI_i \times BW_{mem}\}$.

Using $perf_{peak}$ and BW_{mem} as knobs in our methodology, we estimate their impact on the maximum attainable performance $perf_{max}$ for each workload layer, and thereby the compute delay for each layer i in each training phase as

$$\text{compute delay}_i (s) = \frac{\# \text{ of floating-point operations}}{perf_{max}} \quad (5.2)$$

The total *compute* delay for one training iteration is the sum of compute delays for the forward pass, backward pass, and weight update for each of the model’s layers.

Why Roofline Model? The roofline model makes our workflow fast and versatile, and while not enough to estimate absolute performance, it captures performance trends. Despite having limitations, as a first-order model, the roofline model has still been used widely in a broad range of prior work [153, 154, 155, 156, 151, 157, 158, 152, 159, 160, 161] due to its versatility and utility in quickly and correctly highlighting general *performance trends*, given the general compute and memory capabilities of a compute node. COMET as a general methodology is not limited to roofline and its modular design allows plugging in more detailed compute models, such as runtimes captured from real GPUs or simulated accelerators modeled in appropriate tools, like GPGPU-Sim [162], or ScaleSim [163]. However, in this work we use the more general roofline model to focus on the methodology’s utility and decouple the results of the conducted case studies from any specific compute unit’s

microarchitectural characteristics.

5.1.5 Memory traffic estimation

Memory traffic is the cumulative number of bytes transferred between the main memory and compute unit while performing the desired functionality. For a hypothetical compute node with infinite on-chip buffer space, all operands can be fetched exactly once from the memory, resulting in a very high OI (cf. Equation (5.1)). However, every realistic compute unit’s limited on-chip buffer space can only hold a limited set of data operands. Therefore, a layer’s matrix operands must usually be fetched multiple times from the memory to complete the required operations, thereby lowering the resulting OI . We construct a linear model to better estimate the memory traffic for a GEMM operation on a compute node with an on-chip buffer of configurable size.

Consider a GEMM operation between two matrices of U and V bytes, generating an output matrix of W bytes. We assume one of the input operands is tiled to fit in the on-chip buffer, and the other operand/output are streamed in/out of the compute node, respectively. For an on-chip buffer size of S bytes, we estimate the memory traffic (in bytes) as $\min\{\Psi_1, \Psi_2\} + W$, where $\Psi_1 = \lceil U/S \rceil \times V + U$ and $\Psi_2 = \lceil V/S \rceil \times U + V$. In practice, for U and $V \gg S$, tiling the smaller operand results in less data movement (e.g., if $U < V$, Ψ_1 is the tiling method of choice, resulting in about $V - U$ less data movement).

An additional important architectural design knob we want to investigate with COMET is that of memory expansion, whereby the compute unit’s Local Memory (LM) is enhanced with a secondary level of memory, which we refer to Expanded Memory (EM). Such a setting can be enabled by allowing the compute unit to directly access its host CPU’s memory, or by physically attaching additional memory over CXL [129], photonic links, or other (current or future) technology. Investigating such an option using CXL-attached memory (which offers considerably higher bandwidth per pin than DDR) for DL training is of particularly high relevance given the growing interest in deploying it as a new memory hierarchy

component [95, 11, 164].

To investigate this system design option, we consider per-node memory expansion, with the available bandwidth as our sensitivity analysis knob. To model performance with such a hybrid memory system, we instrument our roofline model with the new memory system’s effective memory bandwidth (bw_{hybrid}), which depends on the fraction of data accessed from local/expanded memory ($data_{LM}/data_{EM}$) at the local/expanded memory’s bandwidth (bw_{LM}/bw_{EM}). We estimate the hybrid memory system’s effective bandwidth as:

$$bw_{hybrid} = \frac{total_data_accessed}{\frac{data_{local}}{bw_{LM}} + \frac{data_{EM}}{bw_{EM}}} \quad (5.3)$$

To illustrate, accessing 240GB of data in a hybrid memory system with 80GB of LM, $bw_{LM} = 2TB/s$, and $bw_{EM} = 1TB/s$ results in $bw_{hybrid} = 1.2TB/s$. Using Equation (5.3), we can determine the cluster’s performance as a function of the bandwidth offered by the hypothetical memory expansion technique used.

5.1.6 Communication delay estimation

During the training process, nodes continuously exchange data. Their communication delay is dictated by the total communication volume, the aggregate network bandwidth available between the nodes, and the dynamic network utilization. In addition, depending on the training phase, the communication among the nodes may be blocking or non-blocking. In the forward-pass and input-gradient phases, communication is blocking along the model-parallel dimension; in the weight-gradient phase, communication is non-blocking across the data-parallel dimension. Blocking communication falls on the critical path of a training phase, while non-blocking communication can be (partially) overlapped with compute, thus ameliorating its impact on resulting training time. The combination of data movement volume, available network bandwidth, communication type, and concurrently performed computation, dictates how much of the occurring communication is exposed, affecting training time. Ultimately, for each layer, the total exposed communication delay determines

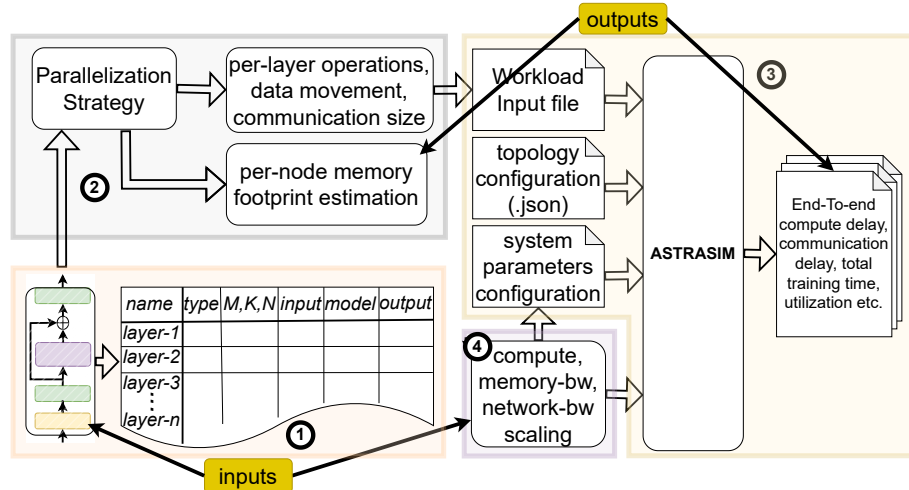


Figure 5.4: COMET implementation and workflow.

whether the layer is compute- or communication-bound on a given system configuration.

subsection Total training time estimation Finally, we combine the per-layer compute delays (Chapter 5.1.4) with each layer’s corresponding communication characteristics (data volume and communication type—Chapter 5.1.6) to determine the degree of computation/communication overlap and derive the training time per iteration and, by extension, total training time.

Overall, COMET comprises an iterative training time estimation process as shown in Figure 5.1. For each model, different training strategies are considered (cf. Chapter 5.1.2) and the training time is estimated as described in Chapter 5.1.3. The process is repeated for different cluster sizes, and compute, network, and memory parameters to guide the user’s selection of parallelization strategy and cluster resource provisioning to optimize for the target metric of merit—raw training performance, or training efficiency (i.e., training time relative to resources deployed).

5.2 COMET Implementation

Figure 5.4 illustrates the implementation of the toolchain that operationalizes the COMET methodology. In the frontend (① and ②), the toolchain characterizes each model layer

by generating parameters that capture its computational workload and communication requirements. These parameters are then passed to the backend (③ and ④), which models end-to-end training performance as a function of the target cluster’s compute, memory, and interconnect configuration.

Figure 5.5 summarizes the hardware components and configuration ranges of the clusters evaluated using COMET. By enabling rapid and holistic exploration across system parameters, COMET provides quantitative guidance for selecting balanced resource provisioning strategies that maximize training performance and efficiency for a target set of deep learning workloads.

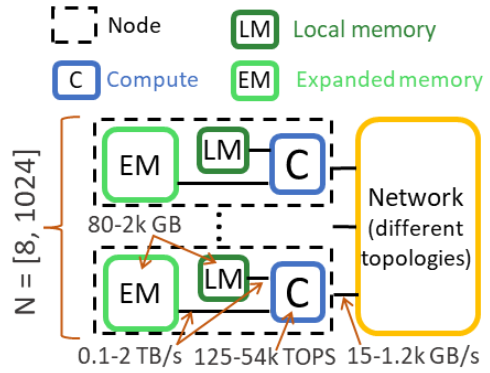


Figure 5.5: COMET evaluation space.

5.2.1 DL Model Analysis

In step ①, we analyze the DL model of interest and break it down into its layers. Each layer is represented as a sequence of GEMMs of input activations, model parameters and resulting output activation matrices. The size of each matrix dimension M , K , N is derived from the model’s hyper-parameters and batch size. Depending on the model type, a set of fixed independent hyper-parameters form the model’s signature. For example, in case of Transformers, hidden-dimension, # layer-stacks, and # attention-heads characterize the model size. Then, based on the derived GEMM dimensions, we compute the total number of model and activation parameters required for each layer. In addition, the size of each operand matrix (i.e., input activations, model parameters, and output matrices) per node is affected by the MP/DP degree.

5.2.2 Parallelization Strategy

In step ②, we select a parallelization strategy for the given workload, generate the corresponding workload input file and feed it to the performance simulator to estimate the distributed training time. In addition, the workload’s required memory footprint is computed by aggregating model parameters, optimizer states, gradients, residual states, and checkpoint-activations. The workload input file must describe the characteristics of each layer of the workload, which includes the number of floating-point operations, data volume (in bytes) moved between memory and compute, communication collective, and communication volume (in bytes). As per ZeRO-Infinity, we compute the total number of operations and matrix operands required in both forward (FP) and backward (i.e., Input Gradients (IG) and Weight Gradients (WG)) training phases.

As described in Chapter 5.1.5, we estimate the bytes transferred between the processor and main memory for each layer in each training phase. Similarly, we estimate the total number of operations and matrix operands required for each layer in each training phase. Based on the parallelization scheme used for the workload, we also determine the communication collective and compute the communication volume required per layer in each training phase, as well as the required per-node memory footprint to fit the model states and working dataset.

We use ZeRO-DP (os+g) [165], a.k.a. ZeRO-2, to derive the per-node memory footprint of model-states. ZeRO-2 avoids replication of optimizer states and gradients on each node and distributes them across the data-parallel dimension to reduce the per-node memory footprint, while avoiding additional communication overhead. For residual states, we estimate the memory footprint as $\# \text{ activation-parameters} \times 2 \text{ bytes}$ assuming fp16 activation parameters. We exclude the memory required for checkpoint activations in our per-node memory footprint estimate. Typically, in large models such as Transformer-1T, the memory footprint required to store the checkpoint activations is significantly larger than the intermediate activations and hence are offloaded to host memory. Therefore, during the training

process, we only consider the Activation Working Memory [87], which is the memory required to hold the intermediate activations between two consecutive checkpoints.

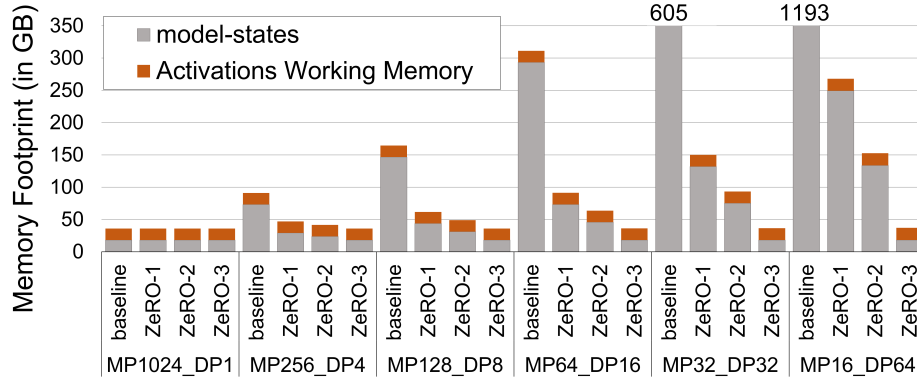


Figure 5.6: Per-node memory footprint for Transformer-1T model with baseline and different ZeRO-DP stages.

As explained in Chapter 5.1.2, the parallelization strategy affects the required memory footprint per node. To illustrate with a concrete use case, Figure 5.6 shows how the per-node memory footprint requirement changes for a Transformer-1T model on a fixed cluster size of 1024 nodes, as a function of different stages of ZeRO and a decreasing MP degree (the invariant being $DP \times MP = 1024$). In baseline (i.e., no ZeRO optimizations), the model footprint per node increases exponentially as the MP degree reduces. The same trend holds even with memory optimizations like ZeRO-2: although the growth is slower, the model footprint per node eventually exceeds the typical memory capacity of a single device, highlighting the value of MP to enable in-memory training of huge models. Among all the ZeRO-DP optimizations, ZeRO-3 stands out as it provides the lowest memory footprint per node and remains unaffected by MP reduction. However, ZeRO-3 incurs a $1.5\times$ communication overhead compared to the baseline. Other approaches such as ZeRO-Offload and ZeRO-Infinity offload data and compute to the host machine resources to reduce the memory capacity pressure on accelerator nodes.

5.2.3 Total Training Time Estimation

As shown in Figure 5.4 step ③, COMET plugs into a cost model for training time estimation. We use the ASTRA-SIM simulator [166, 167] for this purpose. ASTRA-SIM is a discrete event-based simulator developed by Meta, Intel and Georgia Tech that can simulate distributed training for a variety of DL workloads. It accepts a workload configuration, topology description and system parameter file to simulate the distributed DL training on a target cluster, and outputs the end-to-end training time breakdown and resource utilization.

At a high level, ASTRA-SIM consists of a workload, system, and network layers. The workload layer is responsible for instantiating a model and scheduling training loops for simulation. The system layer provides the mechanisms for collective primitives and scheduling of communication tasks, similar to collective communication libraries such as NCCL [168]. The network layer provides the topology interface via network APIs to support a fast analytical model [167] or detailed network simulation using Garnet [169] and NS3 [170].

An ASTRA-SIM workload configuration file describes the DL workload to be simulated, and consists of the compute time, communication collective, and the collective size for each layer of the workload. For each layer, ASTRA-SIM schedules the communication collectives and *overlaps* the communication delay with compute delay to estimate the total training time. ASTRA-SIM’s analytical network backend—used in our current COMET implementation—estimates the communication delay of an event based on the topology and network bandwidths specified in the configuration files. ASTRA-SIM supports multiple collective communication primitives and scheduling algorithms. The symmetric network topology of distributed training platforms and topology-aware collective communication algorithms minimize the network congestion, enabling the analytical network backend to accurately model the communication overhead [171, 172, 173, 167]. ASTRA-SIM’s runtime projections for 8–16 node clusters has been validated against real systems to be within 5% difference [167].

For COMET, we chose ASTRA-SIM as the cost model for training time estimation given its modular architecture for plug-and-play compute and communication models, and implementations of diverse collective communication algorithms and scheduling strategies. We integrated our roofline and data movement models (Chapters 5.1.4 and 5.1.5) in ASTRA-SIM to enable modeling a range of compute units with hybrid memories (LM and EM). Using our added models and the data provided in the workload input file (operations and data size per layer), the compute delay per layer is estimated. ASTRA-SIM uses the compute delay, communication collectives and volume, the network topology, and system parameters provided to perform a training simulation and generates an end-to-end per-layer training-time breakdown for each training phase. It reports the compute and exposed (i.e., non-overlapped) communication times for each layer in the FP, IG, and WG phases.

For our design space exploration of parallelization strategies on a cluster of size N , we sweep the degree of MP and DP such that always $MP \times DP = N$. This emulates the effective increase in per-node memory capacity as MP decreases in favor of DP, as demonstrated in Figure 5.6, and generates the corresponding workload input file for each (MP, DP) combination.

5.2.4 Cluster Parameter Reconfiguration and Iterative Modeling

Finally, step ④ provides a set of knobs to perform sensitivity analysis by varying the key component parameters, which include the network topology, compute capability, as well as memory and network bandwidth and latency.

By iterating through steps ② to ④ (Chapters 5.2.2 to 5.2.4), we obtain the resulting training time for different system configurations and hardware parameters to identify the best combination of parallelization strategy and cluster resources. The target optimization metric may be raw performance or cost efficiency (i.e., performance relative to cluster’s provisioned resources). In practice, this iterative analysis helps architects explore trade-offs between scaling efficiency, hardware utilization, and memory system balance when

provisioning large-scale AI training clusters.

5.3 Evaluation and Case Studies

We now leverage COMET to evaluate cluster design decisions across multiple dimensions—including cluster size and network configuration, per-node memory capacity, and compute capability—in the context of large-scale Transformer and DLRM training.

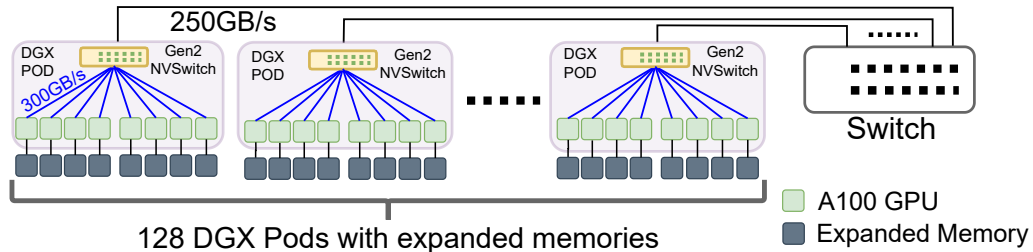


Figure 5.7: Cluster of 1024 A100 GPUs with expanded memories, grouped in 128 8-GPU pods. Link bandwidth is per direction.

5.3.1 Baseline Evaluation Setup and Workloads

Figure 5.7 visualizes our baseline 1k-node DGX A100 [174] cluster with expanded memories (EM) and Table 5.1 summarizes the parameters used to model it in ASTRA-SIM. We evaluate training performance for Transformer and DRAM models, which represent the largest models currently deployed. Our evaluation predominantly focuses on the Transformer model due to its higher complexity and broader range of (MP, DP) training strategies (Chapter 2.1.3). Towards the end we present a subset of DRAM evaluation results in Chapter 5.3.8.

5.3.2 Transformer Evaluation

Transformer-based language models are huge natural language processing models used extensively in language modeling, machine translation, text summarization, AI chatbots (e.g., ChatGPT), etc. We model the Transformer-1T architecture and hybrid model &

Table 5.1: Baseline NVIDIA DGX A100 system parameters.

Single-node Parameters (NVIDIA A100 GPU)	
Peak Performance ($perf_{peak}$)	624 TFLOPS (fp16)
Local Memory Capacity / Bandwidth	80 GB / 2039 GB/s
On-chip SRAM size	40 MB
Cluster Parameters	
Compute Pod	NVIDIA A100 DGX (8-GPU)
Cluster Size	1024 nodes (128 pods \times 8 GPUs)
Intra-pod Network BW per GPU	300 GB/s / direction (NVLink Gen-3)
Inter-pod Network BW per GPU	31.25 GB/s / direction (InfiniBand)
Physical Topology	Hierarchical Switch
Collectives Implementation	Logical Ring

data parallelism approach as described in Megatron-LM [78]. Table 5.2 breaks down the Transformer-1T model into its layers, and summarizes each layer’s type and dimensions.

Table 5.2: Transformer model layers and dimensions.

Layer	Type	#Stacks	GEMM Dimensions		
			M	K	N
Input Embedding	Table Look-up	1	$b \times seq$	sub_vocab	d_{model}
Layer Norm	Element-Wise Mult	N	$b \times seq$	1	d_{model}
Query Projection (Q_i)	GEMM	N	$b \times seq$	d_{model}	$h \times d_k$
Key Projection (K_i)	GEMM	N	$b \times seq$	d_{model}	$h \times d_k$
Value Projection (V_i)	GEMM	N	$b \times seq$	d_{model}	$h \times d_v$
$U_i = softmax(Q_i K_i^T / \sqrt{d_k})$	GEMM	N	$b \times seq$	$h \times d_k$	$b \times seq$
$Y_i = U_i V_i$	GEMM	N	$b \times seq$	$b \times seq$	$h \times d_v$
$concat(Z_i = Y_i B_i, \dots, Z_n)$	GEMM	N	$b \times seq$	$h \times d_v$	d_{model}
Residual Addition	Element-Wise Add	N	$b \times seq$	1	d_{model}
Layer Norm	Element-Wise Mult	N	$b \times seq$	1	d_{model}
$Y_i = GeLU(XA_i + k)$	GEMM	N	$b \times seq$	d_{model}	sub_ff
$Z_i = Y_i B_i + k$	GEMM	N	$b \times seq$	sub_ff	$h \times d_{model}$
Residual Addition	Element-Wise Add	N	$b \times seq$	1	$h \times d_v$
Output Embedding	Table update	1	$b \times seq$	d_{model}	sub_vocab

Legend: d_{model} : hidden dimension, h : # of attention heads, b : mini-batch size, seq : sequence length, d_k/d_v : key/value tensor dimension per attention head, sub_ff : portion of MLP layer per MP node, sub_vocab : chunk of total vocabulary per MP node

A Transformer model comprises a stack of multiple encoder and decoder structures, each composed of multi-head attention layers followed by fully connected feed-forward and residual layers [148]. Each of the encoder and decoder stack takes input and output embeddings as inputs which map a sequence of symbols into a continuous representation.

Latest Transformer models comprise up to trillion parameters and must be trained over

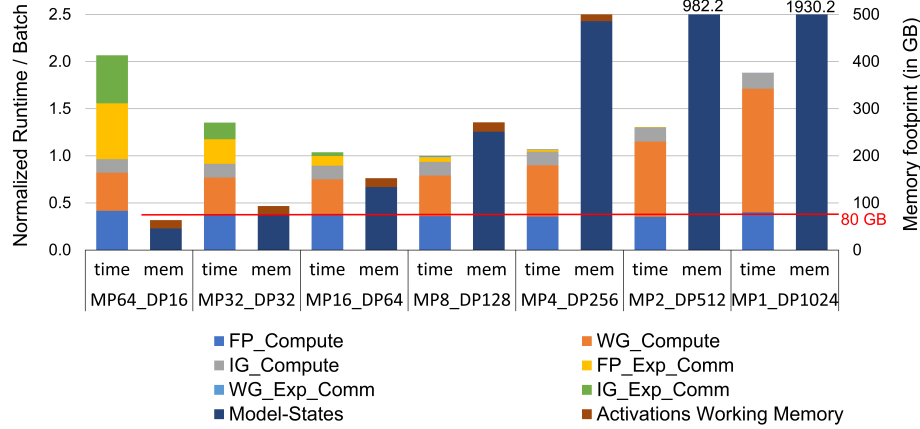
hundreds of high-end processing nodes in a distributed manner using multiple levels of parallelization [80, 78], despite advanced techniques employed reduce their required memory footprint [175, 165, 87].

5.3.3 Parallelization Strategy Impact on Performance and Memory Requirements

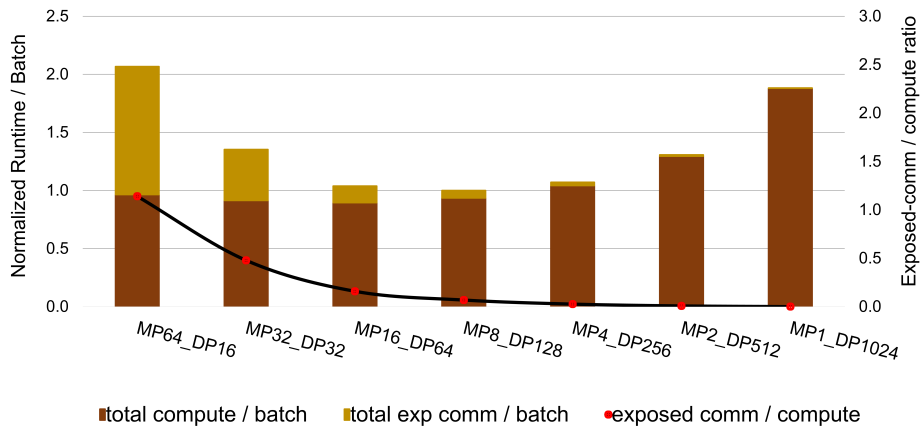
A Transformer model can be trained via a wide range of parallelization strategies. We first evaluate the impact of different (MP, DP) parallelization strategies on that cluster’s performance and per-node memory requirements. We begin by sweeping (MP, DP) under the invariant $MP \times DP = 1024$. In this section, training time estimations ignore per-GPU memory capacity constraints, assuming infinite per-node memory capacity accessible at the baseline system’s peak memory bandwidth.

Figure 5.8a shows the training time breakdown and corresponding per-node memory footprint for several (MP, DP) configurations, assuming a constant memory bandwidth of 2039 GB/s, irrespective of capacity. The total training time is a combination of compute delays and exposed communication delays (Chapter 5.1.6). The compute and exposed communication time is broken down into three components to indicate the three main phases in a training iteration: forward pass (FP), input gradients (IG), and weight gradients (WG). As MP decreases in favor of more DP groups, the required memory footprint per node increases; therefore, fitting the model in our baseline NVIDIA A100 Graphics Processing Unit (GPU)’s 80GB memory requires an MP degree of 64 or higher.

Memory capacity requirements aside, the best-performing configuration is MP8_DP128, where the exposed communication in FP and IG phases, and the compute delay exhibit their lowest values. Note that WG communication (*WG_Exp_Comm*) is fully overlapped by the WG compute (*WG_Compute*) in every configuration, hence not visible. The configurations left of MP8_DP128 (higher MP) are communication bound due to the exposed blocking communication patterns in FP and IG phases across the MP dimension. In contrast, for configurations right of MP8_DP128 (lower MP), the effective model footprint per



(a) Breakdown of comp./comm. time & per-node memory footprint.



(b) Exposed communication to compute ratio.

Figure 5.8: Training runtime of Transformer-1T with varying MP/DP degree (norm. to best-performing MP8_DP128 config.).

node grows as the model is distributed across fewer nodes per Data Parallel Unit (DPU), and hence becomes more memory bound resulting in higher compute delays dominating runtime, while barely any WG communication is exposed.

Figure 5.8b better highlights the changing balance between compute and exposed communication time for the same (MP, DP) range. Under high MP degrees (e.g., MP64_DP16), training time is dominated by exposed communication time. As MP decreases in favor of increasing DP, the fraction of runtime spent on communication becomes negligible from MP8 onwards. *MP8_DP128 is the optimal configuration because it strikes the best balance, effectively overlapping communication delays with compute, without getting into a*

memory-bandwidth-bound region that causes drastic compute time increase.

5.3.4 Effect of Memory System Design

Based on Chapter 5.3.3’s results, the best-performing MP8_DP128 configuration requires ~ 250 GB of memory to fit the model, exceeding the 80GB the NVIDIA A100 GPU baseline’s per-node memory capacity by $3\times$. The best-performing configuration achievable under the 80GB memory constraint is MP64_DP16. The required capacity for MP8_DP128 could be achieved with a hybrid memory system, complementing the GPU’s HBM with a secondary DRAM-based memory that offers additional capacity, albeit accessible at lower bandwidth. As mentioned in Chapter 5.1.5, such additional capacity could be provided by allowing the GPU to access its host CPU’s memory, or by attaching additional memory to the GPU over CXL [129] or other technology.

Figure 5.9 shows the performance results normalized to MP64_DP16, the best-performing configuration of in-memory distributed training that is feasible without memory expansion (Figure 5.8a). The heatmap’s x-axis shows the bandwidth to expanded memory, while the varying (MP, DP) degree on the y-axis is a proxy for the required capacity of that expanded memory (see memory requirements in Figure 5.8a). MP64_DP16 and configurations with higher MP remain unaffected by the expanded memory’s bandwidth, as the dataset entirely fits in each node’s local memory, and configurations with MP higher than 256 are omitted, as they perform strictly worse.

The heatmap guides system architects in determining what memory expansion technology can be used to boost a cluster’s training performance, revealing the range of expanded memory characteristics that allow building a cluster with lower training time than the MP64_DP16 baseline. Conversely, the data can also be leveraged to derive the equally valuable information of memory technologies that would not be applicable.

We provide two illustrative examples derived from Figure 5.9:

Ex. 1: The theoretically optimal MP8_DP128 configuration is achievable with a hybrid

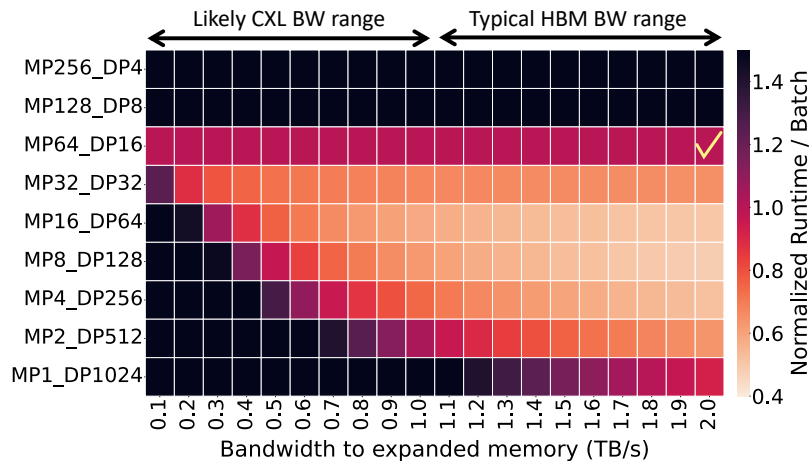


Figure 5.9: Effect of memory bandwidth availability to the expanded memory system. The check-mark indicates the baseline configuration to which all the other values are normalized.

memory system offering at least $4.25\times$ higher aggregate capacity than the baseline (cf. Figure 5.8a), and outperforms the baseline if the expanded memory is accessible at a bandwidth of at least 500GB/s.

Ex. 2: A system architect considering CXL-attached memory can quickly deduce that, in order to benefit the given workload, the technology must be capable of delivering 500GB/s to 340GB of memory, at a minimum. In practice, that would require a memory device accessible over 32 lanes of CXL 3.0.

5.3.5 Effect of Latency-Bandwidth Trade-Offs in Expanded Memory Systems

Emerging memory expansion technologies such as CXL-attached DRAM provide substantial additional capacity but differ from on-package HBM not only in bandwidth but also in access latency. Fully assessing the practicality of expanded-memory training thus requires understanding how both bandwidth and latency jointly shape end-to-end performance, particularly for workloads with varied and latency-sensitive access patterns.

Figure 5.10 examines this bandwidth–latency trade-off by evaluating the MP8_DP128 configuration, which requires approximately 340 GB of capacity beyond the HBM avail-

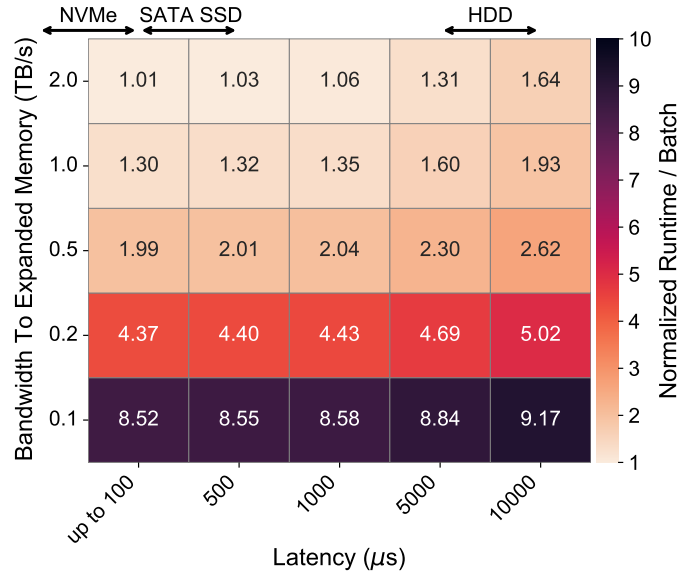


Figure 5.10: Effect of latency vs. memory bandwidth availability to the expanded memory system. All the values are normalized to the baseline MP8_DP128 configuration at 2039 GB/s memory bandwidth.

able on an NVIDIA A100 node. All results are normalized to a hypothetical baseline where the expanded memory operates at HBM-class bandwidth (2039 GB/s) and near-HBM latency. The heatmap varies the link latency on the x-axis and the available bandwidth on the y-axis, spanning a range from NVMe, SATA SSD, and even HDD latencies for contrast.

Overall, the results indicate that bandwidth is the primary determinant of performance under the streaming-dominated access patterns captured in our compute model. With one operand matrix tiled into on-chip SRAM and the other streamed from memory, the resulting fp64 optimizer-state traffic sustains a high demand on memory bandwidth, making performance substantially more sensitive to bandwidth provisioning than to moderate changes in access latency. Consequently, within the sub-100 μ s latency range characteristic of CXL-attached memory devices, the impact of latency remains negligible as long as sufficient bandwidth is available.

However, as access latency approaches storage-class timescales, the balance shifts and latency becomes a more prominent performance limiter. Once latencies enter the hundreds of microseconds to millisecond range, the delay incurred on each streamed access can no

longer be effectively amortized by high bandwidth, resulting in noticeable slowdowns. For example, at 0.5 TB/s of available bandwidth to expanded memory, increasing the link latency from 100 μ s to 10 ms leads to a 31.7% performance decline, reflecting the point at which latency dominates the memory access cost. These effects would be further amplified under access patterns with less inherent streaming regularity, where fine-grained data dependencies increase sensitivity to remote-memory latency.

Taken together, these results underscore that effective memory expansion for large-scale training requires a balanced combination of high bandwidth and reasonably low latency. While bandwidth dictates the achievable throughput under typical training compute patterns, latency bounds the performance envelope when access patterns become less streaming-friendly or when expanded-memory utilization deepens. CXL-attached memory occupies a favorable region in this design space, but technologies with storage-class latencies—even if provisioned with very high bandwidth—would struggle to support training configurations such as MP8_DP128 that rely heavily on timely access to large optimizer and activation state.

5.3.6 Effect of Per-node Compute Capability

The tremendous demand for DL training drives rapid evolution of the hardware used in clusters deployed for such purpose. COMET can be used to study the effect of per-node compute capability by scaling each node’s peak performance ($perf_{max}$) to model hardware of different generations. Figure 5.11 shows the effect of per-node compute scaling on the MP8_DP128 configuration, assuming a memory system of varying bw_{hybrid} as a function of bw_{EM} , and sufficient capacity to hold the model. At the highest bw_{EM} of 2TB/s, halving compute capability — e.g., by replacing the baseline A100 with a lower-end GPU — increases the runtime by 50%, while doubling it reduces the runtime by 25%. Scaling compute further has diminishing returns, as training becomes communication bound. For lower memory bandwidth availability, the impact of compute capability scaling further

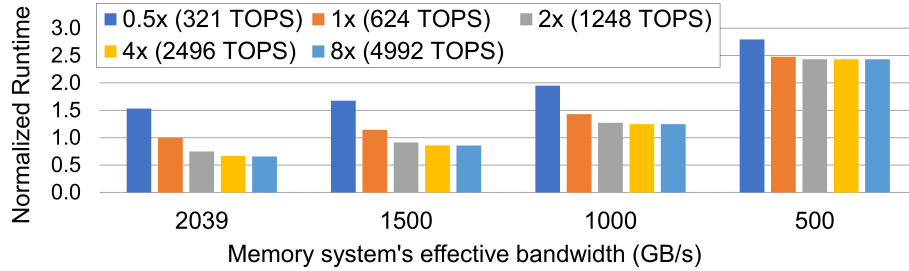


Figure 5.11: Effect of node compute capability relative to baseline A100 GPU (MP8_DP128 configuration).

diminishes due to the additional bottleneck of memory bandwidth. *System designers can employ such studies to predict the impact of a next-generation GPU on a cluster’s overall performance.*

5.3.7 Effect of Networking Capability

A cluster’s network bandwidth plays a critical role in the overall training time. Especially at higher MP configurations, training time is dominated by the exposed blocking communication patterns in forward and input gradient phases. In our modeled cluster, there is an *intra-pod* bandwidth to communicate with GPUs within a pod and a lower *inter-pod* bandwidth for communication across pods (see Table 5.1). We use a *Hierarchical Collective* implementation [176, 173] in our evaluations, which first reduces data across GPUs of the same pod, followed by inter-pod reduction. Such local network bandwidth-aware collective optimization reduces the communication volume on the lower-bandwidth inter-pod links.

Figure 5.12 shows the effect of both intra- and inter-pod network bandwidth scaling on training time for two different (MP, DP) configurations: the communication-bound MP64_DP16 and the compute-bound MP8_DP128. In each plot, we vary the two network bandwidths while keeping the compute and memory bandwidth constant to the baseline system’s values (Table 5.1).

Unsurprisingly, MP64_DP16 is majorly affected by network bandwidth, as the MP dimension straddles several 8-node pods, which feature high intra-pod bandwidth. Halving

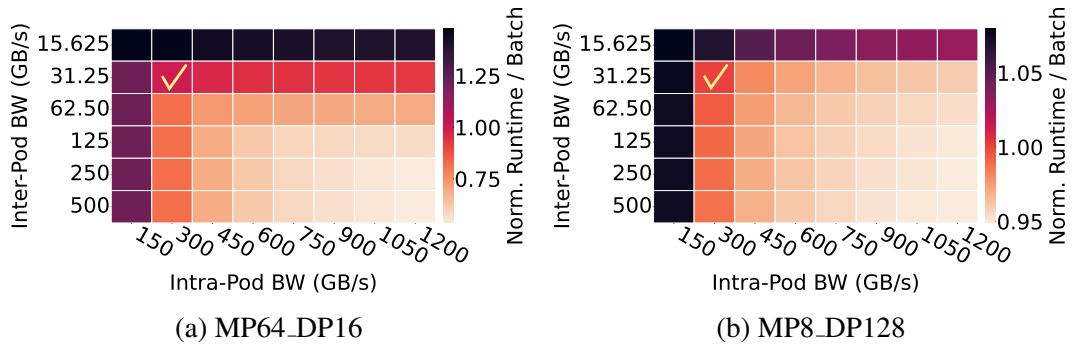


Figure 5.12: Effect of network capabilities on cluster-scale performance. The check-mark indicates the baseline configuration to which all the other values are normalized. 300/31.25 GB/s is currently a typical NVLink/InfiniBand configuration.

intra-/inter-pod bandwidth results in a 48%/22% slowdown, respectively. On the contrary, doubling intra-/inter-pod bandwidth reduces runtime by 3%/18%, respectively, while doubling both reduces runtime by 27%. Evidently, the most effective network scaling scales bandwidth on both dimensions, while scaling only one dimension’s bandwidth provides reduced or marginal gains.

This effect is attributed to the mapping of the workload on the underlying cluster. The performance-critical all-reduce communication in forward and backward propagation is bottlenecked by *both intra- and inter-pod links*. Thus, increasing only one dimension’s capability yields limited gains (as the other dimension remains a bottleneck), while boosting the capabilities of both dimensions has an amplificatory effect. In contrast, when MP8_DP128 training is feasible, the network’s role is much less critical. Reducing both intra- and inter-pod bandwidth to 50% only degrades performance by 11%, while even increasing both by 4× only boosts performance by 5%.

We conduct an additional experiment to focus on identifying the ideal balance of the bandwidth available at the two dimensions. Instead of varying the two values independently, as previously done in Figure 5.12, Figure 5.13 shows the performance trends when the available network bandwidth is re-distributed between intra-/inter-pod links, while its aggregate value always remains fixed. For the baseline MP64_DP16 configuration with an aggregate full duplex bandwidth of 331.25 GB/s per direction (300 GB/s intra-pod and

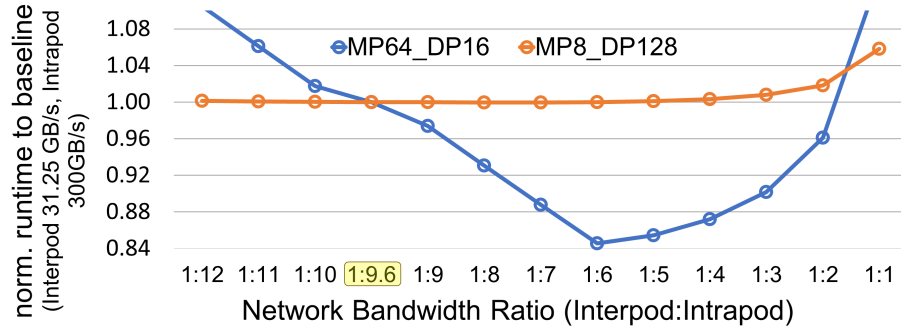


Figure 5.13: Impact of relative inter-/intra-pod network bandwidth allocation on training runtime. The total bandwidth is kept constant at 331.25 GB/s across all the ratios. The highlighted 1:9.6 ratio represents the baseline cluster’s configuration.

31.25GB/s inter-pod), we find an optimal bandwidth ratio of 1:6 between inter-/intra-pod network bandwidth (i.e., 284 GB/s intra-pod and 47.32 GB/s inter-pod). For lower/higher ratios, inter-/intra-pod network traffic becomes a bottleneck, respectively. In the case of compute-bound MP8_DP128, where the MP dimension is entirely contained within a pod, the performance-critical MP communication is entirely dictated by intra-pod bandwidth while the DP communication across the pods is affected by the inter-pod link bandwidth. Therefore, MP communication is not a bottleneck and performance is largely insensitive to the rebalanced bandwidth ratio. Performance starts dropping beyond 1:5 (i.e., with intra-pod bandwidth less than 276GB/s), as MP communication starts reappearing as a bottleneck. Overall, 1:6 inter-/intra-pod network bandwidth provisioning appears to be the ratio that best accommodates both training configurations, improving training time by up to 15% compared to the default 1:9.6 ratio.

System designers can use such analysis to determine the topology and interconnect technology to use (e.g., Ethernet, InfiniBand, NVLink), while balancing the resulting performance with the associated cost of the selected hardware resources.

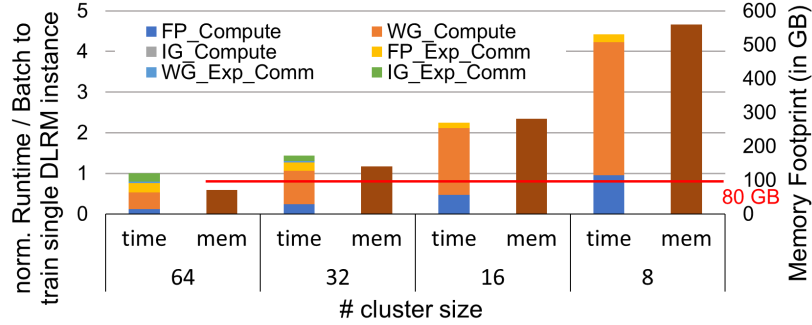
5.3.8 DRAM Evaluation

DLRMs are among the largest deep learning models used widely for generating personalized content [177, 178, 179] (e.g., movie recommendations [180, 181], e-commerce cat-

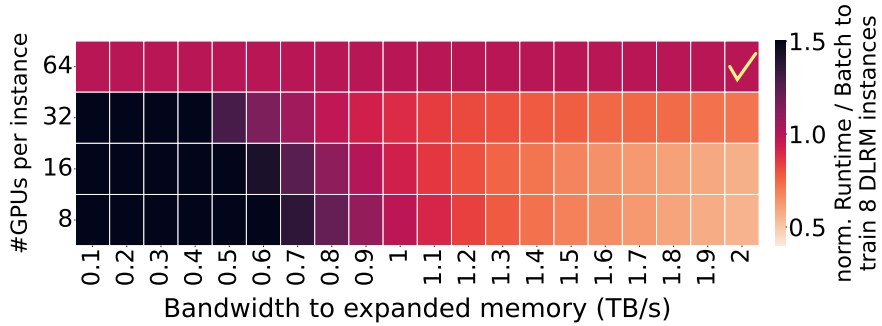
alogs [67, 182, 183]), comprising up to trillions of parameters. DLRM model contains large *embedding tables* that capture the latent space of user and product feature interaction. In each DLRM, inputs are classified into sparse and dense features which represent the categorical data and continuous features. While sparse features are used to look up the embedding tables, dense features are processed using a bottom stack of MLP layers. The result of embedding lookup and output of bottom MLP layers are combined and given as input to the top Multi Layer Perceptron (MLP) layers which finally predict the probability of an event (such as click-rate) occurrence. We model the DLRM architecture and parallelization strategy as described by Rashidi et al. [184].

Unlike Transformer models that can be trained using a wide range of model versus data parallelism configuration points, the training structure for DLRMs is more rigid. DLRM follows a hybrid parallelization strategy where the bottom embedding layers are sharded across multiple nodes and performs an all-to-all communication during forward and backward propagation while the MLP layers are replicated on each node in a data parallel fashion and perform an all-reduce during backward propagation. DLRM therefore does not offer the same MP/DP configuration knob we sweep for Transformer models.

We now briefly cover evaluation highlights for the training of a 1.2 trillion parameter DRAM, modeled as described in Table V of Rashidi et al. [184]. Figure 5.14a shows the training time breakdown for single DRAM instance and corresponding per-node memory footprint for different cluster sizes. Since the DRAM’s memory footprint is relatively smaller than Chapter 5.3.2’s Transformer-1T model, we start with a smaller cluster comprising only 8 pods of our baseline DGX cluster (64 GPUs in total). As the cluster’s size decreases, the exposed communication delay decreases at the cost of increased memory footprint per node required, resulting in a compute delay increase. However, the overall increase in training time is sublinear with the node count reduction, especially in the 64–16 range. Thus, *memory expansion can not only be used to improve DRAM training efficiency, but also better performance for a training workload comprising several DRAM models.*



(a) Breakdown of comp./comm. time & per-node memory footprint.



(b) Effect of memory bandwidth availability to the extended memory system.

Figure 5.14: DLRM’s training performance normalized to 64 nodes with 2TB/s memory bandwidth..

This is a common use case, as large corporations often need to train multiple DRAMs for different purposes [185, 186].

Figure 5.14b evaluates the overall turnaround time of training 8 DRAMs on 64 GPU nodes as a function of available bandwidth to the expanded memory. *While DRAM’s performance is more sensitive to memory bandwidth, results qualitatively match Chapter 5.3.4’s takeaways.* Performance improvement opportunities require memory expansion solutions delivering at least 75% additional memory capacity at 800GB/s; a 200GB expanded memory accessible at 1.5TB/s improves training time by 1.5 \times .

5.3.9 Comparative DL Training on Different Clusters

We conclude COMET’s utility demonstration by comparing 11 different DL training clusters, summarized in Table 5.3: nine GPU-based clusters, a Google TPU v4, and a Tesla

Table 5.3: Per-node details of various cluster configurations.

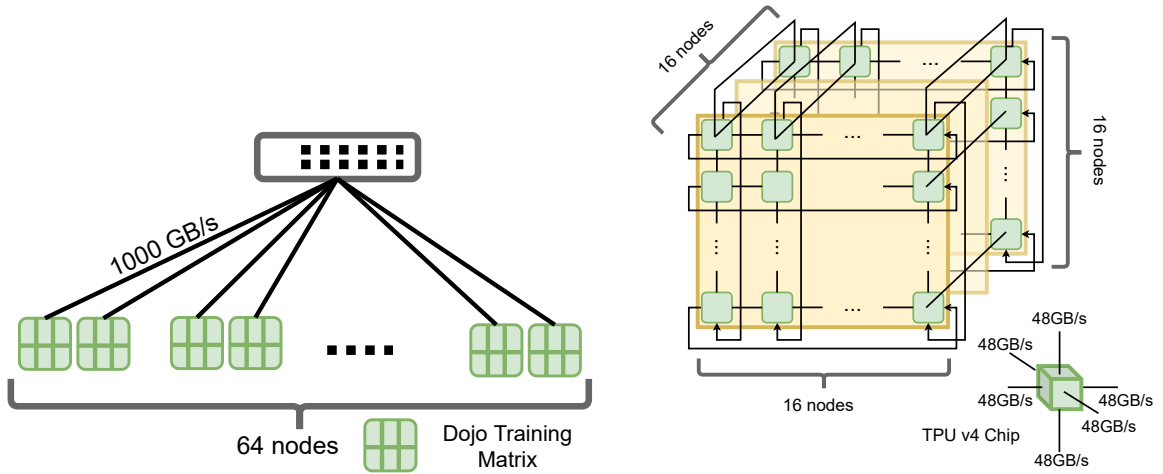
config	compute		memory				network (topology : bandwidth per node)
	node	Peak TOPS	local cap. (GB)	local bw (GB/s)	exp. cap. (GB)	exp. bw (GB/s)	
A0	V100	125	80*	900	0	0	two-level switch :
A1					480	500	150 GB/s intra-pod, 6.25 GB/s inter-pod
A2					201	1000	
B0	A100	625	80	2039	0	0	two-level switch :
B1					480	500	300 GB/s intra-pod, 31.25 GB/s inter-pod
B2					201	1000	
C0	H100	1979	80	3350	0	0	two-level switch :
C1					480	500	450 GB/s intra-pod, 62.5 GB/s inter-pod
C2					201	1000	
Dojo	Tray	54,300	640	16000	0	0	one-level switch : $20 \times 50GB/s$ per direction
TPU v4	TPU	275	32	1200	39	1200	3D torus : $6 \times 48GB/s$ per direction

*Although the V100 GPU features 32GB of memory, we model 80GB instead to keep the memory system configuration options of clusters A, B, and C aligned.

Dojo cluster. Our goal is *not* to declare the “best” system—as they drastically differ in cost, node count, definition of a “node”, etc.—but to demonstrate the different behavior across three very dissimilar clusters when training the same huge model.

DGX cluster variants: We model three base 1024-GPU cluster variants—A, B, and C—with different resource (compute, memory, network) provisioning as described in Table 5.3. We base each design on a major GPU model (V100, A100 and H100). All GPU cluster variants are organized in 16-GPU pods and feature a two-dimensional network like the one shown in Figure 5.7. For each base cluster variant, we evaluate three memory systems—0, 1 and 2—with different characteristics, for a total of nine GPU cluster variants. Memory system 0 consists of only local GPU memory without any capacity expansion. Memory systems 1 and 2 are hypothetical expanded memory systems accessible at 0.5TB/s and 1TB/s, respectively.

Dojo cluster: We model a Dojo cluster of 64 nodes (“trays”, each comprising several training tiles and interface processors) as shown in Figure 5.15a. Each node has 66GB of on-chip SRAM, 640GB of memory with a 16 TB/s bandwidth, and 54.3 PFLOPS peak performance. Given limited publicly available information, we model the network topology



(a) Cluster of 64 Dojo D1 Training Matrices connected by a switch.

(b) Cluster of 4096 TPU v4 chips connected in 3D Torus topology.

Figure 5.15: Evaluated configurations for Dojo and TPU clusters.

as a single logical switch delivering 1 TB/s of full duplex network bandwidth to each node [187].

TPU v4 cluster: We model a TPU cluster of 4096 TPU-V4 chips, as shown in Figure 5.15b, connected in a 3D Torus topology with 48GB/s full duplex links. Each TPU features 32MB of on-chip SRAM, a 32GB HBM with a memory bandwidth of 1.2TB/s, and 275 TFLOPS peak performance [188].

Figure 5.16 shows the speedup for DRAM and Transformer-1T across different cluster configurations. Every cluster except A0, B0, C0, and Dojo assumes per-node memory expansion with capacity and bandwidth characteristics listed in Table 5.3. DLRM training is modeled as described in Chapter 5.3.8: Clusters A0, B0, and C0 use 64 nodes to run a single DLRM instance. A1, B1 and C1 leverage their expanded memory to train one DLRM per 16 nodes. Likewise, A2, B2 and C2 use 8 nodes per instance. The reported speedup for DRAM refers to training a total of 8 model instances. For Transformer-1T, speedup refers to training a single instance on the entire cluster. All reported speedups are normalized to cluster A0 as baseline.

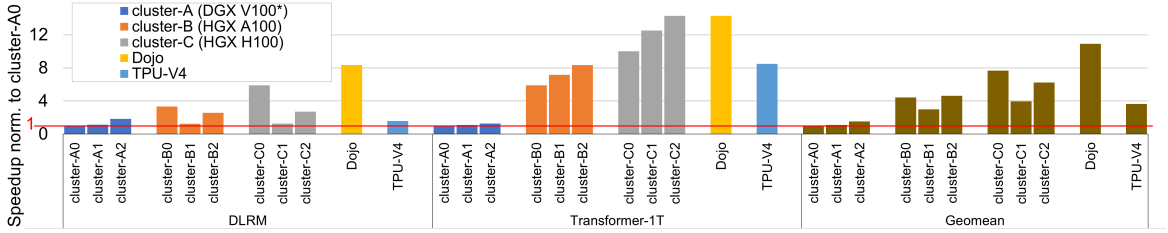


Figure 5.16: Comparison of runtime across different cluster configurations, normalized to cluster A0.

Clusters A2, B2 and C2 benefit from their expanded memory at 1TB/s, delivering $1.8\times$, $2.6\times$, and $2.7\times$ speedups, respectively, for DLRM. While clusters A1, B1 and C1 fare poorly for DLRM due to their lower bandwidth to expanded memory, B1 and C1 deliver a good speedup of $7.2\times$ and $12.5\times$, respectively, for Transformer-1T. Increasing expanded memory bandwidth to 1TB/s further improves speedup for Transformer-1T (e.g., to $14.3\times$ for C2).

Cluster A2’s double bandwidth to expanded memory improves cluster A1’s DLRM performance by $1.64\times$. Due to the memory-bound nature of DLRM, memory bandwidth improvements are more critical than compute capacity or network bandwidth. On the other hand, Transformer-1T is more sensitive to the compute capacity and network bandwidth, and therefore low compute and network bandwidth drastically reduces speedup opportunities for clusters A1 and A2. Transformer-1T benefits from Tensor Processing Unit (TPU)’s large compute capacity and network bandwidth, but the DLRM suffers from its low memory capacity and local memory bandwidth. In contrast, Dojo significantly benefits both workloads due to large on-chip SRAM, memory capacity, and high network bandwidth.

Overall, among GPU cluster variants, there is no single optimal configuration for both Transformer-1T and DRAM, as different workloads are impacted differently. Disregarding any cost considerations, the best GPU cluster on average is C0, delivering a $7.7\times$ speedup over the baseline A0 cluster. *Memory expansion is an effective technique for all clusters when training Transformers, but only for the lowest-end cluster A on average, due to DRAM’s memory bandwidth sensitivity.* This study demonstrates that, as one size does

not fit all, system architects should evaluate a workload mix representative of the main use cases for the target cluster to determine the system configuration that best fits the ensemble.

5.4 COMET takeaways: versatility and speed

Our extensive evaluation demonstrates COMET’s versatility and the breadth of case studies it facilitates. We illustrated that COMET enables joint sensitivity analysis of the effect of node compute capability, memory system design, and network provisioning on cluster performance, facilitating balanced resource provisioning and identification of cost-reduction opportunities. COMET also allows rapid exploration and evaluation of memory system design on a cluster’s performance as a function of its capacity and bandwidth characteristics, helping system designers determine what existing technologies for memory expansion are viable to improve training performance, and gauge the impact of relevant future technologies.

While Chapter 5.3’s evaluation focused on demonstrating the utility and flexibility of the tool, an additional key strength of COMET is the short turnaround time of experiments, allowing researchers to rapidly glean performance trends. Once a target model is broken down into its layer-wise representation as specified in Chapter 5.1.1, exploring a broad design space is a matter of few hours. COMET’s methodology allows modeling different compute nodes, network topologies, memory bandwidths and parallelization strategies in an embarrassingly parallel fashion on commodity processors. To provide some concrete data points, generating the two memory bandwidth sensitivity heatmaps (Figure 5.9 for Transformer-1T and Figure 5.14b for DLRM) takes about 5 hours / 45 minutes, respectively, on a single 24-core Intel Xeon Silver server. Other analyses presented in the paper require comparable runtimes. Such rapid exploration of a wide range of design choices is a valuable capability COMET contributes.

5.5 Chapter Summary

The rapid growth in scale and complexity of modern AI/ML models has redefined the performance boundaries of large distributed training infrastructures. As computational capabilities continue to scale faster than memory and interconnect subsystems, overall efficiency is increasingly constrained by the ability to deliver adequate memory capacity and bandwidth per node. While scaling out across additional accelerator nodes remains the dominant strategy for accommodating massive models, it inherently introduces communication and synchronization overheads that limit scalability. Expanding per-node memory capacity presents an alternative path—reducing inter-node communication pressure and improving data locality—provided that the underlying memory system can sustain the required bandwidth and latency characteristics. This growing tension underscores the need for a unified design methodology that captures how innovations in the memory hierarchy, such as CXL-based memory expansion, interact with compute and interconnect subsystems to shape end-to-end distributed training performance in ever-larger clusters.

To address this need, COMET introduces a cluster-scale methodology for evaluating and optimizing distributed DL training through joint co-design of compute, memory, and interconnect resources. COMET models the coupled effects of workload structure and cluster architecture, quantifying how model- and data-parallel strategies, memory hierarchy parameters, and network bandwidth collectively influence training efficiency. By combining roofline-based analytical modeling, hybrid memory traffic estimation, and parameterized communication modeling within the ASTRA-SIM simulation framework, COMET enables rapid and extensible exploration of system performance without dependence on specific hardware microarchitectures. This capability is particularly important as model sizes, memory footprints, and cluster scales grow to the point where exhaustive measurement or full-cycle simulation across the entire design space becomes impractical.

Building on the insights from prior cluster-scale modeling frameworks discussed in

Chapter 6.3, COMET integrates the strengths of analytical, profile-guided, and heterogeneous modeling approaches. It achieves the scalability and generality of analytical frameworks while incorporating the extensibility and fidelity of trace-driven analysis through its integration with the latest ASTRA-SIM-2.0 [167] infrastructure. The enhanced toolchain supports trace-based, profile-guided estimation using the industry-standard Chakra Execution Trace format [189, 190], which provides a unified, graph-based schema for representing AI/ML workloads. By capturing compute, memory, and communication operations—along with their dependencies and metadata—in an interoperable representation, this integration enables consistent modeling of execution traces from diverse frameworks and supports evaluation of both conventional and heterogeneous systems, including those augmented with disaggregated or CXL-attached memory.

Ultimately, the primary goal of COMET is not to prescribe a single “best” parallelization or hardware configuration, but to provide a practical, scalable lens through which architects can reason about performance trends in the face of rapidly growing model and cluster sizes. By supporting fast, holistic sweeps over large algorithmic–architectural design spaces and enabling more detailed analysis to be focused on a narrowed set of promising configurations, COMET helps identify where additional memory capacity, bandwidth, or new architectural features such as CXL-enabled memory expansion are most impactful. In doing so, COMET equips system designers with a flexible and timely methodology for uncovering co-design opportunities, quantifying trade-offs, and guiding the evolution of future large-scale distributed AI training infrastructures.

CHAPTER 6

RELATED WORK

6.1 Prior work related to HTM capacity enhancements

We discuss prior work most closely related to HinTM’s mechanisms—compiler-driven classification in STM and user-annotated privatization—and shall highlight the distinction between HinTM’s page-based memory safety classification and prior proposals on page-based TMs.

User-annotated privatization. Notary [111] is the only work we know that uses a privatization method to boost an HTM’s effective TXN capacity. It provides a programmer interface for *manual* coarse-grained marking of thread-private data structures, and mentions automatic marking of thread-private stack pages. However, Notary’s proposed design is more conceptual than algorithmic, lacking concrete implementation details of how the hardware maintains and retrieves that information to use it. In contrast, we meticulously detail the methods HinTM employs to identify safe memory and pass that information to hardware. Although we focused on *automation*, HinTM can trivially support both coarse-grained (i.e., Notary-style) and fine-grained programmer annotations as well.

ISA escape actions support for HTM. Intel and IBM HTMs offer suspend/resume operations that can be used to mark windows of instructions within a TXN to skip HTM controller tracking [191, 192]. LogTM [39] uses similar escape actions to skip tracking non-TM code blocks during TXNs (traps, interrupts, etc.). Such coarse-grained pause/resume approach is typically used to execute sizeable code blocks of secondary software synchronization mechanisms within a TXN without aborting [193, 113, 194, 195, 196] and therefore conceptually differs from HinTM’s automated identification and fine-grained dif-

ferentiation of memory accesses that actually belong to a TXN, with the goal of conserving the HTM’s transactional capacity. Instead of our proposed *safe load/store* instructions, a compiler could use suspend/resume to wrap each load/store identified by HinTM’s static classification as safe. IBM’s System Z features a non-transactional store for debugging purposes [197]. Rock [198] and AMD’s ASF [199] are HTMs that featured non-transactional load instructions. Prior work [200] discussed different ways of using such load instructions to improve HTM performance, and seminal HTM work [38, 195] hinted at selectively using non-transactional instructions, but only at a conceptual level. TCC [201] marked stack references as local to avoid broadcasts and improve scalability. We are the first to comprehensively study and evaluate automatic generation (by the compiler or the hardware) of non-transactional memory operations within transactions to mitigate HTM capacity limitations. HinTM can use these instructions on ISAs that feature them, as instances of our proposed safe load/store instructions. Finally, neither suspend/resume nor non-transactional instructions can be used to mark dynamically identified safe loads/stores, which yield most of HinTM’s performance benefits.

Other metadata tracking and hint-based approaches. Memory access differentiation lies at the core of HinTM’s principle of operation. Such differentiation information could be encoded by leveraging systems featuring tagged memory [202, 203, 204, 205, 206] and capability-based approaches like CHERI [207] or Mondrian [208], using flexible metadata management frameworks like XMem [209] or MetaSys [210].

Other approaches addressing HTM capacity limitations. Pre-abort handlers [114] provide a software fallback path to attempt to save TXNs before aborting. For capacity aborts, a pre-abort handler would convert the aborting TXN into a critical section, preventing work loss, but still resulting in serialization. The technique can be used in conjunction with HinTM, which reduces capacity overflows. Cai et al. [51] study the effect of cache replacement policy on Intel TSX capacity aborts. SI-HTM [193] builds on the semantics

of snapshot isolation to expand POWER8 HTM’s limited transactional capacity by only buffering a TXN’s writeset. SI-HTM combines roll-back only TXNs [211] (ROT—TXNs that don’t track loads) with a software mechanism that delays all TXNs with a non-empty writeset to commit after all concurrent TXNs that started earlier have already committed. Rather than snapshot isolation, HinTM targets the stricter transactional model of 2-phase locking (2PL). Issa et al. [113] combine ROTs with a software technique to prevent the readset from occupying transactional capacity, but still achieve strict 2PL semantics, at the cost of tracking the TXN’s whole readset in software and touching it again before a TXN commits. HinTM does not require an additional TXN validation phase with extra instructions and memory accesses, and in general leaves TXN structure unmodified.

Dynamic page-based classification in non-TM contexts. Singh et al. [212] utilize static and dynamic memory access classification mechanisms to relax the ordering of thread-private memory accesses on sequentially consistent CPUs. OS page classification has been used by Cuesta et al. [213] to eliminate directory coherence tracking for data in private pages; in Valid/Invalid – Private/Shared (VIPS) protocol to construct a simple low-cost coherence mechanism [214]; and to inform intelligent data placement decisions in distributed memories [215, 216, 217, 218].

Beyond conventional HTMs. Our work targets “conventional” HTMs [219, 38] that impose rigid, hardware-bound limits on transaction sizes [50]. While all commercial implementations are instances of such conventional HTMs, the research literature is rich in advanced techniques to allow spilling transactional state from the limited hardware structures without aborting. “Large” HTMs (LTM [41], LogTM [39]) modify the cache hierarchy or coherence to track overflown state. “Unbounded” HTMs (UTM [41], VTM [37]) enhance hardware with software mechanisms to enable TXNs to not only exceed hardware structure capacities, but also survive context switches. Hybrid TMs [45, 48, 47] are founded on an underlying STM implementation and leverage HTM as an auxiliary mechanism to

improve performance, whenever possible. HinTM maintains the relative hardware simplicity of conventional HTMs and proposes minimally intrusive extensions that alleviate their rigid capacity constraints without sacrificing HTM’s performance advantages over STM.

6.2 Prior work related to memory system optimizations in server CPUs

High-end processors feature increasing core counts for throughput gains, with the aim of improving resource consolidation and TCO. However, commensurately scaling memory bandwidth availability using DDR technology alone is challenging, resulting in reducing the memory bandwidth availability per core. Hence, such manycore processors cannot be effectively utilized when handling memory-intensive workloads. Some recent high-end CPUs alleviate the memory bandwidth wall by deploying HBM [220, 221] or large die-stacked SRAM [222]. However, in addition to their high cost, such exotic solutions suffer from severely limited capacity [223].

Memory tiering. In the context of CXL, recent works have explored using CXL-attached memory as a secondary tier to expand both capacity and bandwidth [164, 58, 224, 225]. However, CXL-based tiering requires careful hardware-software co-design to mitigate the high serial link latency [226, 227]. As in SURGE, careful page placement across the memory tiers plays an important role in maximizing performance.

Memory bandwidth boosting. BATMAN [228] proposes spreading a workload’s dataset across planar and die-stacked DRAM to maximize aggregate memory bandwidth availability. Caption [229] aims to improve the performance of bandwidth-intensive applications by leveraging CXL memory devices for bandwidth expansion, and uses a linear Machine Learning model to perform bandwidth-aware page allocation across the memory tiers. Sehgal et al. [230] propose software-based weighted interleaving of pages to leverage the bandwidth of DDR and CXL memory, considering the workload characteristics and performance characteristics of memories to various memory read/write patterns.

Coaxial proposes complete replacement of the prevalent DDR-based memory system with a bandwidth-rich CXL-based memory system [54], but such radical approach can hurt performance under low system utilization. In contrast to all prior work deploying and managing a secondary memory tier to boost bandwidth, SURGE multiplexes existing (primarily I/O) interfaces to opportunistically boost memory bandwidth availability. SURGE is the first extensive study of the opportunities arising from the CXL-enabled fungibility of I/O and memory traffic, and the first system to leverage that capability to dynamically salvage idle I/O bandwidth, convert it to additional memory bandwidth for memory-bound workload acceleration.

Page placement. SURGE leverages controlled page placement as a way to split traffic between the tiered primary-salvage memory system. We found weighted first-touch page placement sufficient for the simple proof-of-concept deployment scenarios we evaluated. However, more complex cases with workload churn would require periodic re-evaluation of each workload’s traffic split and support for more sophisticated page placement and migration techniques. There are several such mechanisms in the literature, both software-based [231, 232, 233, 234, 235, 236] and hardware-assisted [120, 237].

6.3 Prior work related to Distributed DL Training performance and optimizations

DL training accelerator design. A vast body of prior work focuses on optimizing the individual accelerator node (GPU or custom accelerator) [163, 238, 239, 240, 241, 242, 243]. However, node design in isolation does not capture cluster-scale effects during distributed training and can lead to resource imbalance and cluster under-utilization. Maximizing cluster-wide performance and utilization requires a holistic design approach that jointly considers the impact of compute, network, memory, and workload parallelization strategy.

Cluster-scale DL training performance analysis. A growing body of work has examined the performance characteristics of large-scale distributed deep learning (DL) training across modern compute clusters. Jain *et al.* characterize the training performance of ResNet and Inception-based models on diverse CPU and GPU architectures, analyzing sensitivity to batch size, node count, and intra-node threading configurations [244]. Ren *et al.* extend this analysis to Natural Language Processing (NLP) and computer vision workloads on leading-edge systems, identifying communication bottlenecks within collective kernels and studying the impact of scaling node counts on end-to-end throughput [245]. Complementing these studies, Jeon *et al.* investigate multi-tenant GPU clusters running co-located training jobs, showing how locality-aware scheduling can substantially improve cluster utilization and overall performance [246].

Recent years have also seen the emergence of several modeling frameworks aimed at understanding and predicting distributed DL training performance, reflecting the increasing complexity of large-scale AI infrastructure. Broadly, these efforts fall into two categories: *analytical models* and *hybrid analytical–profile-based models*, each offering a distinct balance between modeling speed, accuracy, and generality.

Analytical frameworks such as Calculon [247] and vTrain [248] employ first-order formulations to estimate training performance by decomposing total iteration time into compute and communication components. Compute time is typically estimated as the ratio of floating-point operations to a device’s theoretical peak throughput, while communication time is modeled as the ratio of data volume to effective network bandwidth. These formulations enable rapid evaluation and lightweight design-space exploration but rely on simplifying assumptions—most notably, that all operators achieve ideal hardware utilization. To compensate for underutilization and architectural inefficiencies, these models introduce empirically tuned scaling factors (“magic factors”), which limit their generality and predictive robustness. Furthermore, purely analytical models lack the expressiveness needed to capture complex runtime phenomena such as overlapping communication, collective op-

eration optimizations, or pipeline scheduling effects that dominate performance in modern large-scale training.

Hybrid analytical–profile-based frameworks, including DistSim [150] and TrioSim [249], enhance modeling fidelity by integrating analytical abstractions with fine-grained runtime traces collected from real hardware. These traces capture kernel- and operator-level execution characteristics and are combined with analytical representations of data, model, and pipeline parallelism to achieve higher predictive accuracy. However, such fidelity comes at the cost of significant profiling overhead—requiring extensive execution of representative workloads to collect traces. This dependency makes them less practical for early-stage architectural exploration or for evaluating hypothetical hardware configurations. In addition, existing hybrid frameworks predominantly target homogeneous or symmetric interconnect topologies and offer limited support for modeling heterogeneous systems incorporating emerging technologies such as CXL-based disaggregated memory.

Bridging this gap, HeterSim [250] extends distributed training modeling into heterogeneous cluster environments and explores the performance implications of disaggregated memory technologies, including CXL. While it offers valuable insights into large-language-model (LLM) training under heterogeneous configurations, its analytical core continues to rely on simplified compute-delay approximations and the framework itself is not publicly available. Consequently, HeterSim remains limited in its ability to capture the nuanced and dynamic interactions between compute, memory, and interconnect subsystems that shape end-to-end distributed training performance.

In contrast, COMET is expressly designed to provide fast yet expressive cluster-scale performance analysis, enabling early-stage architectural exploration that captures the joint impact of training strategies and heterogeneous compute–memory–network resources on distributed DL performance. Its modular methodology allows lightweight analytical models to be combined with more detailed performance backends, offering a continuum of fidelity–versus–speed trade-offs within a single framework. As a result, COMET can

efficiently glean performance trends across vast design spaces—spanning parallelization strategies, interconnect topologies, and CXL-enabled disaggregated memory configurations—and then focus more detailed analysis on a narrowed set of promising design points. Given the current and projected scale of AI models and clusters comprising thousands of nodes, tools such as COMET are increasingly essential for rapid, flexible, and reliable joint exploration of algorithmic and architectural choices.

Cluster communication performance optimizations. Another family of work focuses on cluster communication performance, which is often a major performance determinant. Dong et al. propose an algorithm/system co-design methodology to improve training scalability, by alleviating network congestion with a congestion-less server architecture that uses novel communication collective algorithms and network topology [251]. Jiang et al. accelerate Deep Neural Network (DNN) training with a unified communication framework that dynamically adapts reduction collectives and parameter server tasks to best utilize CPU and bandwidth resources, and overlap communication latency [252]. Shah et al. and Sun et al. propose communication abstraction and improvised communication algorithms [253, 254].

Memory system impact on training. Memory system design and optimizations play a crucial role in the overall performance and efficiency of a distributed training cluster, as well as the cluster size required to train a given large model. Prior works such as Checkmate [255], ZeRO-DP [165], ZeRO-Offload [86], and ZeRO-Infinity [87] focus on reducing the per-node memory footprint required to train large DL models.

Training strategy auto-tuning frameworks. Prior works on auto-tuning frameworks (Alpa [256], AutoDDL [257], Rhino [258], FlexFlow [259]) focus on identifying the optimal parallelization strategy using a combination of data, operator and communication patterns. These frameworks perform an exhaustive design space search to identify the op-

timal parallelization strategy for a *given* cluster. While auto-tuning frameworks predict the optimal parallelization strategy for a DL model training task on a *specific* cluster, COMET develops a generic methodology to jointly evaluate the performance of a training strategy on an *arbitrary* cluster at an earlier design stage: when a cluster’s architecture is still malleable, i.e., going under design considerations.

CHAPTER 7

FUTURE RESEARCH DIRECTIONS

The growing demand for memory capacity and bandwidth stands in stark contrast to the slowing pace of technology scaling. With Dennard scaling and Moore’s Law nearing their limits, the continuous improvements traditionally achieved through conventional memory system enhancements have stagnated. Meanwhile, emerging interconnect technologies such as Compute Express Link (CXL) offer a promising path forward by enabling memory capacity expansion and bandwidth scaling through high-speed serial interfaces. Furthermore, the latest CXL 3.0 specification envisions a unified system fabric that seamlessly integrates compute and memory devices, laying the groundwork for a cohesive and flexible memory hierarchy. In parallel, research efforts in near-memory computing and in-memory computing architectures are gaining momentum, targeting the limitations of traditional memory systems and further expanding the design space for next-generation memory-centric computing.

At the same time, generative AI workloads have emerged as the dominant computational demand in modern datacenters, driving the rapid proliferation of accelerated AI servers that consume vast amounts of hardware and energy resources. In 2024, datacenters accounted for nearly 1.5% of global electricity consumption, a figure projected to rise to approximately 945 terawatt-hours (TWh) by 2030, with nearly half of this increase attributed to the growing deployment of accelerated AI servers [260]. Sustaining this exponential growth in AI computation necessitates major innovations in hardware and software system design to enhance datacenter performance, energy efficiency, and scalability.

7.1 Rethinking Monolithic Server Architectures

Current state-of-the-art datacenter servers employ monolithic system architectures that tightly integrate compute, memory, and networking resources to achieve the throughput required for cloud and AI workloads. Major infrastructure vendors are increasingly deploying “supernodes” and “superpods” that converge vast amounts of compute, memory, and interconnect resources within a single system to support foundation model training and large-scale inference. While these designs deliver exceptional raw performance, they are fundamentally rigid, statically configured, and often reliant on proprietary interconnects, resulting in high cost and power overheads.

Each monolithic system enforces a fixed ratio of CPUs, accelerators, and memory, preventing independent scaling of resources. As workloads in cloud datacenters diversify—from low-latency in-memory databases to AI/ML training that demands high bandwidth and parallelism—monolithic architectures cannot adapt dynamically to these heterogeneous requirements. This inflexibility forces organizations to maintain separate infrastructures for different workload types, leading to resource underutilization and significantly higher operational and management costs [261]. The limitations of monolithic designs are further exacerbated by the growing size of AI models. Models with trillions of parameters or Mixture-of-Experts (MoE) architectures often exceed the memory capacity of individual server nodes, necessitating distributed processing across interconnected supernodes. However, the bandwidth limitations of datacenter networks introduce significant communication overheads, resulting in poor scaling efficiency [262, 263, 264].

In addition, power delivery, cooling, and infrastructure costs present mounting barriers to scaling monolithic servers [8]. Their tightly coupled nature makes incremental expansion or hardware reuse impractical, locking datacenter operators into proprietary ecosystems and unsustainable energy profiles. As workloads continue to evolve, these systems lack the flexibility and adaptability to accommodate new workloads without costly redesigns or

full-system replacement.

The challenges posed by monolithic server designs—with rigid resource ratios, limited adaptability to diverse workloads, and constrained scalability—underscore the need for more flexible and modular infrastructure. Future server architectures must support dynamic allocation of compute, memory, and network resources, enabling operators to scale each component independently in response to evolving workload requirements.

Composable datacenter server systems represent a promising solution to these limitations. By disaggregating hardware resources and allowing them to be pooled and re-allocated dynamically, composable server architectures can improve resource utilization, cost-efficiency, and energy efficiency, while supporting the rapid integration of new accelerators and memory technologies. By aligning resource allocation with the demands of specific workloads, these systems achieve higher end-to-end efficiency across the memory and compute hierarchy.

7.2 Composable AI Systems: A Path Forward

Composable and disaggregated architectures provide a promising alternative to monolithic designs, enabling dynamic allocation of compute, memory, and networking resources to better match heterogeneous workload demands. By decoupling hardware components, such architectures facilitate fine-grained resource provisioning, improved utilization, and cost-efficient scalability across the datacenter. In the context of memory- and bandwidth-intensive AI workloads, this shift opens up new opportunities for jointly optimizing system cost, performance, and energy efficiency.

Scaling memory capacity through memory disaggregation is a fundamental step towards achieving composable architectures. Traditional monolithic systems scale by replicating entire nodes or by provisioning resources in fixed ratios, reinforcing rigidity and inefficiency. In contrast, disaggregated memory mitigates these constraints by allowing flexible capacity scaling, reducing communication overhead, and simplifying synchroniza-

tion across distributed workloads. Moreover, composable systems naturally accommodate heterogeneous compute, memory, and networking devices, enabling configurations that can evolve alongside rapidly changing AI and cloud workload requirements, without compromising local memory performance.

Recent advances in high-speed interconnect technologies, particularly Compute Express Link (CXL), make fine-grained disaggregation increasingly practical. CXL enables low-latency, cache-coherent communication between processors, accelerators, and memory devices over serial links, facilitating both memory and network disaggregation within datacenter infrastructures [265, 266, 267, 64]. By exposing compute, memory, and interconnect bandwidth as first-class, composable resources, CXL-based architectures provide the flexibility to dynamically share and reconfigure resources across nodes, better supporting memory- and bandwidth-intensive workloads, including large-scale AI training and inference.

Building on these foundations, we highlight three promising directions and system-level optimization opportunities for future composable AI systems. First, *memory expansion over serial I/O* links such as CXL can create shared memory pools backed by low-cost commodity DRAM [268]. Rather than over-provisioning per-node capacity, datacenters can expose large, logically unified memory pools that are elastically carved out and mapped to individual jobs. This model improves capacity utilization, lowers total cost of ownership, and enables AI workloads with large working sets to scale without incurring prohibitive communication overheads or requiring specialized high-bandwidth memory across all nodes.

Second, *bringing compute closer to data* represents an important shift towards non-von Neumann architectures. As AI models grow in size and sparsity, data movement increasingly dominates both performance and energy. In-memory and near-memory compute capabilities, when tightly integrated with CXL-attached memory devices, offer a path to offload bandwidth-intensive, data-parallel kernels (e.g., reductions, embedding lookups, and

simple linear algebra operations) closer to where data resides. Such near-data accelerators, orchestrated within a composable fabric, can reduce pressure on host memory channels and interconnects while enabling new execution models that couple host-side training loops with offloaded memory-side operators.

Third, fully realizing the benefits of disaggregated, heterogeneous memory requires *latency-aware data and memory allocation policies*. As AI workloads span multiple tiers—on-package memory, local DRAM, CXL-attached devices, and potentially rack-scale shared pools—software must reason explicitly about latency and bandwidth asymmetries. Latency-aware data placement policies can map latency-critical activations, gradients to faster tiers, while relegating less frequently accessed parameters, optimizer states, and checkpoint data to slower, more distant memory. Complementary latency-aware page-migration mechanisms can dynamically adapt these placements at runtime, monitoring access patterns and proactively migrating hot pages toward lower-latency tiers. Coordinated compiler, runtime, and OS support for such policies will be essential to translate the raw flexibility of composable hardware into predictable and efficient performance for large-scale AI training and inference.

7.3 CXL Shared Memory in Composable AI Systems

Shared memory forms a central mechanism for enabling composable AI server architectures. A shared-memory disaggregated system built using CXL Type-3 devices enables rack-scale AI training for massive models, providing seamless, low-cost, and scalable memory expansion while maintaining the flexibility needed to support heterogeneous hardware configurations. By allowing compute nodes to access and share disaggregated memory pools over high-speed interconnects, CXL-based shared memory systems eliminate the rigid capacity constraints of monolithic servers and pave the way for more elastic and resource-efficient training infrastructures.

In training large-scale AI models, frequent data movements—particularly coarse-grained

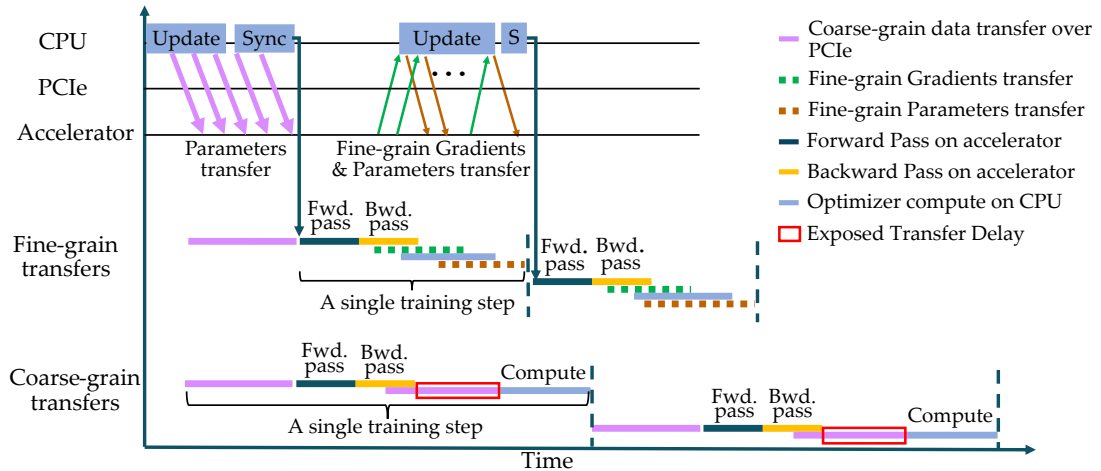


Figure 7.1: Illustration of compute-communication overlap for fine- vs. coarse-grained PCIe transfers.

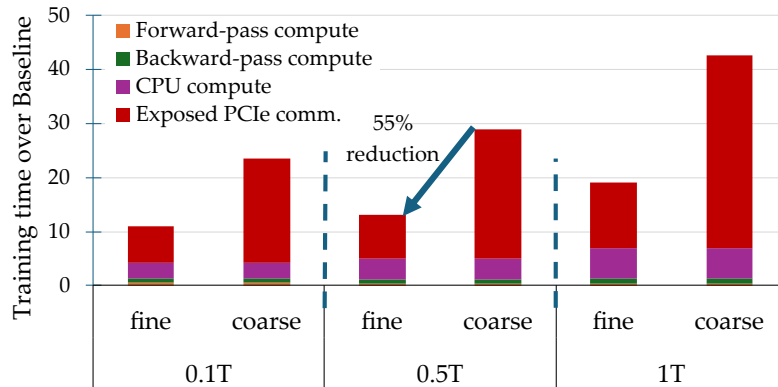


Figure 7.2: Estimated single iteration time with fine- vs. coarse-grained PCIe transfers through analytical modeling over baseline RDMA system.

transfers of parameters and gradients over serial interconnects such as PCIe—can incur significant delays ranging from tens to hundreds of milliseconds—often exceeding layer-wise computation times [269]. This imbalance hinders the ability to overlap computation and communication effectively, resulting in significant performance penalties.

To address this, disaggregated shared memory systems must support pipelined data transfers at cacheline granularity, enabling fine-grained overlap between computation and communication. Decomposing bulk data movement into small, pipelined cacheline-sized transfers significantly improves concurrency and mitigates serial link latency. As model sizes continue to grow, this fine-grained pipelining becomes increasingly essential for

maintaining high training throughput. Figure 7.1 illustrates the improved overlap achieved through fine-grained pipelining, while Figure 7.2 presents the estimated iteration times for 0.1-trillion, 0.5-trillion, and 1-trillion parameter LLMs relative to the baseline RDMA-based distributed training system. Employing fine-grained transfers reduces the overall training time by up to 55% compared to coarse-grained transfers for a 0.5-trillion parameter model.

Designing practical shared-memory for composable AI systems requires addressing performance, scalability, flexibility, and cost efficiency—each of which must be evaluated in the context of the workload and underlying hardware.

Performance. Achieving high performance is a key challenge, as composable AI systems must deliver competitive throughput without relying on specialized high-bandwidth interconnects. Unlike proprietary designs with dedicated intra-node links, composable architectures must operate efficiently over commodity interfaces such as PCIe and CXL. Fine-grained, pipelined data movement enables effective overlap between computation and communication, mitigating the bandwidth and latency limitations of serial interconnects. Decomposing large parameter and gradient transfers into cacheline-sized operations further improves iteration time by enhancing compute–communication concurrency, adapting dynamically to the needs of the workload.

Fine-grained, pipelined, cacheline-sized data transfers reduce PCIe transfer latency by improving the overlap between computation and data movement.

Scalability. As model sizes grow, memory scalability becomes more critical than compute scalability, since the memory footprint of large-scale training increases far faster than compute demand. Scaling memory across devices introduces consistency challenges among sharers. Traditional hardware-based cache coherence mechanisms scale poorly in such environments and can limit both system size and performance [270, 271, 272,

273]. Addressing this limitation requires architectural strategies that exploit the structured memory access patterns of deep learning workloads to manage consistency without costly invalidation-based coherence protocols.

Deep learning training workloads typically exhibit a well-defined “producer–consumer” relationship between parameters and gradients when training is partially offloaded to CPUs [269]. During backpropagation, accelerators produce gradients consumed by the host CPU for parameter updates, which then produces updated parameters used by the accelerators in the next iteration [86, 87]. This structured access pattern reduces multi-writer contention on shared cachelines, allowing shared memory usage without requiring full hardware-based coherence. By leveraging this insight, memory capacity can be scaled efficiently through proper memory isolation, avoiding the complexity and overhead of traditional coherence mechanisms while supporting high-performance training of large models across shared memory environments.

LLM training workloads exhibit structured producer–consumer relationships between gradients and parameters, reducing multi-writer contention and simplifying consistency management.

Flexibility. Composable AI systems must support rapidly evolving models, training paradigms, and heterogeneous hardware while maintaining performance and scalability. Traditional tightly coupled architectures lack this adaptability, often requiring extensive redesigns to accommodate new workloads. Overcoming this limitation calls for comprehensive hardware–software co-design across the stack—spanning frameworks, memory allocators, and runtimes to efficiently manage disaggregated resources. Abstracting disaggregation while preserving workload-specific control enables support for emerging paradigms such as Mixture-of-Experts (MoE) and multimodal models. In addition, flexible composable systems must balance backward compatibility with forward adaptability, empowering innovation without imposing system-level constraints and enabling scalable, democratized AI infrastructure.

Hardware–software co-design enables composable AI systems to flexibly support diverse workloads and emerging computational paradigms.

Cost Efficiency. Finally, cost efficiency is essential for democratizing access to large-scale AI infrastructure. The growing resource demands of modern AI systems have significantly increased deployment costs [274], creating barriers to affordability and broad accessibility. Utilizing commodity devices instead of proprietary hardware can substantially reduce costs while maintaining reliability and fault tolerance through component redundancy. Such cost-effective designs promote wider adoption and scalability, enabling sustainable growth of AI infrastructure across institutions and research environments.

In summary, realizing disaggregated memory–based composable architectures requires a holistic, workload-aware optimization of the memory hierarchy, dynamically tailoring memory access, allocation, and data movement to the computational needs of each workload. By harmonizing performance, scalability, flexibility, and cost efficiency across the memory and compute stack, these architectures overcome the rigidity of monolithic designs and establish a foundation for next-generation AI infrastructure capable of supporting diverse workloads at scale. Optimizing the memory system architecture in accordance with workload characteristics and underlying hardware enables higher efficiency and accelerates computation across large-scale AI applications.

7.4 Dynamic Page Migration for Disaggregated Memory Systems

In Chapter 4.2.2, we introduced a methodology to distribute memory traffic between main memory and extended salvage memory at the time of workload scheduling. While the initial page placement implementation demonstrates the potential for splitting memory traffic optimally, there remains substantial opportunity to enhance utility, flexibility, and system integration.

One promising direction is dynamic migration of pages between main memory and salvage memory at periodic intervals during execution. A key challenge in this approach is the overhead associated with tracking page hotness. Monitoring individual pages at memory-access granularity can introduce significant performance penalties. Moreover, frequent page migrations impose additional costs due to TLB shutdowns and page table updates.

Prior work on page migration in tiered memory systems [231, 275, 276, 236, 277, 278, 235, 279, 280, 281] has proposed software-based mechanisms for mitigating migration overheads. However, these solutions generally do not exploit the availability of idle memory bandwidth or opportunities to leverage unused I/O bandwidth to improve effective memory throughput.

Careful extensions at the hardware level, combined with Linux-based software enhancements, can reduce migration overhead and better utilize serial interconnects for memory expansion. Furthermore, hardware–software co-design approaches [282] can enable runtime page migration that minimizes the performance penalties associated with frequent migrations. By integrating dynamic page migration into memory-disaggregated, CXL-enabled composable systems, it becomes possible to maintain high utilization, improve memory bandwidth efficiency, and support the elastic memory demands of large-scale AI workloads.

CHAPTER 8

CONCLUSION

The memory system has emerged as a central determinant of both performance and cost in modern computing platforms. As computation capabilities continue to scale faster than memory capacity and bandwidth, the efficiency of parallel and distributed workloads is increasingly defined by how effectively the memory hierarchy delivers data to the compute substrate. Across the spectrum of hardware—from individual multicore processors to large-scale AI clusters—memory system design has become a primary factor shaping scalability, utilization, and energy efficiency. This thesis has explored this challenge holistically, presenting architectural and methodological advances that collectively promote memory systems as active enablers of performance rather than passive constraints.

At the chip level, this work introduced techniques to alleviate on-chip memory capacity limitations in Hardware Transactional Memory (HTM). By identifying and relaxing unnecessary tracking of memory operations through compiler- and runtime-guided classification, the proposed approach improves transactional concurrency without increasing hardware complexity. This study illustrated how fine-grained, workload-aware capacity management within the memory hierarchy can substantially enhance performance under constrained resources, reinforcing the broader theme that efficiency begins with locality and contention-aware design at the lowest system levels.

At the server level, the thesis examined the long-standing imbalance between processor core count growth and off-chip bandwidth provisioning. Modern CPUs statically partition bandwidth between I/O and memory channels, resulting in inefficient resource utilization across diverse workloads. The proposed architectural framework introduced a mechanism for dynamically sharing unused I/O bandwidth with memory traffic, enabled by emerging interconnect technologies such as Compute Express Link (CXL). This dynamic cross-

domain bandwidth reallocation redefines the off-chip interface as a flexible, composable resource, demonstrating how system-level coordination between I/O and memory can mitigate the bandwidth wall that increasingly limits server performance.

At the cluster level, the work extended this holistic design perspective to distributed training systems, addressing the interplay between workload behavior and large-scale resource provisioning. The study developed a systematic methodology to jointly analyze model parallelization strategies, compute density, memory hierarchy parameters, and interconnect characteristics to identify balanced configurations for scalable performance. The resulting insights highlight that effective cluster design arises not from isolated optimization of components, but from the integrated consideration of compute, memory, and network resources as a unified system, capable of sustaining the ever-growing computational and memory demands of large-scale AI workloads.

Together, these studies advance a unified vision of holistic memory system optimization—a view that spans from on-chip management of limited capacity, to server-level bandwidth reconfiguration, to cluster-scale coordination of distributed memory resources. Each level reinforces the other, underscoring the importance of co-optimizing compute, memory, and communication as interdependent components of a shared performance fabric.

Looking forward, the principles developed in this thesis open several promising research directions. Future systems will increasingly rely on memory disaggregation, composable fabrics, and heterogeneous compute environments, where dynamic adaptation of memory capacity and bandwidth will be key to sustained scalability. Building upon the methodologies and insights presented here, continued work can extend these ideas toward energy-aware memory scheduling, real-time workload adaptation, and multi-cluster optimization frameworks.

Ultimately, this thesis contributes to a broader rethinking of the memory system’s role—from a limiting factor in performance scaling to a cohesive, designable resource that unifies hardware and workload behavior across the compute continuum.

REFERENCES

- [1] Robert A. Lee, “*Google Usage Statistics 2025: Key Trends and Data Insights*”, <https://sqmagazine.co.uk/google-usage-statistics>, Published: Sep 30, 2025. Last Accessed: Oct 11, 2025.
- [2] Maria Younus and Rabia Mahmood, “*Facebook Stats & Facts 2025: Users, Engagement, and Trends*”, <https://www.tekrevol.com/blogs/facebook-stats-facts>, Published: July 9, 2025. Last Accessed: Oct 11, 2025.
- [3] Barry Elad, “*OpenAI Statistics 2025: Adoption, Integration & Innovation*”, <https://sqmagazine.co.uk/openai-statistics/>, Published: October 7, 2025. Last Accessed: Oct 11, 2025.
- [4] K. Alaamer, “*This is the state of play in the global data centre gold rush*”, <https://www.weforum.org/stories/2025/04/data-centre-gold-rush-ai>, Published: Apr 22, 2025. Last Accessed: Oct 11, 2025.
- [5] Jesse Noffsinger, Mark Patel, and Pankaj Sachdeva, with Arjita Bhan, Haley Chang, and Maria Goodpaster, “*The cost of compute: A \$7 trillion race to scale data centers*”, <https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-cost-of-compute-a-7-trillion-dollar-race-to-scale-data-centers>, Published: Apr 28, 2025. Last Accessed: Oct 11, 2025.
- [6] Fortune Business Insights, “*Data Center Market Size, Share, And Growth Report [2032]*”, <https://www.fortunebusinessinsights.com/data-center-market-109851>, Last Updated: Sep 22, 2025. Last Accessed: Oct 11, 2025.
- [7] Katerina Henjes, “*The growing demand for data centers in the U.S.*”, <https://www.aaas.org/news/growing-demand-data-centers-us>, Published: Aug 21, 2025. Last Accessed: Oct 11, 2025.
- [8] C. Tang, “*Meta’s hyperscale infrastructure: Overview and insights*,” *Commun. ACM*, vol. 68, no. 2, pp. 52–63, 2025.
- [9] Bhargs Srivathsan, Marc Sorel, and Pankaj Sachdeva, with Arjita Bhan, Haripreet Batra, Raman Sharma, Rishi Gupta, and Surbhi Choudhary, “*AI power: Expanding data center capacity to meet growing demand*”, <https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/ai-power-expanding-data-center-capacity-to-meet-growing-demand>, Published: Oct 29, 2024. Last Accessed: Oct 11, 2025.
- [10] Steven Woo, Wendy Elsasser, and Taeksang Song, “*Scaling DRAM Technology to Meet Future Demands: Challenges and Opportunities*”, <https://www.rambus.com/>

wp-content/uploads/2025/07/ScalingDRAMTechnology-ISCA2025_Tutorial.pdf,
Published: June 22, 2025. Last Accessed: Oct 11, 2025.

- [11] H. Li et al., “Pond: Cxl-based memory pooling systems for cloud platforms,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, 2023, pp. 574–587.
- [12] L. James, “AI Data Centers Are Swallowing the World’s Memory and Storage Supply, Setting the Stage for a Pricing Apocalypse That Could Last a Decade”, <https://www.tomshardware.com/pc-components/storage/perfect-storm-of-demand-and-supply-driving-up-storage-costs>, Published: Oct 3, 2025. Last Accessed: Oct 11, 2025.
- [13] A. Daglis, “Network-compute co-design for distributed in-memory computing,” Ph.D. dissertation, EPFL, Switzerland, 2018.
- [14] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, “High performance rdma-based mpi implementation over infiniband,” in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS ’03, San Francisco, CA, USA: Association for Computing Machinery, 2003, pp. 295–304, ISBN: 1581137338.
- [15] C. Guo et al., “Rdma over commodity ethernet at scale,” in *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016, pp. 202–215.
- [16] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, “Scale-out numa,” in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2014, pp. 3–18.
- [17] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012, ISBN: 978-0-12-383872-8.
- [18] Lauro Rizzatti, “Generative AI and memory wall: A wakeup call for IC industry”, <https://www.edn.com/generative-ai-and-memory-wall-a-wakeup-call-for-ic-industry>, Published: Aug 15, 2023. Last Accessed: Oct 13, 2025.
- [19] G. E. Moore, “Cramming more components onto integrated circuits,” *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, 1998.
- [20] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *Proc. IEEE*, vol. 87, no. 4, pp. 668–678, 1999.

- [21] K. Itoh, Y. Nakagome, S. Kimura, and T. Watanabe, “Limitations and challenges of multigigabit DRAM chip design,” *IEEE J. Solid State Circuits*, vol. 32, no. 5, pp. 624–634, 1997.
- [22] F. T. Hady, A. P. Foong, B. Veal, and D. Williams, “Platform storage performance with 3d xpoint technology,” *Proc. IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.
- [23] Jim Handy, “Why ‘emerging’ memories have not succeeded – yet”, <https://www.eenewseurope.com/en/why-emerging-memories-have-not-succeeded-yet>, Published: September 16, 2022. Last Accessed: Oct 11, 2025.
- [24] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, “Clio: A hardware-software co-designed disaggregated memory system,” in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, 2022, pp. 417–433.
- [25] E. Desjardins, “Jedec publishes new ddr5 standard for advancing next-generation high performance computing systems,” *JEDEC*, 2020.
- [26] Micron Technology Inc., “DDR5 SDRAM Datasheet,” 2022.
- [27] R. Rooney and N. Koyle, “Micron® ddr5 sdram: New features,” *Micron Technology Inc., Tech. Rep*, 2019.
- [28] B. Nitin et al., “DDR5 design challenges,” in *2018 IEEE 22nd Workshop on Signal and Power Integrity (SPI)*, 2018, pp. 1–4.
- [29] P. Stanley-Marbell, V. C. Cabezas, and R. P. Luijten, “Pinned to the walls — impact of packaging and application properties on the memory and power walls,” in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011, pp. 51–56.
- [30] Q. Zhu, S. Venkataraman, C. Ye, and A. Chandrasekhar, “Package design challenges and optimizations in density efficient (intel® xeon® processor d) soc,” in *2016 IEEE Electrical Design of Advanced Packaging and Systems (EDAPS)*, 2016, pp. 47–49.
- [31] S. Lie, “Inside the cerebras wafer-scale cluster,” *IEEE Micro*, vol. 44, no. 3, pp. 49–57, 2024.
- [32] C. He et al., “Wafer-Scale AI Compute: A System Software Perspective,” in *USENIX publications*, USENIX, 2025.

- [33] A. Jain, D. K. Kadiyala, and A. Daglis, “Safety hints for htm capacity abort mitigation,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 206–219.
- [34] D. K. Kadiyala and A. Daglis, *Pushing the memory bandwidth wall with cxl-enabled idle i/o bandwidth harvesting*, 2025. arXiv: 2511.12349 [cs.AR].
- [35] D. K. Kadiyala, S. Rashidi, T. Heo, A. R. Bambhaniya, T. Krishna, and A. Daglis, “COMET: A comprehensive cluster design methodology for distributed deep learning training,” *CoRR*, vol. abs/2211.16648, 2022. arXiv: 2211.16648.
- [36] D. Kadiyala, S. Rashidi, T. Heo, A. Bambhaniya, T. Krishna, and A. Daglis, “Leveraging memory expansion to accelerate large-scale dl training,” in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 292–294.
- [37] R. Rajwar, M. Herlihy, and K. K. Lai, “Virtualizing Transactional Memory,” in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 494–505.
- [38] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, 1993, pp. 289–300.
- [39] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “LogTM: log-based transactional memory,” in *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 254–265.
- [40] W. Chuang et al., “Unbounded page-based transactional memory,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006, pp. 347–358.
- [41] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded Transactional Memory,” in *Proceedings of the 11th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 316–327.
- [42] H. Q. Le et al., “Transactional memory support in the IBM POWER8 processor,” *IBM J. Res. Dev.*, vol. 59, no. 1, 2015.
- [43] S. Park, M. Prvulovic, and C. J. Hughes, “PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory,” in *Proceedings of the 22nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 285–296.
- [44] Intel Corporation, *Restricted transactional memory overview*, Sep. 2022.

- [45] L. Dalessandro, M. F. Spear, and M. L. Scott, “NOrec: streamlining STM by abolishing ownership records,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010, pp. 67–78.
- [46] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun, “The OpenTM Transactional Application Programming Interface,” in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007, pp. 376–387.
- [47] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. D. Nguyen, “Hybrid transactional memory,” in *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006, pp. 209–220.
- [48] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, “Hybrid transactional memory,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006, pp. 336–346.
- [49] C. Jacobi, T. J. Slegel, and D. F. Greiner, “Transactional memory architecture and implementation for IBM system Z,” in *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, IEEE Computer Society, 2012, pp. 25–36.
- [50] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, “Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8,” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 144–157.
- [51] Z. Cai, S. M. Blackburn, and M. D. Bond, “Understanding and Utilizing Hardware Transactional Memory Capacity,” in *Proc. 2021 ACM SIGPLAN Int. Symp. on Memory Management (STMM)*, 2021.
- [52] W. Hasenplaugh, A. Nguyen, and N. Shavit, “Quantifying the Capacity Limitations of Hardware Transactional Memory,” *7th Workshop on the Theory of Transactional Memory (WTTM)*, 2015.
- [53] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.
- [54] A. Cho, A. Saxena, M. Qureshi, and A. Daglis, “COAXIAL: A CXL-Centric Memory System for Scalable Servers,” in *Proceedings of the 2024 ACM/IEEE Conference on Supercomputing (SC)*, 2024, p. 95.
- [55] A. Chatterjee, P. Ellervee, V. J. M. III, J.-C. Park, K.-w. Choi, and K. Puttaswamy, “System Level Power-Performance Trade-Offs in Embedded Systems Using Volt-

- age and Frequency Scaling of Off-Chip Buses and Memory,” in *Proceedings of the 15th International Symposium on System Synthesis*, 2002, pp. 225–230.
- [56] Q. Zhu, S. Venkataraman, C. Ye, and A. Chandrasekhar, “Package design challenges and optimizations in density efficient (Intel® Xeon® processor D) SoC,” in *2016 IEEE Electrical Design of Advanced Packaging and Systems (EDAPS)*, 2016.
- [57] P. Esmaili-Dokht et al., “A Mess of Memory System Benchmarking, Simulation and Application Profiling,” in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 136–152.
- [58] M. Vuppapapati and R. Agarwal, “Tiered Memory Management: Access Latency is the Key!” In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, 2024, pp. 79–94.
- [59] P. Stanley-Marbell, V. C. Cabezas, and R. P. Luijten, “Pinned to the walls: impact of packaging and application properties on the memory and power walls,” in *Proceedings of the 2011 International Symposium on Low Power Electronics and Design*, 2011, pp. 51–56.
- [60] Backblaze, “The ssd edition: 2023 drive stats mid-year review,” *Accessed: Oct 2025*, 2023.
- [61] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. I. T. Rowstron, “Migrating server storage to SSDs: analysis of tradeoffs,” in *Proceedings of the 2009 EuroSys Conference*, 2009, pp. 145–158.
- [62] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, M. Allman, Ed., ACM, 2010, pp. 267–280.
- [63] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” *Comput. Commun. Rev.*, vol. 40, no. 1, pp. 92–99, 2010.
- [64] X. Zhang et al., “Drack: A cxl-disaggregated rack architecture to boost inter-rack communication,” in *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, D. Altinbükten and R. Stutsman, Eds., USENIX Association, 2025, pp. 1261–1279.
- [65] W. Fedus, B. Zoph, and N. Shazeer, “Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity,” *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

- [66] J. Lin et al., “M6-10T: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining,” *CoRR*, vol. abs/2110.03888, 2021. arXiv: 2110.03888.
- [67] B. Smith and G. Linden, “Two decades of recommender systems at amazon.com,” *IEEE Internet Computing*, vol. 21, no. 3, pp. 12–18, 2017.
- [68] Meta, “The llama 4 herd: The beginning of a new era of natively multimodal ai innovation,” April 5, 2025.
- [69] H. Touvron et al., *Llama: Open and efficient foundation language models*, 2023. arXiv: 2302.13971 [cs.CL].
- [70] G. T. et. al., “Gemini: A family of highly capable multimodal models,” *CoRR*, vol. abs/2312.11805, 2023. arXiv: 2312.11805.
- [71] OpenAI, “GPT-4 technical report,” *CoRR*, vol. abs/2303.08774, 2023. arXiv: 2303.08774.
- [72] P. Kharya and A. Alvi, “Using deepspeed and megatron to train megatron-turing nlg 530b, the world’s largest and most powerful generative language model,” *Accessed: June 2024*, 2021.
- [73] A. G. et. al., *The llama 3 herd of models*, 2024. arXiv: 2407.21783 [cs.AI].
- [74] N. Shazeer et al., “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, OpenReview.net, 2017.
- [75] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.
- [76] S. Li et al., “Pytorch distributed: Experiences on accelerating data parallel training,” *CoRR*, vol. abs/2006.15704, 2020. arXiv: 2006.15704.
- [77] N. Shazeer et al., “Mesh-tensorflow: Deep learning for supercomputers,” in *Advances in Neural Information Processing Systems*, vol. 31, 2018.
- [78] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using model parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.

- [79] Y. Huang et al., “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” *CoRR*, vol. abs/1811.06965, 2018. arXiv: 1811.06965.
- [80] D. Narayanan et al., “Efficient large-scale language model training on gpu clusters using megatron-lm,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
- [81] D. Narayanan et al., “Pipedream: Generalized pipeline parallelism for dnn training,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19, Huntsville, Ontario, Canada, 2019, pp. 1–15, ISBN: 9781450368735.
- [82] Y. Zhao, R. Varma, C.-C. Huang, S. Li, M. Xu, and A. Desmaison, *Introducing PyTorch Fully Sharded Data Parallel (FSDP) API*, <https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/>, [Online; accessed 28-June-2022], 2022.
- [83] P. Belevich et al., *Training a 1 Trillion Parameter Model With PyTorch Fully Sharded Data Parallel on AWS*, <https://medium.com/pytorch/training-a-1-trillion-parameter-model-with-pytorch-fully-sharded-data-parallel-on-aws-3ac13aa96cff>, [Online; accessed 28-June-2022], 2022.
- [84] M. Ott, S. Shleifer, M. Xu, P. Goyal, Q. Duval, and V. Caggiano, *Fully Sharded Data Parallel: faster AI training with fewer GPUs*, <https://engineering.fb.com/2021/07/15/open-source/fsdp/>, [Online; accessed 28-June-2022], 2021.
- [85] D. Lepikhin et al., “Gshard: Scaling giant models with conditional computation and automatic sharding,” in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021.
- [86] J. Ren et al., “ZeRO-Offload: Democratizing Billion-Scale model training,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, Jul. 2021, pp. 551–564, ISBN: 978-1-939133-23-6.
- [87] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, “Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21, St. Louis, Missouri, 2021, ISBN: 9781450384421.
- [88] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, “AI and memory wall,” *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024.
- [89] The Register, “CXL absorbs OpenCAPI on the road to interconnect dominance,” 2022.

- [90] S. Tamimi, F. Stock, A. Koch, A. Bernhardt, and I. Petrov, “An evaluation of using ccix for cache-coherent host-fpga interfacing,” in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–9.
- [91] G.-Z. Consortium et al., “About-gen-z consortium,” *Accessed: May*, vol. 11, 2018.
- [92] CXL Consortium, “CXL Consortium Members,”
- [93] D. D. Sharma, “Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing,” in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022, pp. 5–12.
- [94] D. D. Sharma, “Novel composable and scaleout architectures using compute express link,” *IEEE Micro*, vol. 43, no. 2, pp. 9–19, 2023.
- [95] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, “Memory pooling with cxl,” *IEEE Micro*, vol. 43, no. 2, pp. 48–57, 2023.
- [96] K. Lepak, G. Talbot, S. White, N. Beck, S. Naffziger, S. FELLOW, et al., “The next generation amd enterprise server product architecture,” *IEEE hot chips*, vol. 29, p. 182, 2017.
- [97] A. Smith et al., “Realizing the AMD exascale heterogeneous processor vision : Industry product,” in *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, IEEE, 2024, pp. 876–889.
- [98] C. Pearson, *Interconnect bandwidth heterogeneity on amd mi250x and infinity fabric*, 2023. arXiv: 2302.14827 [cs .DC].
- [99] AMD, “Xgmi configuration — amd instinct virtualization driver,” *Accessed: Oct 2025*,
- [100] G. Schieffer, R. Shi, S. Markidis, A. Herten, J. Faj, and I. Peng, “Understanding data movement in AMD multi-gpu systems with infinity fabric,” in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, November 17-22, 2024*, IEEE, 2024, pp. 567–576.
- [101] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, “Intel® quickpath interconnect architectural features supporting scalable system architectures,” in *IEEE 18th Annual Symposium on High Performance Interconnects, HOTI 2010, Google Campus, Mountain View, California, USA, August 18-20, 2010*, F. Petrini, D. Abts,

- R. Brightwell, P. Balaji, and C. Minkenberg, Eds., IEEE Computer Society, 2010, pp. 1–6.
- [102] NVIDIA, “Nvidia nvl link and nvl link switch,” *Accessed: Oct 2025*,
- [103] C. Lattner and V. S. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–88.
- [104] Arm Limited, *ARM Architecture Reference Manual: Coprocessor Instructions*, <https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/The-Instruction-Sets/Coprocessor-instructions>.
- [105] C. Villavieja et al., “DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory,” in *Proceedings of the 20th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2011, pp. 340–349.
- [106] A. Chang and M. F. Mergen, “801 Storage: Architecture and Programming,” *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 28–50, 1988.
- [107] J. Chung et al., “Tradeoffs in transactional memory virtualization,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006, pp. 371–381.
- [108] J. Renau et al., *SESC simulator*, <http://sesc.sourceforge.net>, Jan. 2005.
- [109] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, “CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories,” *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, 14:1–14:25, 2017.
- [110] *LLVM Project Github Source*, <https://github.com/llvm/llvm-project/tree/llvmorg-9.0.1>.
- [111] L. Yen, S. C. Draper, and M. D. Hill, “Notary: Hardware techniques to enhance signatures,” in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 234–245.
- [112] M. Becker and S. Chakraborty, “Measuring Software Performance on Linux,” *CoRR*, vol. abs/1811.01412, 2018.
- [113] S. Issa, P. Felber, A. Matveev, and P. Romano, “Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume: Power8 tm,” *Distrib. Comput.*, vol. 33, no. 3–4, pp. 327–348, Jun. 2020.

- [114] S. Park, C. J. Hughes, and M. Prvulovic, “Transactional pre-abort handlers in hardware transactional memory,” in *Proceedings of the 27th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2018, 33:1–33:11.
- [115] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford Transactional Applications for Multi-Processing,” in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization (IISWC)*, 2008, pp. 35–46.
- [116] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “DHTM: Durable Hardware Transactional Memory,” in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465.
- [117] Transaction Processing Performance Council, *TPC Benchmark C*, http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, 2010.
- [118] Steve Zagorianakos, *RISC-V Instruction Set Manual: Standard Extension for Transactional Memory*, Dec. 2018.
- [119] D. S. Berger, Y. Zhong, P. Zardoshti, S. Teng, F. Kazhemiaka, and R. Fonseca, “Octopus: Scalable Low-Cost CXL Memory Pooling,” *CoRR*, vol. abs/2501.09020, 2025.
- [120] A. Cho and A. Daglis, “Starnuma: Mitigating NUMA challenges with memory pooling,” in *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*, IEEE, 2024, pp. 997–1012.
- [121] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: improving resource efficiency at scale,” in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 450–462.
- [122] C. Delimitrou and C. Kozyrakis, “Quasar: resource-efficient and QoS-aware cluster management,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 127–144.
- [123] C. Delimitrou and C. Kozyrakis, “Paragon: QoS-aware scheduling for heterogeneous datacenters,” in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, 2013, pp. 77–88.
- [124] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the 2015 EuroSys Conference*, 2015, 18:1–18:17.

- [125] Rambus, *Compute Express Link (CXL) Controller IP*, <https://www.rambus.com/interface-ip/cxl/>, 2024.
- [126] H. Mujtaba, “AMD EPYC Bergamo ‘Zen 4C’ CPUs Being Deployed In 1H 2023 To Tackle Arm CPUs, Instinct MI300 APU Back In Labs,” 2022.
- [127] H. Mujtaba, “Intel Granite Rapids & Sierra Forest Xeon CPU Detailed In Avenue City Platform Leak: Up To 500W TDP & 12-Channel DDR5,” 2023.
- [128] D. Sanchez and C. Kozyrakis, “Zsim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13, Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 475–486, ISBN: 9781450320795.
- [129] D. D. Sharma, R. Blankenship, and D. S. Berger, “An introduction to the compute express link (CXL) interconnect,” *ACM Comput. Surv.*, vol. 56, no. 11, 290:1–290:37, 2024.
- [130] D. D. Sharma, “Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing,” in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022, pp. 5–12.
- [131] H. Kasture and D. Sánchez, “Tailbench: a benchmark suite and evaluation methodology for latency-critical applications,” in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 3–12.
- [132] Standard Performance Evaluation Corporation, “SPEC CPU2017 Benchmark Suite,”
- [133] S. Beamer, K. Asanovic, and D. A. Patterson, “The GAP benchmark suite,” *CoRR*, vol. abs/1508.03619, 2015. arXiv: 1508.03619.
- [134] M. Vemmou, A. Cho, and A. Daglis, “Patching up Network Data Leaks with Sweeper,” in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 464–479.
- [135] D. Gallatin, “The “other” FreeBSD optimizations used by Netflix to serve video at 800Gb/s from a single server,” *EuroBSDCon*, 2022.
- [136] S. Smith et al., “Using deepspeed and megatron to train megatron-turing NLG 530b, A large-scale generative language model,” *CoRR*, vol. abs/2201.11990, 2022. arXiv: 2201.11990.
- [137] T. Wolf et al., “Huggingface’s transformers: State-of-the-art natural language processing,” *CoRR*, vol. abs/1910.03771, 2019. arXiv: 1910.03771.

- [138] D. V. Gupta, A. S. A. Ishaqui, and D. K. Kadiyala, “Geode: A zero-shot geospatial question-answering agent with explicit reasoning and precise spatio-temporal retrieval,” *CoRR*, vol. abs/2407.11014, 2024. arXiv: 2407.11014.
- [139] D. Grechishnikova, “Transformer neural network for protein-specific de novo drug generation as a machine translation problem,” *Scientific Reports*, vol. 11, no. 1, p. 321, Jan. 2021.
- [140] J. Vamathevan et al., “Applications of machine learning in drug discovery and development,” *Nature Reviews Drug Discovery*, vol. 18, no. 6, pp. 463–477, Jun. 2019.
- [141] D. Amodei et al., “Deep speech 2 : End-to-end speech recognition in english and mandarin,” in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA, 20–22 Jun 2016, pp. 173–182.
- [142] Y. Ren et al., “Fastspeech: Fast, robust and controllable text to speech,” in *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [143] R. Thoppilan et al., “Lamda: Language models for dialog applications,” *CoRR*, vol. abs/2201.08239, 2022. arXiv: 2201.08239.
- [144] M. Naumov et al., “Deep learning recommendation model for personalization and recommendation systems,” *CoRR*, vol. abs/1906.00091, 2019. arXiv: 1906.00091.
- [145] S. Zhang, L. Yao, A. Sun, and Y. Tay, “Deep learning based recommender system: A survey and new perspectives,” *ACM Comput. Surv.*, vol. 52, no. 1, Feb. 2019.
- [146] N. P. Jouppi et al., *Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings*, 2023. arXiv: 2304.01433 [cs.AR].
- [147] D. Narayanan et al., “Efficient large-scale language model training on GPU clusters using megatron-lm,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds.
- [148] A. Vaswani et al., “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [149] A. Castelló, M. Catalán, M. F. Dolz, J. I. Mestre, E. S. Quintana-Ortí, and J. Dato, “Performance modeling for distributed training of convolutional neural networks,” in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2021, pp. 99–108.

- [150] G. Lu et al., *Distsim: A performance model of large-scale hybrid distributed dnn training*, 2023. arXiv: 2306.08423 [cs.DC].
- [151] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, “Applying the roofline model,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 76–85.
- [152] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
- [153] N. Ding and S. Williams, “An instruction roofline model for gpus,” in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18.
- [154] K. Z. Ibrahim, S. Williams, and L. Oliker, “Performance analysis of gpu programming models using the roofline scaling trajectories,” in *Benchmarking, Measuring, and Optimizing*, W. Gao, J. Zhan, G. Fox, X. Lu, and D. Stanzione, Eds., Cham: Springer International Publishing, 2020, pp. 3–19, ISBN: 978-3-030-49556-5.
- [155] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, “Xsp: Across-stack profiling and analysis of machine learning models on gpus,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 326–327.
- [156] T. Miao et al., “Md-roofline: A training performance analysis model for distributed deep learning,” in *2022 IEEE Symposium on Computers and Communications (ISCC)*, 2022, pp. 1–8.
- [157] M. Wang, L. Dong, W. Dong, and Z. Lv, “A deep neural network optimization strategy based on roofline model,” in *2022 3rd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, 2022, pp. 103–109.
- [158] Y. Wang, C. Yang, S. Farrell, Y. Zhang, T. Kurth, and S. Williams, “Time-based roofline for deep learning performance analysis,” in *2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS)*, 2020, pp. 10–19.
- [159] C. Yang, “8 steps to 3.7 tflop/s on NVIDIA V100 GPU: roofline analysis and other tricks,” *CoRR*, vol. abs/2008.11326, 2020. arXiv: 2008.11326.
- [160] F. Checconi, J. J. Tithi, and F. Petrini, *Ridgeline: A 2d roofline model for distributed systems*, 2022. arXiv: 2209.01368 [cs.DC].

- [161] C. Yang, Y. Wang, T. Kurth, S. Farrell, and S. Williams, “Hierarchical roofline performance analysis for deep learning applications,” in *Intelligent Computing*, K. Arai, Ed., Cham: Springer International Publishing, 2021, pp. 473–491, ISBN: 978-3-030-80126-7.
- [162] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-sim: An extensible simulation framework for validated GPU modeling,” in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, 2020, pp. 473–486.
- [163] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “A systematic methodology for characterizing scalability of dnn accelerators using scale-sim,” in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 58–68.
- [164] H. A. Maruf et al., “TPP: Transparent Page Placement for CXL-Enabled Tiered Memory,” *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, 2023.
- [165] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “Zero: Memory optimizations toward training trillion parameter models,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–16.
- [166] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, “Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms,” in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2020, pp. 81–92.
- [167] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, “Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale,” *CoRR*, vol. abs/2303.14006, 2023. arXiv: 2303.14006.
- [168] NVIDIA, *NVIDIA Collective Communication Library (NCCL)*, <https://developer.nvidia.com/nccl>.
- [169] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “Garnet: A detailed on-chip network model inside a full-system simulator,” in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 33–42.
- [170] G. F. Riley and T. R. Henderson, “The *ns-3* network simulator,” in *Modeling and Tools for Network Simulation*, 2010, pp. 15–34.

- [171] T. Khan, S. Rashidi, S. Sridharan, P. Shurpali, A. Akella, and T. Krishna, “Impact of roce congestion control policies on distributed training of dnns,” in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022, pp. 39–48.
- [172] S. Rashidi et al., “Enabling compute-communication overlap in distributed deep learning training platforms,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 540–553.
- [173] S. Rashidi, W. Won, S. Srinivasan, S. Sridharan, and T. Krishna, “Themis: A network bandwidth-aware collective scheduling policy for distributed training of dl models,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22, New York, New York, 2022, pp. 581–596, ISBN: 9781450386104.
- [174] NVIDIA, “Nvidia dgx a100,” *Cloud & Data Center DGX A100*, Aug. 2022, [Online; accessed on 26-October-2022].
- [175] P. Ganesh et al., “Compressing Large-Scale Transformer-Based Models: A Case Study on BERT,” *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 1061–1080, Sep. 2021. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00413/1964006/tacl_a_00413.pdf.
- [176] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter, “Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy,” *IBM Journal of Research and Development*, vol. 63, no. 6, 1:1–1:11, 2019.
- [177] P. Covington, J. Adams, and E. Sargin, “Deep neural networks for youtube recommendations,” in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys ’16, Boston, Massachusetts, USA, 2016, pp. 191–198, ISBN: 9781450340359.
- [178] U. Gupta et al., “The architectural implications of facebook’s dnn-based personalized recommendation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.
- [179] M. Naumov et al., “Deep learning training in facebook data centers: Design of scale-up and scale-out systems,” *CoRR*, vol. abs/2003.09518, 2020. arXiv: 2003.09518.
- [180] C. A. Gomez-Uribe and N. Hunt, “The netflix recommender system: Algorithms, business value, and innovation,” *ACM Trans. Manage. Inf. Syst.*, vol. 6, no. 4, Dec. 2016.
- [181] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

- [182] M. Wang et al., “Characterizing deep learning training workloads on alibaba-pai,” in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 189–202.
- [183] G. Zhou et al., “Deep interest evolution network for click-through rate prediction,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 5941–5948, Jul. 2019.
- [184] S. Rashidi et al., “Scalable distributed training of recommendation models: An astra-sim + ns3 case-study with tcp/ip transport,” in *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2020, pp. 33–42.
- [185] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, “Understanding training efficiency of deep learning recommendation models at scale,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 802–814.
- [186] D. Mudigere et al., “Software-hardware co-design for fast and scalable training of deep learning recommendation models,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, New York, New York, 2022, pp. 993–1011, ISBN: 9781450386104.
- [187] F. Lambert, “Tesla releases new deep-dive presentations on its dojo ai supercomputer,” Aug. 2022, <https://electrek.co/2022/08/24/tesla-deep-dive-presentations-dojo-ai-supercomputer/>.
- [188] “Tpu v4 system architecture,” *Cloud TPU Documentation Guides*, [Online; accessed on 27-October-2022].
- [189] L. Feng, S. Zheng, Z. Wang, W. Fu, and J. H. Zeng, “Using chakra execution traces for benchmarking and network performance optimization,” *Engineering at Meta*, 2023.
- [190] S. Sridharan et al., “Chakra: Advancing performance benchmarking and co-design using standardized execution traces,” *CoRR*, vol. abs/2305.14516, 2023. arXiv: 2305.14516.
- [191] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Q. Le, “Robust architectural support for transactional memory in the power architecture,” in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 225–236.
- [192] I. Corporation, *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*, 2020.

- [193] R. Filipe, S. Issa, P. Romano, and J. Barreto, “Stretching the capacity of hardware transactional memory in IBM POWER architectures,” in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, pp. 107–119.
- [194] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. S. Emer, and D. Sánchez, “Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 217–230.
- [195] A. McDonald et al., “Architectural Semantics for Practical Transactional Memory,” in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, 2006, pp. 53–65.
- [196] C. B. Zilles and L. Baugh, “Extending hardware transactional memory to support non-busy waiting and non-transactional actions,” in *Proc. of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.
- [197] IBM, *Transactional execution debugging*, <https://www.ibm.com/docs/en/zos/2.1.0?topic=execution-transactional-debugging>.
- [198] S. Chaudhry et al., “Rock: A High-Performance Sparc CMT Processor,” *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.
- [199] D. Christie et al., “Evaluation of AMD’s advanced synchronization facility within a complete transactional memory stack,” in *Proceedings of the 2010 EuroSys Conference*, 2010, pp. 27–40.
- [200] Y. Afek and H. Avni, “Evaluating the Addition of Non-Transactional Loads to HTM,” *6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.
- [201] L. Hammond et al., “Transactional Memory Coherence and Consistency,” in *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004, pp. 102–113.
- [202] J. R. Crandall and F. T. Chong, “Minos: Control Data Attack Prevention Orthogonal to Memory Model,” in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004, pp. 221–232.
- [203] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: a flexible information flow architecture for software security,” in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 482–493.

- [204] E. Spertus, S. C. Goldstein, K. E. Schauer, T. von Eicken, D. E. Culler, and W. J. Dally, “Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5,” in *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, 1993, pp. 302–313.
- [205] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” in *Proceedings of the 11st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004, pp. 85–96.
- [206] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, “Hardware Enforcement of Application Security Policies Using Tagged Memory,” in *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 225–240.
- [207] J. Woodruff et al., “The CHERI capability model: Revisiting RISC in an age of risk,” in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 457–468.
- [208] E. Witchel, J. Cates, and K. Asanovic, “Mondrian memory protection,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002, pp. 304–316.
- [209] N. Vijaykumar et al., “A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory,” in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 207–220.
- [210] N. Vijaykumar et al., “MetaSys: A Practical Open-source Metadata Management System to Implement and Evaluate Cross-layer Optimizations,” *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, 26:1–26:29, 2022.
- [211] IBM, *POWER9 Processor User’s Manual (version 2.0)*, <https://ibm.ent.box.com/s/tmklq90ze7aj8f4n32er1mu3sy9u8k3k>, 2018.
- [212] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi, “End-to-end sequential consistency,” in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 524–535.
- [213] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, “Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks,” in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 93–104.

- [214] A. Ros and S. Kaxiras, “Complexity-effective multicore coherence,” in *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2012, pp. 241–252.
- [215] N. Beckmann and D. Sánchez, “Jigsaw: Scalable software-defined caches,” in *Proceedings of the 22nd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2013, pp. 213–224.
- [216] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: near-optimal block placement and replication in distributed caches,” in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009, pp. 184–195.
- [217] P.-A. Tsai, N. Beckmann, and D. Sánchez, “Jenga: Software-Defined Cache Hierarchies,” in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 652–665.
- [218] P.-A. Tsai, C. Chen, and D. Sánchez, “Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 641–654.
- [219] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory, 2nd edition* (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, 2010.
- [220] J. Kim and Y. Kim, “HBM: Memory solution for bandwidth-hungry processors,” in *Hot Chips Symposium*, 2014, pp. 1–24.
- [221] U. Pirzada, *Intel Announces The Worlds First x86 CPU With HBM Memory: Xeon Max ‘Sapphire Rapids’ Data Center CPU*, <https://wccfttech.com/intel-announces-the-worlds-first-x86-cpu-with-hbm-memory-xeon-max-sapphire-rapids-data-center-cpu/>, 2022.
- [222] Advanced Micro Devices, Inc., *Nothing Stacks Up to Epyc: Elevating Data Center Computing with AMD 3D V-Cache*, <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/product-briefs/3d-vcache.pdf>, 2021.
- [223] M. K. Qureshi and G. H. Loh, “Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 235–246.
- [224] M. Arif, K. Assogba, M. M. Rafique, and S. Vazhkudai, “Exploiting cxl-based memory for distributed deep learning,” in *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*, ACM, 2022, 19:1–19:11.

- [225] W. Huang, M. Sha, M. Lu, Y. Chen, B. He, and K. Tan, “Bandwidth expansion via CXL: A pathway to accelerating in-memory analytical processing,” in *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*, VLDB.org, 2024.
- [226] J. Sim et al., “Computational cxl-memory solution for accelerating memory-intensive applications,” in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*, IEEE, 2024, p. 615.
- [227] M. K. Aguilera, E. Amaro, N. Amit, E. Hunhoff, A. Yelam, and G. Zellweger, “Memory disaggregation: Why now and what are the challenges,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 57, no. 1, pp. 38–46, 2023.
- [228] C.-C. Chou, A. Jaleel, and M. K. Qureshi, “BATMAN: techniques for maximizing system bandwidth of memory systems with stacked-DRAM,” in *Proceedings of the 2017 International Symposium on Memory Systems (MEMSYS)*, 2017, pp. 268–280.
- [229] Y. Sun et al., “Demystifying CXL memory with genuine cxl-ready systems and devices,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*, ACM, 2023, pp. 105–121.
- [230] R. Sehgal, V. Tanna, V. Petrucci, and A. Godbole, “Optimizing system memory bandwidth with micron CXL memory expansion modules on intel xeon 6 processors,” *CoRR*, vol. abs/2412.12491, 2024. arXiv: 2412.12491.
- [231] H. A. Maruf et al., “Tpp: Transparent page placement for cxl-enabled tiered memory,” in *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.
- [232] N. Agarwal and T. F. Wenisch, “Thermostat: Application-transparent page management for two-tiered main memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds., ACM, 2017, pp. 631–644.
- [233] J. Póvoas et al., “Better memory tiering, right from the first placement,” in *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, ICPE 2025, Toronto, ON, Canada, May 5-9, 2025*, M. Litoiu, E. Smirni, A. V. Papadopoulos, and K. Wolter, Eds., ACM, 2025, pp. 253–265.
- [234] J. Liu, H. Hadian, H. Xu, and H. Li, “Tiered memory management beyond hotness,” in *19th USENIX Symposium on Operating Systems Design and Implementation*,

OSDI 2025, Boston, MA, USA, July 7-9, 2025, L. Zhou and Y. Zhou, Eds., USENIX Association, 2025, pp. 731–747.

- [235] L. Xiang et al., “Nomad: Non-exclusive memory tiering via transactional page migration,” in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, A. Gavrilovska and D. B. Terry, Eds., USENIX Association, 2024, pp. 19–35.
- [236] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, “Hemem: Scalable tiered memory management for big data applications and real NVM,” in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds., ACM, 2021, pp. 392–407.
- [237] V. Petrucci, F. Zacarias, and D. Roberts, “A limits study of memory-side tiering telemetry,” *CoRR*, vol. arXiv:2508.09351, 2025.
- [238] T. Chen et al., “Dianna: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, Utah, USA, 2014, pp. 269–284, ISBN: 9781450323055.
- [239] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
- [240] N. P. Jouppi et al., “A domain-specific supercomputer for training deep neural networks,” *Commun. ACM*, vol. 63, no. 7, pp. 67–78, Jun. 2020.
- [241] H. Kwon, A. Samajdar, and T. Krishna, “Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 461–475, ISBN: 9781450349116.
- [242] A. Parashar et al., “Timeloop: A systematic approach to dnn accelerator evaluation,” in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.
- [243] E. Qin et al., “Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.
- [244] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda, “Performance characterization of dnn training using tensorflow and pytorch on modern clusters,”

- in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–11.
- [245] Y. Ren, S. Yoo, and A. Hoisie, “Performance analysis of deep learning workloads on leading-edge systems,” in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 103–113.
- [246] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, Jul. 2019, pp. 947–960, ISBN: 978-1-939133-03-8.
- [247] M. Isaev, N. McDonald, L. Dennison, and R. W. Vuduc, “Calculon: A methodology and tool for high-level co-design of systems and large language models,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, D. Arnold, R. M. Badia, and K. M. Mohror, Eds., ACM, 2023, 71:1–71:14.
- [248] J. Bang, Y. Choi, M. Kim, Y. Kim, and M. Rhu, “Vtrain: A simulation framework for evaluating cost-effective and compute-optimal large language model training,” in *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*, IEEE, 2024, pp. 153–167.
- [249] Y. Li et al., “Triosim: A lightweight simulator for large-scale DNN workloads on multi-gpu systems,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*, ACM, 2025, pp. 1524–1538.
- [250] Y. Tang et al., “Simulating LLM training in cxl-based heterogeneous computing cluster,” in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops, Vancouver, BC, Canada, May 20, 2024*, IEEE, 2024, pp. 1–6.
- [251] J. Dong et al., “Eflops: Algorithm and system co-design for a high performance distributed training platform,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 610–622.
- [252] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, “A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Nov. 2020, pp. 463–479, ISBN: 978-1-939133-19-9.
- [253] A. Shah et al., “Synthesizing collective communication algorithms for heterogeneous networks with TACCL,” *CoRR*, vol. abs/2111.04867, 2021. arXiv: 2111.04867.

- [254] P. Sun, Y. Wen, R. Han, W. Feng, and S. Yan, “Gradientflow: Optimizing network performance for large-scale distributed dnn training,” *IEEE Transactions on Big Data*, vol. 8, no. 2, pp. 495–507, 2022.
- [255] P. Jain et al., “Checkmate: Breaking the memory wall with optimal tensor rematerialization,” in *Proceedings of Machine Learning and Systems*, vol. 2, 2020, pp. 497–511.
- [256] L. Zheng et al., “Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Jul. 2022, pp. 559–578, ISBN: 978-1-939133-28-1.
- [257] J. Chen, S. Li, R. Guo, J. Yuan, and T. Hoefler, “Autoddl: Automatic distributed deep learning with asymptotically optimal communication,” *CoRR*, vol. abs/2301.06813, 2023. arXiv: 2301.06813.
- [258] S. Zhang et al., “Auto-parallelizing large models with rhino: A systematic approach on production AI platform,” *CoRR*, vol. abs/2302.08141, 2023. arXiv: 2302.08141.
- [259] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” in *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, 2019.
- [260] IEA, Energy and AI, “*Energy demand from AI*”, <https://www.iea.org/reports/energy-and-ai>, Published: 2025. Last Accessed: Oct 13, 2025.
- [261] A. Kokolis et al., “Revisiting reliability in large-scale machine learning research clusters,” in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025*, IEEE, 2025, pp. 1259–1274.
- [262] S. Rashidi et al., “Enabling compute-communication overlap in distributed deep learning training platforms,” in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*, IEEE, 2021, pp. 540–553.
- [263] Y. Li, I. Liu, Y. Yuan, D. Chen, A. G. Schwing, and J. Huang, “Accelerating distributed reinforcement learning with in-switch computing,” in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, S. B. Manne, H. C. Hunter, and E. R. Altman, Eds., ACM, 2019, pp. 279–291.
- [264] A. Sapio et al., “Scaling distributed machine learning with in-network aggregation,” in *18th USENIX Symposium on Networked Systems Design and Implemen-*

- tation, *NSDI 2021, April 12-14, 2021*, J. Mickens and R. Teixeira, Eds., USENIX Association, 2021, pp. 785–808.
- [265] M. Ewais and P. Chow, “Disaggregated memory in the datacenter: A survey,” *IEEE Access*, vol. 11, pp. 20 688–20 712, 2023.
- [266] H. A. Maruf and M. Chowdhury, “Memory disaggregation: Advances and open challenges,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 57, no. 1, pp. 29–37, 2023.
- [267] R. Wang, J. Wang, S. Idreos, M. T. Özsu, and W. G. Aref, “The case for distributed shared-memory databases with rdma-enabled memory disaggregation,” *Proc. VLDB Endow.*, vol. 16, no. 1, pp. 15–22, 2022.
- [268] T. Heo et al., “Exploring memory expansion designs for training mixture-of-experts models,” in *Workshop on Hot Topics in System Infrastructure (HotInfra) 2023, Orlando, Florida, USA, June 18, 2023*.
- [269] D. Xu, Y. Feng, K. Shin, D. Kim, H. Jeon, and D. Li, “Efficient tensor offloading for large deep-learning model training based on compute express link,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2024, Atlanta, GA, USA, November 17-22, 2024*, IEEE, 2024, p. 94.
- [270] M. M. K. Martin, M. D. Hill, and D. J. Sorin, “Why on-chip cache coherence is here to stay,” *Commun. ACM*, vol. 55, no. 7, pp. 78–89, 2012.
- [271] N. Agarwal, D. W. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, “Selective GPU caches to eliminate CPU-GPU HW cache coherence,” in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, IEEE Computer Society, 2016, pp. 494–506.
- [272] S. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, and W. Hwu, “Analysis and optimization of I/O cache coherency strategies for soc-fpga device,” in *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019*, I. Sourdis, C. Bouganis, C. Álvarez, L. A. T. Díaz, P. Valero-Lara, and X. Martorell, Eds., IEEE, 2019, pp. 301–306.
- [273] Y. Yu, S.-s. Lee, A. Khandelwal, and L. Zhong, *Gcs: Generalized cache coherence for efficient synchronization*, 2023. arXiv: 2301.02576 [cs.DC].
- [274] A. Shilov, “Nvidia’s next-gen Blackwell AI Superchips could cost up to \$70,000 — fully-equipped server racks reportedly range up to \$3,000,000 or more,” *Accessed: October 2025*, May 14, 2024.

- [275] P. Duraisamy et al., “Towards an adaptable systems architecture for memory tiering at warehouse-scale,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., ACM, 2023, pp. 727–741.
- [276] J. Kim, W. Choe, and J. Ahn, “Exploring the design space of page management for multi-tiered memory systems,” in *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, I. Calciu and G. Kuenning, Eds., USENIX Association, 2021, pp. 715–728.
- [277] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, “MEMTIS: efficient memory tiering with dynamic page classification and page size determination,” in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, 2023, pp. 17–34.
- [278] S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang, “Vtmm: Tiered memory management for virtual machines,” in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds., ACM, 2023, pp. 283–297.
- [279] Y. Sun et al., “M5: mastering page migration and memory management for cxl-based tiered memory systems,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds., ACM, 2025, pp. 604–621.
- [280] A. Maruf, A. Ghosh, J. Bhimani, D. Campello, A. Rudoff, and R. Rangaswami, “MULTI-CLOCK: dynamic tiering for hybrid memory systems,” in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*, IEEE, 2022, pp. 925–937.
- [281] H. Lee et al., “Beyond page migration: Enhancing tiered memory performance via integrated last-level cache management and page migration,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’25, Association for Computing Machinery, 2025, pp. 1763–1776, ISBN: 9798400715730.
- [282] K. Zhao et al., “Contiguitas: The pursuit of physical memory contiguity in datacenters,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Y. Solihin and M. A. Heinrich, Eds., ACM, 2023, 44:1–44:15.

- [283] L. T. Clark, S. B. Medapuram, D. K. Kadiyala, and J. Brunhaver, “Physically unclonable functions using foundry sram cells,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 3, pp. 955–966, 2019.
- [284] L. T. Clark, S. B. Medapuram, and D. K. Kadiyala, “Sram circuits for true random number generation using intrinsic bit instability,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 2027–2037, 2018.

VITA

Divya Kiran Kadiyala was born in Nellore, Andhra Pradesh, India, in August 1991. He completed his schooling in Nellore under the guidance of his grandfather, Sri K. Penchalayya garu, an academic, scholar, and visionary whose emphasis on intellectual curiosity, discipline, and values-based spiritual education strongly shaped his academic and professional trajectory.

He earned his Bachelor's degree in Electronics and Communication Engineering from KL University in 2013 and subsequently worked as an Assistant Systems Engineer at Tata Consultancy Services in Chennai, India, from 2013 to 2015. In 2015, he moved to the United States to pursue a Master's degree in Electrical Engineering at Arizona State University, Tempe, where he conducted research on SRAM-based physically unclonable functions [283] and hardware-based true random number generators [284] under the supervision of Dr. Lawrence Clark. From 2017 to 2019, he worked as a Senior Applications Engineer at Cadence Design Systems in San Jose, California.

In August 2019, he began his doctoral studies in Electrical and Computer Engineering at the Georgia Institute of Technology and completed his Ph.D. in 2025 under the advisement of Dr. Alexandros Daglis. His dissertation, *Memory System Optimizations for Parallel and Bandwidth-Intensive Workloads*, presents innovative approaches to enhance the performance of parallel and bandwidth-intensive workloads in contemporary datacenter and cloud environments. The research pioneers methods for leveraging memory system optimizations tailored to the workload behaviors and underlying hardware to improve the capacity, bandwidth, and latency characteristics of memory systems at the chip, server, and cluster levels. His research work has been published in premier computer architecture venues.

Alongside his academic work, Divya Kiran has cultivated strong industry experience

through roles and internships at Luminous Computing, Samsung, and Hewlett Packard Enterprise, where he worked on novel memory system architectures and composable systems for distributed AI training. He also served as a graduate student senator in the Georgia Tech Graduate Student Government Association, representing graduate student interests and contributing to initiatives that improved the graduate student experience.

Looking ahead, he aspires to pursue a research-oriented career, ideally as a tenure-track faculty member, advancing novel computing systems and device architectures to meet the growing demands of artificial intelligence and machine learning. He is equally passionate about teaching and creating accessible, high-quality educational materials. In his free time, he enjoys reading literature and is an ardent fan of light music.