Design, Modeling and Optimization of Large-Scale Disaggregated Memory Systems

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

By

AMIT PURI

Under the Supervision of

Dr. JOHN JOSE

&

Prof. TAMARAPALLI VENKATESH

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

August, 2024

Declaration

This is to certify that the thesis entitled "**Design, Modeling and Optimization of Large-Scale Disaggregated Memory Systems**", submitted by me to the *Indian Institute of Technology Guwahati*, for the award of the degree of Doctor of Philosophy, is a bonafide work carried out by me under the supervision of Dr. John Jose and Prof. Tamarapalli Venkatesh. The content of this thesis, in full or in parts, have not been submitted to any other University or Institute for the award of any degree or diploma. I also wish to state that to the best of my knowledge and understanding nothing in this report amounts to plagiarism.

Signed: _____

Amit Puri
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati-781039, Assam, India.

Date: _____

Certificate

This is to certify that the thesis entitled "**Design, Modeling and Optimization of Large-Scale Disaggregated Memory Systems**", submitted by Amit Puri (186101102), a Ph.D student in the *Department of Computer Science and Engineering*, *Indian Institute of Technology Guwahati*, for the award of the degree of Doctor of Philosophy, is a record of an original research work carried out by him under my supervision and guidance. The thesis has fulfilled all requirements as per the regulations of the institute and, in my opinion, has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Signed: _____

Supervisor: Dr. John Jose
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati-781039, Assam, India.

Signed: _____

Supervisor: Prof. Tamarapalli Venkatesh
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati-781039, Assam, India.

Date: _____

Acknowledgements

It has been a long journey at IIT Guwahati during my PhD. This Thesis would not have been completed without the support of many well-wishers. The following is a small attempt to acknowledge their support. First, I would like to express my deepest gratitude to my Ph.D. supervisors, Dr. John Jose and Prof. Tamarapalli Venkatesh, for their valuable guidance and advice and for giving me the opportunity to work with them. Their constant support and guidance paved the way for my development as a researcher and a professional with genuine skills. I am continually amazed at their clarity of thought and ability to get a fundamental idea behind a given problem. I benefited immensely from their unique advising style, which combines the correct balance between allowing students to perform independent research versus guiding them to ensure they pick the right topics of interest. It has been an honor to work with them.

Besides my supervisors, I would like to thank other members of my doctoral committee, Prof. Jatindra Kumar Deka, Dr. Moumita Patra, and Dr. Salil Kashyap, for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives. I want to express my heartful gratitude to the director, the deans, and other management of IIT Guwahati, whose collective efforts have made this institute a place for world-class studies and education. I am also thankful to all faculty and staff of the Department of Computer Science and Engineering for extending their cooperation in terms of technical and official support to complete my research work successfully. I am incredibly thankful to Prof. Vijaykrishnan Narayanan from Pennsylvania State University, USA, for his guidance and valuable and timely input on many of my thesis-related contributions. His expertise in the topic has been invaluable to me. I am thankful to the anonymous reviewers of my papers for their critical reviews and for providing genuine suggestions and feedback.

No journey is complete without a group of friends, and thanking them is just not enough. They are the support structure away from home. I am grateful to Abir Banerjee, Suvarthi, Soumyadeep, Rohit, Neelmadhav, and Ajanta, who stood by me through my difficult times. I would also like to wholeheartedly thank my lab mates from Multi-Core ARchitecture and Systems (MARS) lab, with a special mention to Manju R, Dipika Deb, Siva Kumar, Abhijit Das, Syam Sankar, Rajeshwari, Vivekananda, Ilina Sinha, and Debarshi. Last, I thank my batch mates from the CSE and ECE departments, Sujit Kumar, Karnish, Vanshali Sharma, and Yogesh, who were always available for me. I am very thankful to the family of Dr. John Jose, who always treated the lab members like a family friend and made us feel at home.

Most importantly, none of this would have been possible without the love and patience of my family. I am grateful to my parents, especially my mother, for their unconditional love, patience, and the sacrifices they have made for me. I will forever stay indebted to them. I am also very grateful to my sister and brother-in-law for their support and for taking great care of my parents in my absence, making it possible for me to focus on work. Finally, my sweet nephew Arnav, who is the center of attraction of our family, always cheered me up, even during my hard days. I dedicate this thesis to all of my family members.

Sincerely
Amit Puri

Abstract

Modern server workloads continue to increase in their main memory requirements. However, the traditional server system fails to scale up with the memory requirements of these workloads due to the memory capacity wall. The memory allocation in server systems is based on peak requirements, which has led to the under-utilization of onboard memory resources. Further, the hardware refresh cycles in data centers are compromised due to rigid and coupled hardware resources in traditional server systems. Disaggregated Memory Systems (DMS) have emerged as a strong alternative to monolithic servers, where memory can be attached as remote memory nodes/pools and is connected to compute nodes over high-speed memory-centric coherent interconnect. Memory allocation in DMS is more flexible as it allows on-demand memory allocation from the shared memory pools. It also eliminates scalability and under-utilization issues while allowing decoupled up-gradation of server memory resources, reducing the total cost of ownership. However, DMS introduces new design and architectural challenges for properly utilizing the multi-tiered memory system. Memory disaggregation also increases the Average Memory Access Time (AMAT) due to a network interconnect between compute nodes and remote memory pools. These delays can significantly impact the application performance, which may further degrade in expected large-scale configurations of DMS. Lastly, there is no architectural simulator for the performance evaluation of DMS.

This thesis addresses the new design challenges and focuses on reducing AMAT in DMS by implementing various system-level or architectural optimizations. The DMS is expected to be deployed inside data center racks in large-scale configurations. This led to the first contribution of this thesis towards building an architectural simulator for the performance evaluation of a scalable DMS. The proposed simulator models a fully configurable DMS with all the required components. Next, we explore the possibility of a hot-page migration system on a large-scale DMS, widely used in multi-tiered memory systems, to bring frequently accessed pages into fast memory. We studied Epoch-based and on-the-fly page migration techniques and proposed hardware mechanisms to accelerate the data movement between slow and fast memory, effectively reducing the AMAT and improving application performance. In large-scale configurations, multiple compute nodes access remote memory pools simultaneously, introducing the possibility of memory bandwidth contention. Due to uninformed memory allocation, the memory access traffic can become skewed towards a particular memory pool at every interval. This motivates the need for smart remote memory allocation techniques such that memory traffic is load-balanced among different pools. Finally, we study the implications of network resource allocation in providing Quality of Service (QoS) to different applications running at compute nodes. We first measure the impact of scheduling remote memory requests of different compute nodes on the application performance. We perform the experiment analysis with various data-centric and HPC workloads on our proposed simulator. The results show that the proposed mechanisms can significantly improve the application performance for large-scale DMS.

Contents

Contents iii

List of Figures vii

List of Tables xi

List of Abbreviations xiii

1 Introduction 1

1.1 Traditional Server System Design 1

1.2 Disaggregated Memory Systems 4

1.2.1 Performance Improvement with Disaggregated Memory System . . 7

1.3 Challenges in Disaggregated Memory System 8

1.3.1 System Design 8

1.3.2 High Average Memory Access Time 9

1.3.3 Remote Memory Management and Control 9

1.3.4 Architectural Simulation 10

1.4 Thesis Contribution 10

1.4.1 DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems 10

1.4.2 A Practical approach for workload-aware data movement in Disaggregated Memory Systems 11

1.4.3 CosMoS: Architectural Support for Cost-Effective Data Movement in a Scalable Disaggregated Memory Systems 11

1.4.4 Design and Analysis of Memory Allocation Policies for Disaggregated Memory System 12

1.4.5 QoS Management in Large-Scale Disaggregated Memory Systems . 12

1.5 Thesis Organization 13

2 Background and Literature Survey 15

2.1 Existing Solutions for Memory Scalability 16

2.1.1 Scaling Memory Locally . . . 16
2.1.2 Scaling Memory Remotely . . . 16
2.2 Interconnects . . . 18
2.3 Disaggregated Memory Systems . . . 19
2.4 Reducing Memory Delays with Page Migration . . . 23
2.4.1 Page Migration Overheads and Parameters . . . 26
2.4.2 Different mechanisms for Hot Page Migration . . . 27
2.5 Fairness and QoS in Large-Scale Systems . . . 28
2.6 Performance Evaluation of Large-Scale DMS . . . 28
2.6.1 Simulation Tools and Techniques . . . 28
2.6.2 Binary Instrumentation with Intel PIN . . . 29
2.7 Summary . . . 30

3 DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems 33
3.1 Introduction . . . 33
3.2 Motivation . . . 35
3.3 Baseline Hardware Disaggregated Memory Systems . . . 36
3.3.1 Remote Memory Organization . . . 37
3.4 DRackSim Design and Operations . . . 38
3.4.1 Trace-Based Model . . . 39
3.4.2 Cycle-Level Simulation Model . . . 40
3.4.3 Back-end Modeling . . . 42
3.5 Validation . . . 46
3.6 Evaluation . . . 49
3.6.1 Design Space Exploration . . . 50
3.6.2 Multi-Node Disaggregated Memory Systems . . . 55
3.6.3 Sensitivity to Local Memory Footprint . . . 57
3.6.4 Network Latency and Bandwidth Test . . . 58
3.6.5 Simulator Performance . . . 59
3.7 Summary . . . 59

4 A Practical Approach For Workload-Aware Data Movement in Disaggregated Memory Systems 61
4.1 Introduction . . . 62
4.2 System Design . . . 64
4.2.1 Hot Page Tracking . . . 65
4.2.2 Performing Migration and Using Page Buffers . . . 66
4.2.3 Access Controller . . . 66

4.2.4 Pending Block Accesses . . . 67
4.3 Remote Memory Access Data Path . . . 68
4.4 Hardware Overheads . . . 69
4.5 Characterizing Workloads with Training . . . 69
4.6 Experimental Analysis . . . 71
4.6.1 Results . . . 71
4.6.2 Sensitivity Analysis . . . 74
4.7 Summary . . . 76

5 CosMoS: Architectural Support for Cost-Effective Data Movement in a Scalable Disaggregated Memory Systems 79
5.1 Introduction . . . 80
5.2 Background and Motivation . . . 81
5.2.1 Workload Characterization . . . 82
5.2.2 Analysis and Limitations . . . 84
5.3 *CosMoS* Architecture . . . 85
5.3.1 Design Modules . . . 86
5.3.2 CosMoS Complete Design . . . 89
5.4 Experiment Analysis . . . 91
5.4.1 Results . . . 92
5.5 Summary . . . 98

6 Design and Analysis of Memory Allocation Policies for Disaggregated Memory System 101
6.1 Introduction . . . 101
6.2 Memory Allocation Policies . . . 102
6.2.1 Conventional Allocation Policy . . . 103
6.2.2 Smart-idle Selection . . . 104
6.2.3 Uniform Load Partitioning . . . 106
6.3 Experimentation Methodology and Results . . . 107
6.3.1 Impact on Memory Latency . . . 108
6.3.2 Impact on Tail Latency . . . 110
6.3.3 Overall Latency Breakdown . . . 111
6.3.4 Performance Slowdown . . . 111
6.3.5 Impact on HPC workloads . . . 112
6.3.6 Complexity Analysis and Performance Impact . . . 113
6.4 Summary . . . 114

7 Understanding the Performance Impact of Queue-Based Resource Allocation in Scalable Disaggregated Memory Systems **115**
7.1 Introduction . . . 116
7.2 Background and Motivation . . . 116
7.3 Design . . . 118
7.3.1 Weighted Round-Robin Scheduling . . . 118
7.3.2 Round-Robin Scheduling . . . 119
7.3.3 Priority Scheduling . . . 119
7.3.4 Priority-based Weighted Scheduling . . . 120
7.4 Methodology and Results . . . 121
7.5 Summary . . . 125

8 Conclusion and Future Work **127**
8.1 Summary . . . 127
8.2 Future Work . . . 129

Bibliography **131**

List of Publications **143**

List of Figures

1.1 Comparison of Parameter Count in AI models and Memory Capacity Growth in Accelerators over the years [1] 2
1.2 Recent trends in performance improvement of Compute (CPU/GPU/Accelerators) and Memory over time [2] 3
1.3 Stranded memory resources due to fragmentation in traditional servers . . 3
1.4 (a) Traditional Server Design (b) Disaggregated Memory System Design . 5
1.5 Improved resource utilization with Disaggregation 5
1.6 Performance Difference Local vs RDMA vs DMS 8

2.1 Binary Instrumentation through PIN 30

3.1 Overview of DMS and its Interface with Host Compute Nodes 36
3.2 Remote Memory Exposure to Compute Nodes (a) Shared Organization (b) Distributed Organization 38
3.3 Address Translation at Remote Memory Controller 38
3.4 *DRackSim* infrastructure overview; "LM" Local memory, "MMU" Memory management unit, "IF" Interface 39
3.5 Trace Generation (a) Recording Main-memory access (b) Final Multi-Threaded Trace 39
3.6 Out-of-Order Core Modeling Subsystem 40
3.7 (a) Interconnect Simulation Model detail (b) Packet Structure for Remote Memory Access 44
3.8 Validation on *Splash-3* benchmarks (a) Normalized IPC (b) Normalized LLC Misses (c) *DRackSim* vs DirectCXL 47
3.9 Last Level Cache Misses normalized against Gem5 over different cache configurations 48
3.10 Impact on system performance (top) and memory cost (bottom) on all the workloads over all four configurations 52
3.11 Impact on system performance for all workloads on changing the network latency and bandwidth parameters 54
3.12 Impact on system performance for all the workloads on increasing the number of memory pools with a single compute pool (Top) or by increasing the number of compute nodes with a single memory pool (Bottom) . . 56
3.13 Impact on system performance with different compute-to-memory node configurations and workload combinations over 4 Compute Nodes (Top) and 16 Compute Nodes (Bottom) 56
3.14 Impact on system performance on changing the local memory footprint . 57
3.15 IPC (Top) and Average memory access delay (Bottom) for STREAM benchmark on changing the network bandwidth and latency 58

3.16 Simulator Performance Compared to Gem5 . . . 58
3.17 Increase in simulation time *DRackSim* for large-scale simulations for one million instructions per node (Normalized against the performance of a single node) . . . 58

4.1 Baseline Hardware Disaggregated Memory System . . . 63
4.2 Centralized Page Migration Support with Workload-Aware Efficient Data Movement, 'R' represents a Request Selector . . . 65
4.3 (a) Hot Page Tracking Table structure (b) Page Buffer structure . . . 65
4.4 Access Controller to control Multi-granularity Access . . . 66
4.5 Flowchart representing the data path for remote memory access . . . 68
4.6 Performance Slowdown for all the workloads with different data movement policies . . . 72
4.7 Increase in memory access cost for all the workloads with different data movement policies . . . 73
4.8 Percentage of memory access at local memory due to migration of pages . 74
4.9 Performance slowdown on changing the local memory footprint . . . 74
4.10 Performance slowdown on using multiple memory pools over different network configurations . . . 75
4.11 Performance Slowdown with Multiple Compute and Memory Nodes over different network configurations . . . 76

5.1 Memory Controller at Compute and Memory Node . . . 81
5.2 %age of pages with different access frequency for different workloads . . . 82
5.3 Rolling Average Access Frequency plot for majority of Pages in different Workloads (same colors as depicted in Fig. 5.2), X-axis represents n^{th} access to a Page, Y-Axis represents the number of CPU Cycles . . . 82
5.4 Access order within the pages in different workloads (using page offset), X-axis represents n^{th} access to a Page, Y-Axis represents the percentage of the occurrence of the majority offset . . . 83
5.5 Design Components in *CosMoS* . . . 86
5.6 (a) Complete Design of Memory Controllers in CosMoS, 'P': Page Request, 'B': Block Request | (b),(c) Possible Placement of Scheduling Page Queues (PQ) and Bandwidth Partition Controller (BP) – (b) at the Memory Node (Mnode, MN) (c) at the Compute Node (Cnode, CN) . . . 88
5.7 Flowchart representing the data path for remote memory access . . . 89
5.8 Performance Speed-Up with *CosMos* vs *Daemon* vs *TPP*, Normalized to Baseline . . . 93
5.9 Improvement in Memory Latency for *CosMos* vs *Daemon* vs *TPP*, Normalized to Baseline . . . 94
5.10 Local Hit-Ratio with *CosMos* vs *Daemon* vs *TPP*, Normalized to Baseline 94
5.11 *CosMoS* vs *Daemon* vs *TPP* in Large-Scale Configurations . . . 95
5.12 Performance Impact of Page Scheduling and Bandwidth Partitioning . . . 97
5.13 Impact of changing Bandwidth Parameters . . . 97
5.14 Impact of changing Latency Parameters . . . 97
5.15 Impact of Changing Remote Memory Footprint . . . 98

6.1 Contention at remote memory node-1 due to an imbalance in the number of memory accesses across memory nodes . . . 103

6.2 Random selection with alternate local-remote page allocation (a) Average memory access latency (b) Remote memory access latency (c) memory access latency distribution (d) Access variation in memory nodes 104
6.3 Average memory access latency with *Local-First* allocation over workload *WL-Mix1* (a) Round-Robin selection (b) Smart-Idle selection (c) Uniform load partition (d) Average remote memory latency 108
6.4 Average memory access latency with *Alternate Local-Remote* allocation over workload WL-Mix1 (a) Round-Robin selection (b) Smart-Idle pool selection (c) Uniform load partition (d) Average remote memory latency . 110
6.5 Distribution of remote memory accesses based on access latency 110
6.6 Local/Remote/Network Latency breakdown 111
6.7 Execution times normalized against entirely local memory (as 1) vs. DMS with two different scenarios (256MB and 512MB of local memory at each compute node) 112
6.8 Average memory latency for HPC workloads and mini-apps with different proportions of local and remote memory using alternate local-remote and uniform-load partition 112
6.9 Normalized execution time for HPC workloads and mini-apps with different proportions of local and remote memory using alternate local-remote and uniform-load partition 113

7.1 Overview of Global Memory Controller 117
7.2 Increase in Memory latency on changing the nodes sharing same memory node 117
7.3 Drop in IPC on changing the nodes sharing the same memory node 117
7.4 Multiple queues at Global Memory Controller 118
7.5 Normalized memory latency for Mix-4A with 60% of workloads footprint at remote to a system with entirely local memory (a) 1-shared memory node (b) 2-shared memory nodes 122
7.6 Normalized IPC on workload Mix-4A with 60% of workloads footprint at remote to a system with entirely local memory (a) 1-shared memory node (b) 2-shared memory nodes 123
7.7 Workload Mix-4B using 2-shared memory node with 60% of workloads footprint at remote to a system with entirely local memory (a) Normalized memory latency (b) Normalized IPC 124
7.8 Workload Mix-8 using 4-shared memory node with 50% of workloads footprint at remote to a system with entirely local memory (a) Normalized memory latency (b) Normalized IPC 125

8.1 Thesis Summary 128

List of Tables

3.1 Validation Parameters 47
3.2 Benchmarks 49
3.3 Simulation Parameters 50

4.1 Simulation Parameters 71
4.2 Benchmarks 71

5.1 Benchmarks 92
5.2 Simulation Parameters 92

6.1 Benchmarks 107
6.2 Simulation Parameters 107

7.1 Simulation Parameters 121
7.2 Benchmarks 121
7.3 Workload Mixes 121

List of Abbreviations

TCO	Total Cost of Ownership
HPC	High-Performance Computing
DSM	Distributed Shared Memory
RDMA	Remote Direct Memory Access
DMS	Disaggregated Memory System
LLC	Last-Level cache
AMAT	Average Memory Access Time
QoS	Quality-of-Service
HBM	High-Bandwidth Memory
SCI	Scalable Coherent Interconnect
NTB	Non-Transparent Bridge
RoCE	RDMA over Convereged Ethernet
NIC	Network Interface Card
OCI	On-Chip Interconnect
VMM	Virtual Memory Manager
DMC	Disaggregated Memory Controller
PEBS	Precise Event-Based Sampling
MMU	Memory Management Unit
ARF	Architecture Register File
RAT	Register Alias Table
ROB	Re-Order Buffer
AGU	Address Generation Unit
MSHR	Miss-Status Handling Register
PTE	Page-Table Entry
TLB	Translation Look-aside Buffer
IPI	Inter-Processor Interuppt
MPKI	Miss Per Kilo Instructions
FIFO	First-In-First-Out
PEBS	Precise Event-Based Sampling

Chapter 1
Introduction

In this era of computing, a large amount of data is generated endlessly from billions of Internet users through social media, e-commerce websites, mobile devices, and so on. This data is processed at large computing facilities such as data centers to produce meaningful information. These data centers have sizes bigger than a football stadium and host lots amount of computing infrastructure such as server systems where the number of servers can go as high as 1 million at a single facility. The computer servers are managed inside racks housing multiple servers depending on their size. Over the past few years, the demand for such computing infrastructure has continuously increased, and more such hyper-scale facilities are being constructed worldwide. Considering the importance of data centers that involve substantial economic costs, it is observed that there lies a good scope for improvement and opportunity to perform research in this area. The architecture of futuristic data centers will decide whether or not the rising demand for hardware infrastructure will be met smoothly. Plenty of architectural and system-level design concerns exist in the current generation of data center hardware. So, these hardware designs require an upgrade with more efficient designs to reduce the total cost of ownership (TCO).

1.1 Traditional Server System Design

The traditional servers in the data centers have a fixed set of memory units that cannot be upgraded once manufactured, as the CPU socket count limits memory expansion. On the other hand, workloads arising out of Machine Learning, Artificial Intelligence, Big Data, Graph/Video/Text Analytics, and scientific computing applications have large requirements for both computing power and memory. The requirements are ever-increasing with every new application in this category. Although computing power is growing rapidly in modern systems, memory is not growing at the same rate. The trend can be

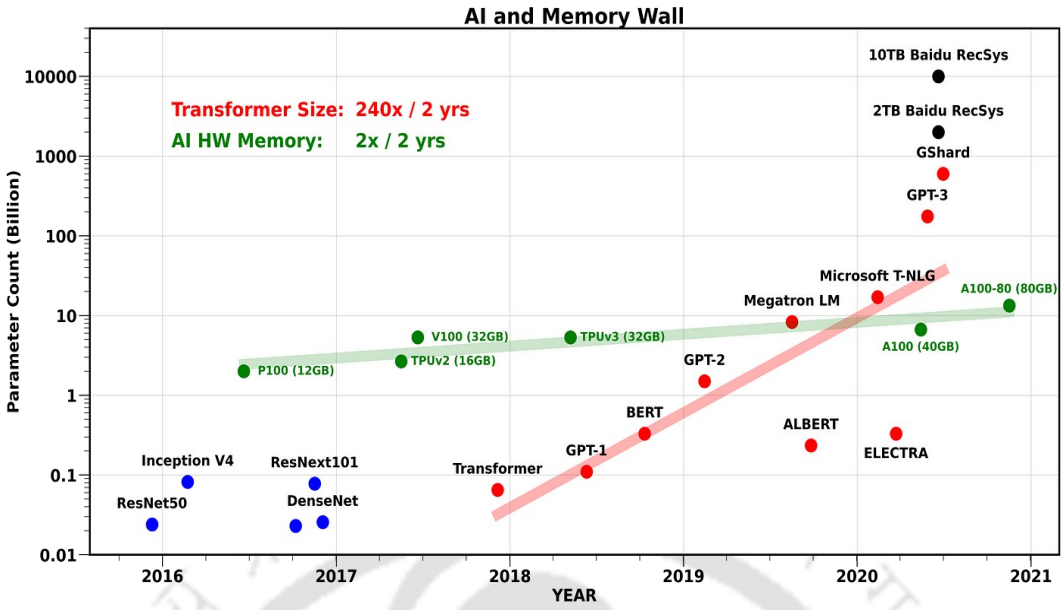


FIGURE 1.1: Comparison of Parameter Count in AI models and Memory Capacity Growth in Accelerators over the years [1]

understood from the graph in Fig. 1.1. The graph explains the emerging challenge of training and serving state-of-the-art neural network models. The LLM model sizes have increased by 410x every two years in their parameter count. However, the accelerator DRAM memory has only scaled at a rate of 2x per two years. The memory requirements of these large models are several times larger than the number of parameters during the training. These challenges are commonly referred to as the memory wall problem. A similar trend can be observed from the graph in Fig. 1.2. While the CPU performance increases at the rate of 50% per year, it is only 7% for the memory. This results in an increasing gap between CPU and memory performance, which is growing at 50% per year. The gap is even more when compared to GPUs and Deep Learning accelerators. As a consequence of the memory wall, there is a limit to both memory capacity and bandwidth that significantly impacts the application performance. The imbalance between the memory capacity/bandwidth and workload demands has introduced multiple problems in traditional server systems. We summarize these problems below:

- **Memory Under-utilization:** The data-centric workloads have unpredictable memory demands [3] and require over-provisioning during memory allocation to meet the peak requirements. This results in under-utilization of onboard memory resources as the memory remains stranded with the applications even if it is not used. A study on Google Cloud [4] reported that an average of 30% memory was idle during 70% of the running time, and only 50% on average was used out of the 80% memory allocated. Similarly, when workloads are allocated to different server nodes, fragments of memory get stranded in the nodes and become unusable. The problem of memory under-utilization can be observed with a hypothetical example

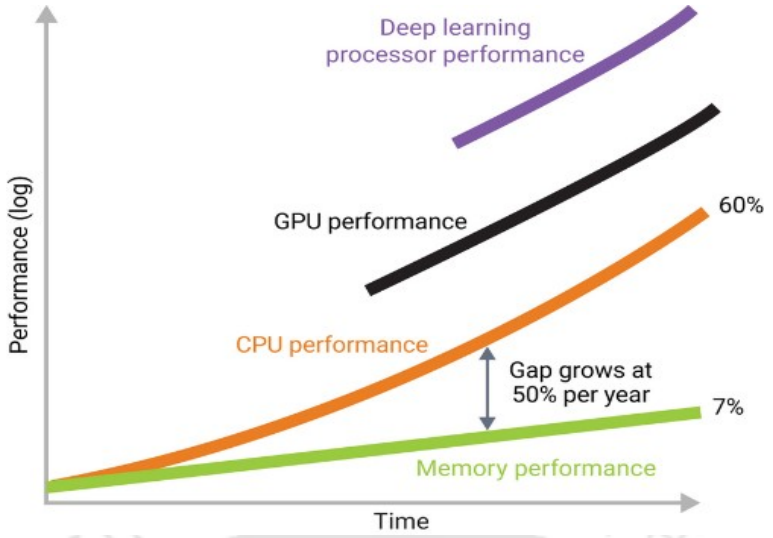


FIGURE 1.2: Recent trends in performance improvement of Compute (CPU/GPU/Accelerators) and Memory over time [2]

shown in Fig. 1.3, where multiple requests with specific CPU/memory demands are to be allocated resources from the available server nodes. Three server nodes are assumed to be available, each with a 4-memory and 3-compute unit capacity. However, due to a lack of flexibility in resource allocation, only three requests could be allocated out of four. Even though enough memory was available in fragments, W4 could not be allocated due to constraints in the underlying server architecture.

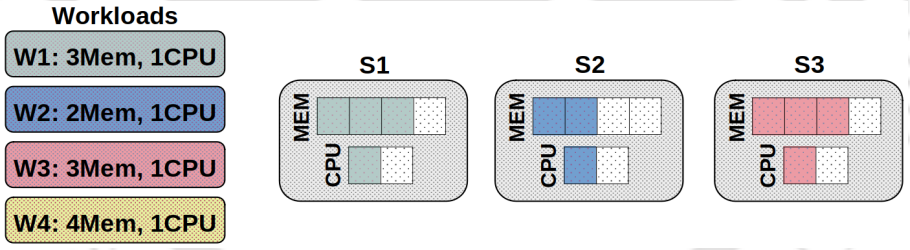


FIGURE 1.3: Stranded memory resources due to fragmentation in traditional servers

- **Memory Scalability:** Traditional server systems fail to scale up with the increasing memory requirements of modern workloads that can expand to terabytes. This causes a significant performance impact as application data must be frequently retrieved from the disk to the main memory which is very slow. Some efforts have been made in the past to improve memory scalability, such as software/hardware Distributed Shared Memory (DSM) [5] and Remote Direct Memory Access (RDMA) [6, 7] that allows memory sharing among server nodes. However, these systems suffer from significant data movement overhead and have lower performance due to high tail latency. We discusses more about these systems in section 2.1.

- **Energy Loss and Increased TCO:** The unpredictable memory usage in large data center facilities requires extra investment in purchasing more server nodes to handle the same level of workloads. This increases energy consumption and requires more cooling equipment to remove excess heat production, significantly increasing the TCO.

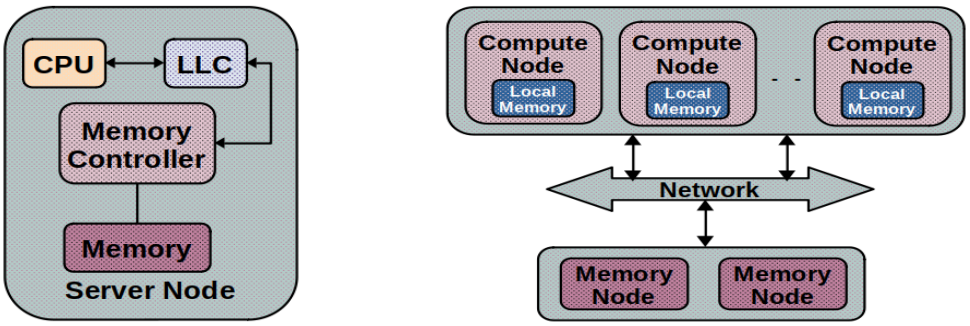
- **Impact on Data Center Refresh Cycle:** The fixed box architecture of traditional servers also constrains the data center hardware up-gradation cycle. If there is a requirement to upgrade the main memory in the server nodes due to the availability of new technology in the market, it requires replacing of complete server unit. This is because the hardware resources inside a server node are tightly coupled, and the exclusive replacement of the main memory is not feasible. This results in the shortening of data center refresh cycles.

1.2 Disaggregated Memory Systems

A fundamental change in the data center's server architecture is long-awaited due to the above-discussed problems. Hardware Disaggregated Memory Systems (HDMS or simply DMS) [8–12] have emerged as a strong alternative to traditional server system architecture. Unlike monolithic servers completely relying on local onboard memory resources or using DSM/RDMA with memory sharing among server nodes, DMS introduces a memory pooling system. A large amount of remote memory (DRAM) can be connected to the compute nodes (or server nodes) as independent memory pools via a high-speed interconnect. Fig. 1.4 shows the high-level comparison of system design in traditional servers and DMS. The compute nodes in DMS have a small local memory and rely on remote memory for most of their application requirements. DMS supports on-demand allocation of remote memory to the compute nodes, which is added as an extension to the local memory address space in a flat address structure. The DMS is expected to be configured in large-scale configurations, as shown in Fig. 1.4b, where multiple compute nodes share the remote memory pools/nodes [1] for memory allocation. Remote memory management is performed through an in-network memory manager and is separate from the memory management at the compute node.

The memory-centric fabric supports cache coherent access to the remote memory and allows memory access at cache block granularity on a last-level cache (LLC) miss. Such interconnects have been proposed earlier for low latency and high bandwidth access to remote memory [13–15]. CXL 3.0 [16, 17] is the latest industry standard for similar remote memory binding fabric supporting coherent access to pooled memory systems using CXL.mem protocol. It also includes a CXL switch supporting multiple hosts

[1] Memory nodes and memory pools are used interchangeably in this thesis



(a) (b)

FIGURE 1.4: (a) Traditional Server Design (b) Disaggregated Memory System Design

and CXL devices through the CXL (PCIe 6.0) interface. The host interface (or CXL root port) holds the memory access logic controller and is integrated into the on-chip interconnect bus.

In Fig. 1.5, we illustrate the earlier example for allocating four workloads on disaggregated hardware resources. The CPU and memory are now decoupled with each other and available as independent resource pools. The workloads are free to be allocated CPU/memory from any of the available pools. As we can see, all four workloads can get the required CPU/memory from the same hardware resources as in the example shown in Fig. 1.3, where W4 could not be served. Our work in this thesis focuses on having separate pools of memory while the compute units still have a small amount of local onboard memory with them. The application can allocate memory from local DRAM or remote memory pools at any ratio, depending on the availability and suitability for best performance.

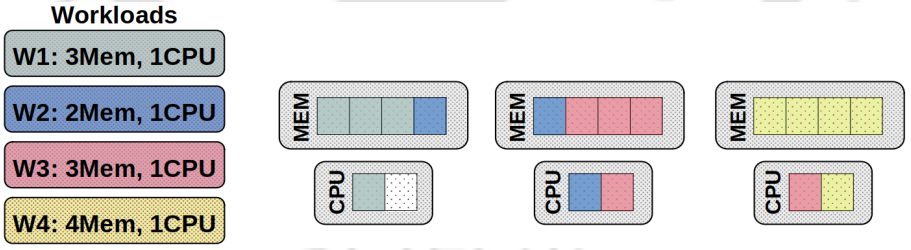


FIGURE 1.5: Improved resource utilization with Disaggregation

DMS eliminates the issues in traditional server system architecture and supports additional use case benefits. We explain below how a DMS can overcome those challenges:

- **Improved memory scalability and no under-utilization:** The workloads running on compute nodes can be allocated as much memory as required from the on-network memory pools. The CPU socket count no longer limits the memory scalability, and the remotely connected memory increases memory bandwidth and

capacity [18]. A large amount of memory is now available as remote memory pools. Thus, the applications running on the compute node need not over-provision the memory for peak requirements, as it can always be allocated whenever required. This reduces the memory under-utilization as memory is not stranded with different workloads or at different compute nodes.

- **Online memory upgrade:** Another feature of DMS is that the pooled memory capacity can be dynamically increased or upgraded with new technology without affecting the compute nodes and workloads running on it. This allows the server activities to continue without any shutdown during the upgrade process.
- **Improved energy consumption, data center refresh cycle, and TCO:** The improved memory utilization optimizes the overall energy consumption in a data center and reduces the expense of the cooling system. Further, the hardware refresh cycle is improved as the CPUs or memory can be upgraded independently, which normally have different refresh cycles. All these advantages reduce the data center TCO and support modern workloads to scale up easily with improved performance.

The recent trends in the industry and the latest research work go in the direction of seriously considering the DMS architecture for the next generation of data centers. Samsung had recently launched a CXL-based rack scale memory module [19] with all the software toolkits being open-sourced. It offers around 569ns on CPU-based load-store access to remote memory. The latest server processors from Intel and AMD also have added support for the CXL v1.1 [20, 21]. Even though the remote memory latency is 3x-5x slower than the local memory, it is still faster compared to RDMA-based memory expansion. The memory latency can be improved by implementing system optimizations like page migration [22, 23] and cache block prefetching [24, 25], some of which we will explore in the subsequent chapters.

On the other hand, authors in [26] presented a contrary view on the cost-effectiveness of CXL-based memory pooling. The argument is based on the higher hardware cost, lesser utility and requirement of new complex software systems for implementing disaggregated memory design. However, the experimentation conducted is very limited, which uses the VM traces from data centers to perform cost-based analysis and determine the utilization. Further, logical pools over a CXL interconnect have been proposed as a better alternative to physical memory pools [27]. The server design remains the same as the present, but each of them flexibly shares some part of its memory to form shared logical memory pools. The benefit is the early adoption of CXL and lesser TCO, as there is no need to invest in separate memory-only modules. However, the proposed solution fails to attract due to multiple reasons. Firstly, when a server flexibly shares the memory, it may withdraw the shared memory regions, forcing the shared data to move to another server. This causes irrelevant data movement in already congested networks.

Secondly, the software managing the shared memory within a cluster becomes complex. It will not be able to make optimal decisions due to multiple trade-offs (optimal memory usage, less network traffic, allocation near the computing node). Lastly, the memory capacity and bandwidth remain limited within a cluster. With each server node running memory-demanding workloads, the memory expansion will only be possible by adding more servers. However, servers usually hold enough processing power and only fall short of memory. Therefore, the idea is not feasible in the long run for reducing the TCO.

Similar to main memory, servers' storage capacity is also limited due to fixed PCIe slots. Therefore, disaggregated storage is also popular for creating common networked storage pools. The existing mechanism, like software-defined storage, can also combine the storage capacity of individual servers to create virtual storage pools. But it also adds extra layers of complexity and high CPU load, limiting the total storage that can be pooled. With storage disaggregation, the storage hardware is physically separated from the compute resources and is provisioned logically to servers based on requirements. Although DMS and disaggregated storage are based on the same underlying technology, their use cases differ. Cloud-based disaggregated storage is used to back up the contents of main memory to the shared networked disks (by emulating network-attached Storage (NAS)) and is not a replacement for DMS. The large capacity disks cannot support load/store operations at the cache block level without significantly reducing the system performance due to high latency access and lesser bandwidth. With DMS, a last-level cache miss can be directly forwarded to remote memory pools. The disaggregated memory and memory can both be implemented on the cloud data center. To be precise, both these can be implemented in any data center providing cloud services or high-performance computing (HPC) facilities. The memory/storage capacity and memory bandwidth requirement of cloud and HPC workloads can be really high, so the workloads can get significant performance gain with memory/storage disaggregation.

1.2.1 Performance Improvement with Disaggregated Memory System

In a data center, server systems running multiple workloads will have to share the main memory and cannot allocate all the required pages due to limited memory capacity. In this section, we compare the performance of three different systems. *Local*, a hypothetical system that always has free local memory available to allocate to workloads. *RDMA*, A state-of-the-art mechanism where server systems share free main memory with each other, and the memory transfer happens in large page granularity, which can be slow and degrade performance. *DMS*, a disaggregated memory system with pooled main memory that can be accessed as load/store at cache block granularity of 64KB using CXL. *RDMA* and *DMS* both allocate 50% of the workload pages at local and remote memory each. Figure 1.6 shows the performance difference of *RDMA* and *DMS* normalized against IPC

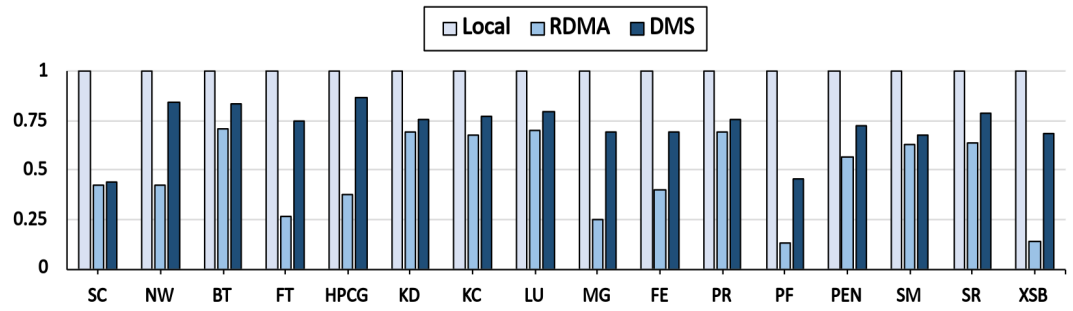


FIGURE 1.6: Performance Difference Local vs RDMA vs DMS

(Instructions per cycle) of local only system. We measure the performance of all three mechanisms over cloud and HPC workloads as mentioned in 3.2. As we can see, *DMS* performs much better as compared to *RDMA*, and performance reaches up to 80% of what we see with hypothetical system *Local*. The mean performance degradation across all the workloads is 52% with *RDMA* and just 28% with *DMS*. The significant reason behind this is the granularity of remote data access with this mechanism. The remote page access in *RDMA* is slower due to kernel-level calls that add long delays. Whereas *DMS* utilizes the CXL support for low-latency remote memory access directly on a cache miss without any system calls.

1.3 Challenges in Disaggregated Memory System

1.3.1 System Design

Memory disaggregation is still under research, and there is no clear understanding of how the final prototypes will be conceptualized [3, 10–12, 28]. The next generation of Intel [29] and AMD [30] processors are expected to support CXL 1.1 protocol for memory scaling. The limitation of CXL 1.1 is that it only supports a single CXL memory node to be connected with one compute node. The current state of research suggests that the DMS will be initially deployed at rack scale with a single-hop remote memory access (intra-rack) [10, 11, 28, 31, 32]. Currently, no agreement exists on the number of memory pools and compute nodes that can be configured together to optimally scale the memory bandwidth/capacity in a large-scale DMS. Further, remote memory address management, memory allocation, address translation, etc, are the key elements to be taken care of. This presents a key challenge to present a system design addressing all the issues and describing the working of various system components in a fully functional DMS.

1.3.2 High Average Memory Access Time

Although DMS solves memory scalability, the additional DRAM is present across an interconnection network. Due to this, the remote memory access time is 2x to 3x of local memory [8, 9, 33] in a single node (one compute node and one memory pool) configuration with CXL 1.1 protocol. In a scalable DMS configuration with CXL 3.0 protocol, the remote memory access is performed through a CXL switch. The network and remote memory bandwidth in large-scale configurations (such as in a rack) will be shared between multiple compute nodes. The remote memory access delays will be defined by the overall memory access traffic passing through the switch. The network and memory may face additional congestion/contention depending on the workloads running on different compute nodes.

For example, suppose memory is allocated to an application in a ratio of 30:70 at local and remote DRAM. In that case, the Average Memory Access Time (AMAT) will be significantly larger than a hypothetical system with a large enough local DRAM. If the memory traffic is distributed across local and remote memory in the same ratio, then with a remote access time of 3x of local memory access, the AMAT will still be 2.4x of local memory access time. However, if multiple compute nodes run simultaneously, the total switch and remote memory bandwidth will be divided among them. This will further increase the AMAT more than 2.4x, impacting the application performance. DMS presents a unique challenge to reduce the AMAT, which is somewhat similar to multi-tiered memory systems except for the scalability and memory/network bandwidth sharing involved in DMS.

1.3.3 Remote Memory Management and Control

In a large-scale DMS, multiple compute nodes can run a range of workloads with dissimilar memory demands. The compute nodes may perform remote memory allocations unevenly across all the remote memory pools. This may cause a bottleneck at some of the memory pool which gets more memory allocations and consequently faces more memory traffic. This increases the queuing delay at particular network lanes and the memory pool's controller queues, directly impacting the AMAT and the workload performance. Further, the impact of extra remote memory latency will differ on the workload's performance due to their memory access patterns and dependency on memory operations. In such a scenario, it also becomes essential to treat the remote memory requests of compute nodes differently based on their urgency/priority and deliver Quality-of-Service (QoS) to critical applications in a bandwidth-constrained system.

1.3.4 Architectural Simulation

As the DMS is still at the prototype stage, no standard tool exists for performance evaluation. The existing literature relies on completely different methodologies such as FPGA prototyping [13–15, 34, 35], emulation [31, 36] and simulation [37, 38], which has many shortcomings. The lack of standard methodology constrains the modeling and comparison of different state-of-the-art mechanisms and introduces unwanted challenges for the evolution of best designs.

1.4 Thesis Contribution

The work in this thesis deals with the research challenges described in section 1.3. Here, we briefly describe the contributions of our thesis. An architectural simulator is of foremost importance for the performance evaluation of DMS. As our first contribution, we develop a simulator *DRackSim* to evaluate the performance of large-scale DMS with a baseline system design. Our simulator supports multiple simulation modes with different details that can be used as required. Our second and third contributions focus on workload performance improvement through AMAT optimization in a large-scale DMS by implementing cost-effective hot-page migration systems. Considering the constraints in a DMS, an epoch-based page migration is implemented in the second contribution, where the focus is on improving data movement. In the third contribution, we focus on the shortcomings of the epoch-based mechanism and implement an on-the-fly page migration while fine-tuning the data movement through workload characterization. In the fourth contribution, we improve the QoS in large-scale DMS by two different mechanisms. Firstly, we propose novel remote memory allocation policies that can load-balance the memory access traffic across remote memory pools. Secondly, we implement different memory request scheduling mechanisms to understand their impact on QoS on different workloads. We discuss the contributions of the thesis as follows:

1.4.1 DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems

Research in disaggregated memory systems requires a simulator to evaluate new designs with the practical configurations of the latest memory systems connected through a network interconnect. This work develops *DRackSim* a simulator to model scalable hardware DMS. It models multiple compute nodes, memory pools, and a network interconnect for coherent memory access. An application-level simulation approach simulates an out-of-order x86 multi-core processor with a multi-level cache hierarchy at compute nodes. *DRackSim* follows a queue-based approach to model the network interface at

the end nodes and the central switch and simulates remote memory access at multiple granularities. *DRackSim* also models a global memory manager to manage address space in the remote memory pools. We integrate a widely accepted memory simulator *DRAMSim2* [39] to perform DDR4 simulation at local and remote memory by initiating multiple instances of the *DRAMSim2* memory system. We followed an incremental approach to validate the core and cache subsystems of *DRackSim* against *Gem5* [40]. Further, we model various use-case scenarios for disaggregated memory systems and evaluate their performance over various HPC and cloud benchmarks. We rigorously evaluate *DRackSim* for many configurations to simulate real-world deployment models and their impact on system performance.

1.4.2 A Practical approach for workload-aware data movement in Disaggregated Memory Systems

This work presents a hardware mechanism for page migration from remote memory pools to the local memory at the compute nodes. Hot page migration is mostly used in tiered memory systems to place hot pages from slow to fast memory and exploit the locality in memory access in those pages. However, DMS presents a unique challenge as multiple compute nodes share the memory pools, and the underlying hot page detection mechanism differs. Further, the network/memory bandwidth is shared among multiple nodes, which is not true with single-node, multi-tiered memory systems such as DRAM-NVM. Transferring hot pages from remote to local memory consumes extra bandwidth and delays the other cache misses being serviced from the remote memory. The memory read for a page, and the large packet size (e.g., 4KB page vs. 64B block) in the network significantly slow down the application performance. Our approach used an epoch-based page migration policy using an in-network hardware mechanism with efficient data movement between local and remote memory. Our design performs 10% to 100% better than traditional RDMA-based DMS that access remote memory at page granularity and 5% to 35% better than baseline DMS.

1.4.3 CosMoS: Architectural Support for Cost-Effective Data Movement in a Scalable Disaggregated Memory Systems

Epoch-based page migration has the advantage of transferring multiple pages together after every small interval, and the system overheads are low. However, it misses the benefits of accessing remote pages in local memory until they are migrated. The On-the-fly mechanism eliminates this problem by instantly migrating a hot page, but it usually has more system overheads. In this work, we propose *CosMoS* that is built upon the insights gained from the previous contribution to improve the performance

further. *CosMoS* is an architectural solution for on-the-fly page migration to support *Cos*t-effective data *Mo*vement in a *S*calable Disaggregated Memory System. *CosMoS* does not only predict the hot pages but also schedules and optimizes their movement between local and remote memory. The results show 20% performance improvement with *CosMoS* compared to state-of-the-art and 86% improvement compared to baseline large-scale DMS. On a single node, *CosMoS* performs 10% better than state-of-the-art and 57% better than the baseline.

1.4.4 Design and Analysis of Memory Allocation Policies for Disaggregated Memory System

Within a pooled memory (DMS), the global memory manager handles remote memory allocation. Upon receiving a request from a compute node to allocate remote memory, the memory manager selects a suitable memory node. Sub-optimal selection may increase memory latency, consequently affecting application performance. This study introduces a two-phase memory allocation policy for DMS, aiming to improve fairness in memory allocation. Fairness enforce equal distribution of memory allocations and memory access traffic to all the remote memory pools, resulting in reduced latency compared to conventional methods. Through comparisons across diverse benchmarks, our research demonstrates promising outcomes simply by integrating minor logic adjustments into the global memory manager for remote memory allocation.

1.4.5 QoS Management in Large-Scale Disaggregated Memory Systems

Accessing networked memory resources in DMS incurs additional access costs and significantly impacts performance, especially when multiple compute nodes share the same memory nodes. Fairness among compute nodes during network and remote memory queue allocation is crucial to mitigate these issues. This study delves into evaluating the performance of multi-node DMS using various queue allocation methods for network and memory bandwidth partitioning. Our proposal employs an in-network global memory controller that regulates the flow of memory requests from different compute nodes based on its priorities, thereby enforcing QoS. Additionally, we leverage the memory request rate of each compute node to determine their weights or priorities for different queue allocation methods. We evaluate all scheduling policies through a trace-based disaggregated memory simulator across diverse benchmarks with varying access patterns. Our findings demonstrate that these policies significantly influence average memory latency and system performance across different configurations.

1.5 Thesis Organization

The thesis chapters are organized as follows:

Chapter 2 presents a survey on the evolution of memory scalability mechanisms and the usage of remote memory for expansion. We also discusses the widely used solutions for improving the memory latency in multi-tiered memory systems. Finally, we briefly describe the tools and techniques for developing a new architectural simulator.

Chapter 3 presents *DRackSim* [41] , a simulator for performance evaluation of a large-scale DMS. It discusses the proposed system design of our baseline DMS and demonstrates the working of *DRackSim* on multiple use case scenarios over different configurations.

Chapter 4 presents a mechanism for workload-aware data movement [42] to improve the AMAT in a large-scale DMS. It describes implementing an epoch-based centralized page migration system that can effectively transmit hot pages from remote to local memory and utilize the locality in hot pages for future memory accesses.

Chapter 5 presents *COSMOS*, another mechanism to implement a hot page migration system built upon the learning gained from the previous work. It discusses the shortcomings of the state-of-the-art mechanisms. It presents a workload characterization to help design an on-the-fly page migration mechanism for accelerated data movement between local and remote memory.

Chapter 6 proposes new memory allocation policies [43] for DMS to eliminate network congestion and remote memory contention. The policies perform the load-balancing of memory requests among different memory nodes, effectively reducing memory latency and improving application performance.

Chapter 7 discusses QoS management in a scalable DMS [44]. With multiple compute nodes running a variety of workloads, some of the remote memory requests might need prioritized treatment over others. We propose a scheduling mechanism for the global memory requests from different compute nodes and compare multiple request scheduling algorithms that use simple heuristics for parameter tuning.

Chapter 8 summarizes all the contributions presented in the thesis and discusses a few future possible research directions.

Chapter 2

Background and Literature Survey

THIS chapter reviews the studies conducted in the past for memory scalability and improving the performance of multi-tier memory. Section 2.1 discusses different methods to scaleup the memory capacity and bandwidth. Some solutions focus on extending memory locally using multiple memory tiers, while others try to expand local memory through remote memory sharing. Remote memory is not a new concept, and it has been used in the past with DSM and memory-sharing servers for improving memory scalability. However, with the evolution in the interconnects, the way for remote memory access and its performance has changed significantly. This is mainly due to how interconnects are integrated into the overall system design to allow different kinds of remote memory access. Section 2.2 discusses the evolution of interconnects and protocols that drive different generations of solutions for memory scalability. In section 2.3, we present an overview of works related to DMS. In section 2.4, we discuss the different mechanisms proposed in the past for improving AMAT on multi-tiered memory systems. Section 2.5 presents a brief overview of mechanisms such as fairness and QoS, which are equally crucial for any large-scale system to rationalize the distribution of available resources. We finally discuss the simulation tools and techniques in section 2.6 for performance evaluation in a large-scale DMS. We also talk about the tools that can be used for designing new architectural simulators.

2.1 Existing Solutions for Memory Scalability

Multiple solutions exist for memory expansion that were widely used in the past or are currently being considered in the data centers. We briefly describe these approaches below in this section.

2.1.1 Scaling Memory Locally

Several system-level technologies are proposed to scale the memory capacity and bandwidth locally to the computer system. A hybrid main memory [45–56] system puts together different memory technologies with different features like cost, energy, performance, reliability, and endurance. As an example, a hybrid of DRAM-NVM (e.g., Intel 3D XPoint Optane, Phase Change Memory) is deployed to improve memory scalability and increase memory per core ratio while obtaining cost and energy benefits. However, NVM has higher read/write latency than DRAM and may lead to a loss in performance. The memory can be organized in multiple ways. One way is to attach both the memories in a hierarchical address structure where DRAM is used as a large page cache. Whenever a memory address is not present in page cache, it is looked at in the NVM, and the searched page is swapped with a victim page from DRAM. The other way is to attach the byte-addressable NVM as an extension of DRAM using a flat-address structure. In this case, the CPU can directly access any memory address from NVM on a cache miss. However, managing the data movement between two memories is important for optimal performance.

Similarly, high-bandwidth memory (HBM, which is a 3D stacked DRAM technology) and DRAM can be clubbed together [57–60] to increase the memory bandwidth for serving deep learning applications. HBM is costly but significantly increases the memory bandwidth and has low power consumption. It is mostly used to serve as a fast cache by bringing the frequently accessed data from DRAM to the HBM.

2.1.2 Scaling Memory Remotely

Most of the proposed techniques for remote memory extension are based on memory sharing between compute nodes in a multi-server environment like in data centers or HPC. The remote memory can be shared exclusively or in a shared manner among multiple compute nodes. In the former approach, a compute node is granted exclusive access to the free memory in a remote machine. In the latter approach, the memory of all the nodes is aligned in a single address space. Further, the compute node can access the remote memory in two different ways.

Remote Paging: A software-based solution is used to implement remote page swapping [61–63] and is widely used to access remote memory. This technique exclusively allocates the remote memory to a compute node from another machine. The remote memory adds another level in the memory hierarchy between the storage and the local memory. The idea is that the paging swap latency in the remote memory is significantly lesser than the slow disk paging. However, the existing local memory space is not really expanded, and the CPU cannot have direct access to it. Whenever the CPU accesses a memory address not present in the local memory, a page fault is issued to search in the next memory level. Due to this, a large percentage of remote memory access time is spent in the kernel stack for page fault handling and swapping. Further, the remote memory access is performed at large granularity (assuming 4KB pages), which takes more time to access remote memory and transfer pages through the network. Due to these overheads, the remote paging approach introduces high tail latency [63] and gives limited performance benefits, which is not feasible to meet application-level performance [64].

Distributed Shared Memory (DSM): In this mechanism, the memory resources of multiple nodes are aggregated to form a single large addressable memory called DSM. This approach also allows aggregating processors to form a computing cluster. Memory aggregation can be done by either software mechanisms or hardware mechanisms. The software approaches [65–68] rely on the virtualization of hardware resources through a hypervisor that aggregates the hardware of multiple nodes to form a single virtual machine with a single operating system. The hypervisor resolves remote memory access, which brings the memory pages from the remote nodes to the consumer node that runs the application. The processor at the consumer node can access the data once the page is copied to the local memory. This mechanism differs from OS based remote page swapping as the OS is running under the control of hypervisor. When the OS observes a trap due to a page fault, the control is shifted to the hypervisor that transfers the remote page to the local machine and then shifts the control back to the OS. However, the cache-coherence is managed on a page basis through the software handlers across the cluster, adding a large overhead. The software approaches add large latency overheads due to page faults and transferring large-sized pages. Hardware approaches allow direct access to the remote memory on a cache-miss, which is now part of the node's memory address space. However, it is required to manage cache coherency explicitly through extra hardware, which is costly. The cache coherency within the node is managed using the snooping protocols, whereas directory-based protocols are used for managing inter-node coherency using a directory controller. Although remote memory access is faster and does not require transmitting complete pages, the coherency management adds heavy network traffic and makes DSM difficult to scale beyond a certain limit. Thus, careful usage of remote memory is required to allocate and place memory pages. There are multiple systems proposed that use similar technology for memory scalability. Some important mentions are the DASH prototype [69], SGI Origin 2000 [70], FLASH

prototype [71], Scalable Coherent Interconnect (SCI) [72], SGI Altix [73] and CRAY XMT Supercomputer [74].

2.2 Interconnects

The concept of remote memory access is not new. However, the interconnect plays a major role in the system, which results in different access latency, access granularity, and overall performance. Here, we briefly discuss the evolution of interconnects that were used in the past to the latest release of CXL specifications for remote memory access.

Ethernet-based Interconnect: These are the most commonly used interconnects in data centers where the remote memory requests pass through standard network interface cards connected to the PCIe bus and then through an Ethernet link using TCP/IP protocols. The past mechanisms for remote memory paging or DSM were based on Ethernet-based remote memory access [65, 66]. However, the memory latency suffers due to software overhead and multiple protocol layers during the memory access. The remote paging latency can be around 160μs on a Gigabit Ethernet link, whereas the cache block access latency with hardware DSM is reported to be around 1-2μs. The hardware DSM system generates high coherence traffic, which limits its scalability.

PCI Interconnect: Some solutions were proposed using the PCI interconnect [75, 76], which is comparatively faster than the Ethernet. It does not require multiple protocol layers and saves time for protocol conversions and data encoding. However, PCI interconnects only support a single root domain and cannot be used directly to connect multiple nodes. The PCI switches with non-transparent bridges (NTB) allow multiple PCI root domains to be connected in a single subsystem with shared memory. It supports remote memory access at the page, cache block granularity, and a burst transfer mode using a DMA. However, this approach suffers from scalability due to the limitations of PCI switches.

RDMA based Interconnect: Remote direct memory access [61–64, 77] features as the most widely used protocol and interconnect standards in the current data center scenario. The software-based approaches in the past used TCP/IP protocol for remote memory access uses, which has the disadvantage of a heavy protocol stack that introduces long delays. RDMA protocol directly offloads the application data to the network buffers, bypassing the heavy kernel stack. This resulted in a significant reduction in the remote memory access latency. The RDMA protocol has multiple implementations, such as RoCE (RDMA over converged Ethernet), iWARP (TCP-based RDMA), Infiniband, etc. The InfiniBand is costly as it requires a specially designed network but performs the best among the three. RoCE, on the other hand, gives a good trade-off between the performance and the cost, as it can be implemented over the Ethernet.

The NIC (network interface card) plays an important role in RDMA as the application data is directly encoded at the NIC, reducing the software delays. An application using RDMA can use remote paging much faster than earlier approaches. It can also be used as an object-oriented DSM system [77], but the application must handle the cache coherency completely through programming constructs. RDMA memory disaggregation, also termed software/virtual disaggregation, is widely deployed in the present data centers for implementing remote paging mechanisms.

CXL Interconnect: The CXL interconnect allows low latency and high bandwidth access to the main memory by the compute nodes. CXL presents the latest interconnect specifications and protocol [16–18, 78] that supports memory scalability via in-network memory pools/nodes. It supports three different protocols: CXL.io, CXL.mem, and CXL.cache. CXL.io is fundamentally the same as PCIe 5.0 [79] and supports a non-coherent load/store interface for I/O devices. CXL has encompassed similar existing standards such as Gen-Z, CCIX and OpenCAPI. CXL.cache is the coherent protocol that allows accessing and caching host memory by the connected CXL devices with low latency and vice-versa. A common use-case scenario is to use it by the accelerators. The CXL.mem allows the host to access device-attached memory using load/store commands. The specifications support a CXL switch through which remote memory pools can be attached to scale memory capacity and bandwidth. The important distinction is that, it decouples the compute from the memory and has extensive use-case scenarios in data centers with reduced TCO. The low latency access is supported by the integrated on-chip interconnect (OCI) that also allows coherent access to remote address space. The CPU can directly send the memory requests to remote memory with the help of a remote memory controller that implements the network protocol in the hardware to eliminate software stack delays.

2.3 Disaggregated Memory Systems

In this section, we present a survey of the some of the existing solutions relevant to our work solutions for RDMA-based software memory disaggregation and CXL-based (or similar) hardware memory disaggregation.

- Lim et al. [9] proposed one of the initial disaggregated designs that provisions separate CPU and memory blades. Memory blades were connected to compute blades via an enclosure's I/O backplane using a PCIe bridge rather than on-chip controllers. The page-level remote access is initiated through a page fault in the OS, which is trapped to a virtual memory manager (VMM) or hypervisor, triggering a page swap with the remote memory. Page access is slow but the workloads

can take advantage moving data at granularity (pages) and locality of future memory requests. The cache-level access is supported through a new hardware called "cache-coherent filter" to initiate reads and writes access to remote memory. The remote memory address translation is also handled at the VMM, which adds/removes the memory mappings whenever remote memory is (de)allocated.

- Hou et al. [75] presented a remote memory-sharing prototype with multiple nodes. Each node is connected to the backplane via a PCIe adapter. The backplane is connected directly to the primary host via a PCIe bridge and to remote hosts via a non-transparent bridge (NTB). NTB makes it possible to connect multiple PCIe domains and provides isolation to different nodes with shorter communication paths to other hosts. It supports direct and DMA access to the remote memory. With direct access, applications access remote memory in the same way as its local memory. The memory controller determines if the cache miss belongs to a remote address and forwards it through the PCIe switch to the remote host. In DMA access mode, a DMA engine performs the transfer either in block or burst mode. Before sharing memory, multiple hosts negotiate with each other on the shared memory region and set up a mapping table on its side of NTB. The OS at each node does not have the visibility of global address space and is completely managed through the memory module driver.

- In "`Marlin`" [76], authors used a similar approach as the work discussed above (using NTB) at the physical level, but the implementation is quite different. Each machine has visibility of the complete address space of every other attached node, which acts like a global memory pool for each machine. The CPU-based direct remote memory access is supported by explicitly sending MSI read/write interrupts, and the interconnect is not coherent in itself. In the DMA mode, the DMA engine can directly initiate memory copy between any two address spaces, as all machines have global visibility of the address space. At the control plane, the memory address space of each machine is mapped by the management host to its own physical memory address space during system booting. It then exposes the whole address space to all the machines connected to the switch.

- Gu et al. [63] designed a software solution "`INFINISWAP`" for memory sharing among nodes in a cluster. It is based on RDMA and can be deployed over InfiniBand or RoCE networks, which makes it feasible in current commodity hardware without any extra hardware. The only requirement is the network adapters with RDMA support at the nodes. A network block device and a daemon program enable memory sharing without central coordination. Whenever there is a shortage of local memory, the "`INFINISWAP`" block device starts writing pages from the swap partition to the remote memory using RDMA writes. On subsequent access, those pages will be served directly from remote memory rather than going to disk.

The (de)allocation at remote memory is performed at a larger granularity with fixed-sized slabs (collection of pages). The daemon handles the distribution and placement of slabs in the remote machines. For optimal usage of remote memory, pages are removed from remote memory when the paging activity goes below a threshold.

- Montaner et al. [67, 68] proposed "`MEMSCALE`", a mechanism using commodity interconnect to connect multiple nodes. It extends the shared memory beyond a motherboard without considering the coherency, effectively saving the network bandwidth and increasing the remote memory access speed. It is assumed assume that the computing needs of most applications can be fulfilled with a large number of cores present within a single motherboard but only require extra memory, which is mostly unavailable. This effectively means coherency messages are limited to one node and coherency in remote memory need not be maintained explicitly and can be easily handled through write-back caches. HTX, which is an extension of AMD Hyper-Transport for scaling out, is used as the interconnection technology for board-to-board communication. They proposed a new memory controller to access remote memory added to the motherboard through HTX interconnect as an I/O unit. CPU can directly issue remote memory requests, which will be forwarded by the source remote memory controller to the destination remote memory controller. A single global address space is visible to all the machines, and the OS is responsible for knowing the location of free memory, memory reservation, and disposal in any part of the global address space.

- Chang et al. [13] demonstrated a prototype by extending an on-chip interconnect for memory-sharing with other nodes within a rack. They proposed a custom module connected to the on-chip interface that joins other nodes to form a single cache-coherent domain. The architecture allowed remote memory access at the cache line as well as bigger data blocks. The new 4-layer module packs on-chip-interconnect (OCI) packets into transaction layer packets, queued for routing using lightweight network layer protocol, and sent to the physical layer for remote access. The coherent remote memory is accessed in the same way as local addresses. This prototype allowed the remote memory to be accessed at nearly 4 times the access latency of local memory, which is way better than previous approaches.

- Dong et al. [15, 80] presented "`Venice`", which proposes a sharing model that integrates the resource-sharing fabric directly on the chip as mentioned in the previous approach mentioned above. However, it implements a family of architectures to access remote resources, including memory, accelerators, NIC, and other I/O devices. It also implements a multi-channel approach consisting of a cache-based access channel, an RDMA channel, and an asynchronous Queue-pair channel to access remote memory. An adaptive library is implemented at the application level

to make intelligent channel choices based on the communication requirements of the workload.

- Novakovic et al. [14] designed a similar solution, "`Scale-Out NUMA`", that borrowed positives of cache-coherent NUMA (ccNUMA) [70] and RDMA networks to come up with a hardware-based solution for rack scale memory sharing. A new hardware module named remote memory controller is supported by on-chip logic to handle cache coherent memory interface via a private L1 cache, to perform remote memory operations.

- Cao et al. [81] described "`XmemPod`" a system where the authors observed the requirement of efficient memory sharing on the node itself using containers or VMs before it requests memory from the remote node. High memory imbalance can frequently arise between VMs on the same node, and solutions like memory-ballooning [82] do not perform well and are too slow. This work proposed a memory-sharing cluster over RDMA networks, but its main contribution is considering memory sharing within a node. A remote memory manager pool manages the additional memory requirements using a remote swap device. Like in Infiniswap, address size is divided into fixed-sized slabs comprising multiple pages and can be placed on multiple nodes to balance memory allocation. However, it uses round-robin, weighted round-robin, or random placement rather than simply putting the slabs in a node with free memory. Further, a page service manager is implemented inside all the VMM for swap-in and swap-out operations, which also takes care of memory sharing among different applications on the same node.

- "`dRedBox`" [34, 83, 84] proposed a rack-scale prototype that demonstrated the hardware DMS at rack-scale for the first time by using decoupled CPU and memory units. It is based on resource bricks architecture that uses different resource bricks for CPU, memory, and accelerator. A resource tray with fixed slots can host these bricks in any combination. The connectivity among different bricks within a tray is done using electro-optical switches. The CPU bricks have a bare minimum amount of memory only to fulfill the basic OS requirements and most memory requirements are satisfied from the memory bricks. A disaggregated memory controller (DMC) is introduced for the CPU to intercept the memory accesses and prepare it to send through the electro-optical data plane. A memory lookup structure translates intercepted memory requests to a particular memory brick. Similarly, a DMC on the memory brick accepts the requests arriving at the network port, passes them to the local memory controller, and sends back the responses to the respective CPU brick.

- "`DirectCXL`" [85] utilizes the newly introduced CXL protocol by developing a prototype with CPU servers as memory hosts and passive remote memory blades

comprising only DRAM memories along with a CXL-enabled memory controller. A CXL-based interface with a switch supports the connections between multiple CPU and memory units. However, there is no sharing of resources allowed between different CPU hosts. The authors also implemented a kernel module to make the CXL-attached memory available to the user during the OS start-up through memory-mapped files.

- "`Clio`" [12] is another memory pooled system based on memory blades. The unmodified CPU servers are used as the compute nodes with memory bladed as the memory servers. An ASIC handles the remote memory accesses and control operations for the DRAM in memory blades. The (de)allocation and read/write of data to remote memory is performed over the Ethernet using a library. Further, Clio also supports data sharing between compute nodes.

- Finally, "`LegoOS`" [10] proposes a split kernel approach to develop OS for DMS. Although a hardware-based system is not evaluated, the OS assumes different CPU/memory blades and deploys a monitor on each blade to manage it. The local memory on the CPU blades is treated as an extra level of caching rather than in a linear address structure, and manages it through CPU monitor software. It keeps the TLBs and page tables entirely at the memory blades, while the local memory is virtually indexed and tagged to provide separation of local/remote address space. Each application can be deployed on multiple blades of any type. The data sharing is disabled between the CPU blades, and the programmer needs to explicitly use message passing if required. LegoOS supports Ethernet and RDMA network stacks and maintains a data transfer granularity of 4KB.

Although multiple solutions have been proposed in the past, there is no consensus yet on how the real DMS will evolve for deployment in the next generation of data centers. However, decoupling the compute and memory is a promising direction and a design choice supported by the latest trends in developing interconnect standards. Therefore, our system design also assumes decoupled hardware resources for compute and memory. In the next chapter, we discuss our baseline DMS design and detail the approach taken for performance evaluation.

2.4 Reducing Memory Delays with Page Migration

The hybrid or NUMA memory systems often implement a page migration to move hot pages from slow memory to fast memory. In NUMA, hot memory pages are moved between multiple sockets to be resident in the nearest DRAM to the CPUs accessing those pages. In multi-tiered memory systems, the hot pages are moved from slower

memory (NVM) to faster one (DRAM) to reduce memory access delays. It exploits the locality of future memory accesses in the hot pages migrated to faster or nearer memory. Some other proposals to reduce memory delays also focus on cache block prefetching, which tries to access the cache block before the CPU requires it. Although cache block prefetching can effectively hide memory latency, we focus on page migration in this thesis, as data center workloads show good locality in their memory accesses [86, 87] and can be exploited better with page migration. DMS are not much different from the hybrid memory systems except few differences. Memory pooling in DMS opens up a large remote memory address space that expands the host memory to terabytes. The presence of a switch and deployment at scale makes DMS different from the typical multi-tiered hybrid memory systems like DRAM-NVM. The underlying approach can be used with appropriate modifications for DMS. This section presents a brief survey on the state-of-the-art page migration techniques for reducing the AMAT in multi-tiered memory systems. We also discuss the different techniques and complexities of implementing a page migration system.

- Yujuan et al. propose "`APMigration`" for a hybrid DRAM-NVRAM memory system aligned in linear address structure. An adaptive page migration is implemented to store frequently accessed hot pages in DRAM and cold pages in NVRAM. It eliminates the invalid page migrations that create a ping-pong movement between two memories due to incorrect hot/cold page identification. Their unified hot page recognition mechanism compares the relative hotness of hot and cold pages in DRAM and NVRAM, respectively, and then decides if a page should migrate or not based on the expected revenue and access patterns.

- In [23] authors observed that the due to address translation overhead, superpages are commonly used. However, migration overhead is huge due to long access and transfer delays and is highly energy inefficient. Due to the aforementioned issues, they propose a new memory management "`Rainbow`" that can identify the hot super pages and then support a light-weight page migration of hot regions within those super pages. Heo et al. in [88] also added support for migrating huge pages and investigated the migration policy space for huge pages. They propose "`AMP`" that dynamically selects a page migration policy based on the page access patterns of different workloads running concurrently.

- Na et al. proposed "`PFHA`" [89] for a DRAM-PCM hybrid memory system focused on high performance and low energy consumption. Due to its high density, the PCM is suitable for memory scalability. However, the large write latency, write power consumption, and lesser write endurance require smart data placement to

exploit this hybrid system. It proposes a dynamic hardware migration that utilizes the relation of periodical read/write access frequencies to predict page hotness. This mechanism initiates a self-migration to DRAM on write hot pages to concentrate write pages on DRAM and saves energy.

- In [90], authors proposed "`AutoTiering`" and explored the design space for a multi-tiered memory system with DRAM and SCM (Intel's DCPMM). They observed that the existing page migration implementation in Linux is implemented for NUMA, which migrates pages between NUMA nodes, all of which are DRAM-based. The same migration mechanism does not work efficiently with hybrid memory with different features. The new implementation mechanism considers both access locality and hot page placement in the right memory tier. Secondly, they implement an auto demotion mechanism for moving cold pages to SCM to utilize the limited DRAM capacity effectively. During demotion, it also compares the hotness of the page being demoted to the victim page in SCM.

- Hasan et al. proposed "`TPP`" [33], an OS-level application-transparent page placement and migration mechanism. They evaluated its performance on CXL-enabled DSM with a single node (with CXL 1.1 interconnect support). This work presented a workload characterization tool "`Chameleon`" and deployed it on a production data center to get the page-level access information and its hotness longevity. The tool uses precise event-based sampling (PEBS) to collect page access data and pass it to the worker application that decides to migrate a page. PEBS uses hardware counters in the processor and can track certain system events such as loads, stores, cache misses, etc. "`TMTS` [91] is another similar approach that proposed a page migration mechanism based on multi-tiered memory using PEBS hardware counters. The sampling rate has high overhead, and both solutions target a low sampling rate of around 2%-5% and get a good trade-off for performance.

- In [23, 88], authors focus on predicting hot regions within huge/super pages and perform migration at smaller granularity by adding special hardware support. It implements two levels of hotness prediction, one for the huge pages and then to predict hot regions within the predicted huge pages. The migration is performed at 4KB page granularity to save bandwidth and avoid starvation to subsequent cache accesses.

- Komareddy et al. [38] proposed page migration on a scalable DMS for the first time using a global memory controller. However, the epoch-based migration misses out on probable benefits due to migration in large batches. Further, the mechanism lacks any intelligence and leaves it to the users to set migration parameters.

- Finally, Daemon [37] proposed hardware support by implementing page compression to reduce the response packet payload size during a page migration and partitioning the bandwidth between page and regular memory accesses to reduce the delays. The bandwidth partitioning is performed at a coarse granularity and has a minimal impact. However, the page compression resulted in a significantly improved performance.

- Finally, Shuang et al. [92] apply hot-page migration to cloud computing platforms and devise a hot-page capturer for virtual machine migration to reduce the remote page faults during a restart at the remote node. Page migration is also proposed for systems with software/virtual DSM [25, 63, 64, 93, 94]. However, page migration is a compulsion in these systems as it does not support cache-based access. These mechanisms mostly focus on page prefetching, choosing remote nodes for page placement, etc.

Moving pages in DMS can have multiple challenges. Firstly, no single memory manager has full control of both local and remote memory in DMS. Page migration is prevalent in other multi-tiered memory systems that use DRAM and NVM as fast and slow memory tiers, respectively. However, a single memory manager controls all the memory address space. A memory manager or TLB can easily predict hot memory pages in the slower memory, which is impossible in DMS. Secondly, data-centric workloads with large memory footprints expand to tera-bytes of memory, and significant metadata is required to track all the memory pages. Hot page migration is usually performed using a kernel module in the operating system which monitors the page activity to predict hot pages. However, the OS-based techniques will not work well for hardware DMS, which has a significantly big second-tier memory, making page tracking difficult. The data-center applications expand to tera-bytes of memory and will need a dedicated hot-page tracker with the least amount of meta-data. The other hardware-based techniques for hybrid memory systems have many limitations when implemented on a scalable DMS, which we discuss in subsequent chapters. Thus, there is hardly any scope for implementing existing techniques on hardware DMS. Further, there are multiple mechanism that exists for page migration, which need to be exploited for their suitability in the DMS architecture.

2.4.1 Page Migration Overheads and Parameters

Migrating pages requires updating address translations in the page table entries (PTEs) with new mappings. While updating the PTEs, TLBs are locked to perform invalidation of migrated page entries (known as TLB shoot-down), during which OS interrupts the user application and issues an inter-processor interrupt (IPI) [95] to other cores with the same page entry (also to other compute nodes, in shared memory approach). Similarly,

cache invalidation is required for the blocks with old physical tags. The invalidation process (address reconciliation) is expensive and introduces long CPU stalls, and can take around 4-13μs depending on the number of cores, as pointed out by [22, 96, 97]. Further, migration generates extra TLB misses for re-accessing invalidated entries, taking 60-80 cycles per page on a TLB-miss [98]. Lastly, each page transfer from remote to local memory takes around 1.2-1.5 μs and delays the subsequent block access when the page is being read from memory and transferred as a large packet in the interconnect.

2.4.2 Different mechanisms for Hot Page Migration

Hot page migration can be performed in two ways, each with its own advantages and disadvantages. An Epoch-based page migration collects multiple hot pages and migrates them in large bunches from slower to faster memory. On the other hand, On-the-fly page migration moves a page as soon as it is detected as hot in the slow memory. We explain the different mechanism for page migration below:

Epoch-based Page Migration: It primarily requires three parameters. Firstly, an epoch length decides how often the pages should be migrated. If it is small, frequent page migrations introduce continuous CPU stalls and excessive overhead. If it is large, all the future accesses to hot pages will be completed at slower memory even before the migration. Secondly, the hotness threshold describes the minimum criteria for a page to be migrated that can be identified in various ways, such as access count/frequency to a page or other ways. If the threshold is high, it will not migrate many probable hot pages. If it is low, many moderately accessed pages will also become hot, increasing the network traffic and causing more starvation to block-level accesses. However, the threshold varies for different workloads, and the decision for migration should be taken based on the expected benefits from migration rather than a compulsion. Migrating useless pages also means the system is trying to overkill the benefits of page migration. Many pages might not even be accessed after migration, evicting more local victim pages in turn. Lastly, the number of pages to migrate (NPM) describes how many pages should be migrated together. If NPM is less, there will be frequent interrupts with CPU stalls which also invalidate TLB entries for each batch of page migration. If the batch size is large, the benefits of migration will be lost due to extra wait before the pages are brought to local memory.

On-The-Fly Page Migration: This mechanism only requires a hotness threshold for predicting hot pages and moves a page instantly when it is predicted hot. The benefit is that most of the future memory accesses to the hot pages will be performed in the faster memory, and it does not lose out on the benefits while waiting for more hot pages as in the epoch-based mechanism. However, the page migration overhead can be significantly

large due to frequent updates in the page tables and TLBs. This can, in turn, also slow down the application.

2.5 Fairness and QoS in Large-Scale Systems

In any large-scale system, multiple CPUs/cores/sockets compete for shared resources such as shared last-level caches or main memory [99]. However, if there is no check on the distribution of those resources between them, one or the other will face contention in the system in many ways, impacting the workload running on them. Therefore, it is important to enforce fairness and QoS for equal distribution of resources and to provide fair execution to prioritized applications. In a disaggregated memory system, multiple remote pools host the memory requirements for different compute nodes. If the memory allocations and requests are distributed unevenly, they will face longer delays on a pool with more traffic than the others. Similarly, If a node is running a high-priority application, it should also be able to complete its memory request faster than the others to improve the response time on a high-priority application.

2.6 Performance Evaluation of Large-Scale DMS

Without commercial hardware, performance evaluation is only possible through prototype implementation or an architectural simulator. However, no other architectural tool exists for evaluating large-scale DMS. We choose to develop a new simulator as part of this thesis work. This section briefly overviews widely used approaches required for designing an architectural simulator.

2.6.1 Simulation Tools and Techniques

DMS focuses on memory access simulation, as the main memory references are partitioned among local and remote memory. Here, we discuss different methods for memory system simulation that can be utilized whenever necessary. The crucial point is how the main memory traces are generated for simulating the memory system behavior and performance evaluation of the workload. The following two simulation methods exist for the simulation of such a system.

Trace Driven Simulation: Traces of the memory requests are collected before the real simulation and written to permanent storage in a Trace File. The tracing can be performed over a real machine or an emulated non-existing one. The only information required is the type/address of memory operands. Trace Files can grow big on real

workloads, which serve as an input to the memory system simulator (simulating memory hierarchy Cache, DRAM, etc) for performance analysis. The memory simulator can run on any machine, even though the collected traces belong to a different ISA, as it only needs to analyze the memory traces for any base architecture. The trace-based technique is useful as it is less complex, and the same trace files can be reused again. However, the trace is only a snapshot and cannot represent the dynamic behavior of multiple threads. This leads to losing details, which is important for the result's reliability. The heavy storage requirement can also limit the usage of trace-based simulation. However, this disadvantage can be eliminated by simulating the cache hierarchy before the trace generation. This will only generate the main memory traces and significantly reduce the disk requirement by more than 90%. However, this also loses the details and cannot represent the dynamic behavior of caches and TLBs.

Execute Driven Simulation: In this technique, the simulation is followed by the actual execution of a program. It can be carried out either directly through a memory simulator or by coupling a memory simulator with a CPU-driven front end that drives the memory model. In both cases, the memory or instruction trace file is generated on the fly by executing the program's binary. Other ISAs can also be emulated by performing dynamic binary translation on the traced instructions before the actual simulation is performed. This technique is accompanied by the implementation of a timing model in the simulator. This may significantly increase the simulation time, but there is no overhead of storing traces on the disk. Execution-driven simulators also increase the simulation development time but allow capturing more details of a computing system through simulation.

2.6.2 Binary Instrumentation with Intel PIN

Instrumentation tools are used for adding a control code to the program binary that interferes with the program to monitor or modify any program behavior. This tool is extensively used during application code development to identify and eliminate numerous problems. Some of the instrumentation goals can be gathering metrics, automated code debugging, detection of error code and memory leaks, etc. Intel PIN [100, 101] specifically deals with the dynamic binary instrumentation that occurs just before or after the code is executed through a concept called Just In Time. Pin supports multiple operating systems like Linux, Windows, Android, and macOS, while the instrumentation can be performed on IA32 and x86-64 instruction set architectures (ISA). Through this framework an arbitrary code can be attached (using C or C++) at arbitrary places in the executable. When the execution of binary takes place, it dynamically attaches the code to perform the required functionality.

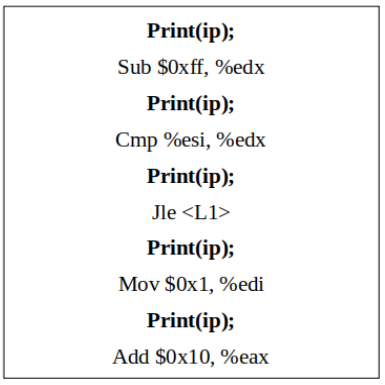


FIGURE 2.1: Binary Instrumentation through PIN

An example of binary instrumentation through PIN is shown in Fig. 2.1. As the instructions in the program execute, the newly added code (instrumentation) is attached to the binary. In this case, the instrumentation code is to print the instruction pointer of the next instruction, as shown by "**Print(ip)**" in the code snippet. PIN supports a rich API that supports writing customized PIN tools from a wide range of instrumentation routines and libraries. Similarly, PIN allows instrumenting every aspect of the instruction from the executed binary, such as instruction operation, its address, memory operands with their address, register dependencies, branch or not, etc. All this information can be utilized for architectural simulation at any level of abstraction.

2.7 Summary

In this chapter, we discuss multiple methodologies used in the past or are being used for the memory scalability of server systems. We classify those mechanisms based on the way the memory is expanded, which is either locally or remotely. We mainly focus on remote memory scalability and discuss the evolution in interconnect technology that has a major impact on the memory system performance for its bandwidth and latency. DMS is not a new technology and it is the outcome of advancement in remote memory expandability mechanisms and interconnect technology that supports low latency and high bandwidth remote memory access at fine granularity. We surveyed the recent research on memory disaggregation and discussed various software/hardware or co-designed mechanisms for improving memory latency through page migration. Finally, to overcome the experimentation challenges of a large-scale DMS, we study different mechanisms for computer system simulation. We eventually used Intel PIN [102] to build our simulator to evaluate DMS, described in detail in Chapter 3. In Chapter 4 and Chapter 5, we present our hardware-based page migration systems that are suitable to be used in a CXL-enabled DMS. Chapters 6 and 7 present methodologies for improving fairness and enforcing QoS in large-scale DMS, respectively. Finally, chapter 8 will summarize the

thesis work and give pointers for future research work to improve the performance of DMS.

Chapter 3

DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems

This chapter discusses the architectural simulator *DRackSim* [41] for performance evaluation of large-scale hardware DMS. We start by explaining the design of the proposed hardware DMS that will remain the same throughout this thesis. We then talk about the simulator functionality and different modes of simulation with all the abstractions that are considered to build them. We also explain in detail all the components modeled in *DRackSim* to simulate a scalable DMS. Finally, we validate *DRackSim* rigorously and perform a wide design space exploration to show the simulator's capabilities for system management, adding architectural optimizations, and handling interconnect challenges.

3.1 Introduction

Considering DMS's lack of commercial availability, a simulation framework is required to translate research ideas into working models by getting enough insight into the system's performance and trade-offs. The focus of DMS deployment is towards cloud data centers and HPC facilities and is expected to be configured at rack-scale with multiple compute and memory nodes. A rack-scale disaggregated simulator primarily requires modeling

compute nodes, memory pools, and other components like memory managers and a cache-coherent interconnect.

This work introduces a simulation framework that models an environment with all the required components mentioned above. *DRackSim* follows an application-level simulation approach that uses Intel's PIN platform [102] and introduces two modes of simulation: a trace-based and a cycle-level simulation model. Although memory trace simulation is fast and easily scalable, it usually lacks modeling details and restricts design space exploration. Therefore, we also build a cycle-level detailed CPU-driven model that uses an instruction stream produced on-the-fly through dynamic binary instrumentation of the workloads with Pintool. The instructions execution is then simulated on a detailed x86-based out-of-order simulation at compute nodes. Both simulation modes support a full spectrum of single to multi-core processor architecture and a multilevel cache hierarchy.

DRackSim models all the necessary system-level components to explore the system design space for DMS and provides an opportunity to evaluate new designs rapidly. The compute node model includes a memory-management unit (MMU) for address translation and an address space management unit similar to an OS memory manager with 4-level page tables for allocating memory pages at local or remote memory. The interconnect is based on a queue simulation model and can be configured to meet the bandwidth and latency of the target interconnect hardware by mapping its network parameters. We modify and integrate an open-source cycle-accurate memory simulator, DRAMSim2 [39], for simulating DRAM at compute nodes locally and at remote memory pools. The simulator is designed from the top down to simulate multiple compute and memory nodes where a global clock maintains the time ordering of global events such as network access and remote memory access. It uses a multi-threaded approach (one for each compute node) to perform fast and scalable simulations even with many multi-core nodes and memory pools. As CXL3.0 interconnect is not yet available and DMS based on this is still in the prototype stage, we use Gem5 [40] to perform modular validation of different components in the simulation framework and perform rigorous testing for the reproducibility of results and portability. The main contribution of our work is as follows:

- We introduce *DRackSim*, an application-level simulation framework for rack-scale DMS that can model multiple compute nodes and memory pools with necessary memory management and interconnect simulation.
- We present two simulation modes in *DRackSim* for compute node simulation: a memory trace and an out-of-order CPU with different levels of details that can be used wherever appropriate.

- We perform modular validation of all the components over a range of single and multi-threaded benchmarks. Finally, we compare the performance of large in-memory workloads on *DRackSim* over various configurations and use cases to show the impact of memory disaggregation and slowdowns due to congestion and contention.

The rest of the chapter is organized as follows: In the next section, we discuss the motivation behind developing a new simulator and the prior work with their limitations. Next, we discuss the DMS based on which we model different components of *DRackSim*. Section 3.4 discusses the design and operations of *DRackSim*. We discuss the validation aspect in section 3.5 and use case experiments over vast configurations in section 3.6.

3.2 Motivation

The first question that arises while building a new simulator is: why yet another simulator? Hardware memory disaggregation is a relatively new research area and an emerging memory system. Although software disaggregation and its real-world implementations [63, 64, 103] have been there for a while now. These systems differ from hardware-based CXL disaggregation and only support page access to remote memory. The concept of remote memory pools (or memory blades/ memory nodes) with coherent access over memory-semantic fabrics is new and yet to be commercialized. Some simulation/emulation environments exist for hardware memory disaggregation [34, 35], but all are limited to evaluating only a single node. Lim et al. emulated hardware memory disaggregation on top of the XEN hypervisor [9] by marking some allocated memory pages as remote in VMM page tables while adding fixed network latency on memory access. However, emulation platforms cannot predict true memory latency, as local and remote accesses both use the same physical memory. On the other hand, some hardware prototypes were also built using FPGA's [13, 15, 34, 35], but evaluation on scale is an issue. Some proprietary disaggregated memory implementations exist, such as Intel RSD [104], Facebook's Open Compute architecture [105], etc., for which little detail is available in public domain. Modifying open-source architectural simulators such as Gem5 [40], MARSSx86 [106], Sniper [107], etc., requires a vast effort to simulate disaggregated memory models while considering their inability to model multiple compute nodes. Further, it will also require the modeling of remote memory managers for remote address space and significant modifications to existing memory managers in compute nodes. In Daemon [37], authors heavily modify Gem5 to model memory disaggregation but test a scaled-up environment with artificial network traffic. Also, the code is not made publicly available. Hence, building a dedicated simulator is worth an idea. Further, it is necessary to model DMS for a scalable environment to study various factors limiting performance. With

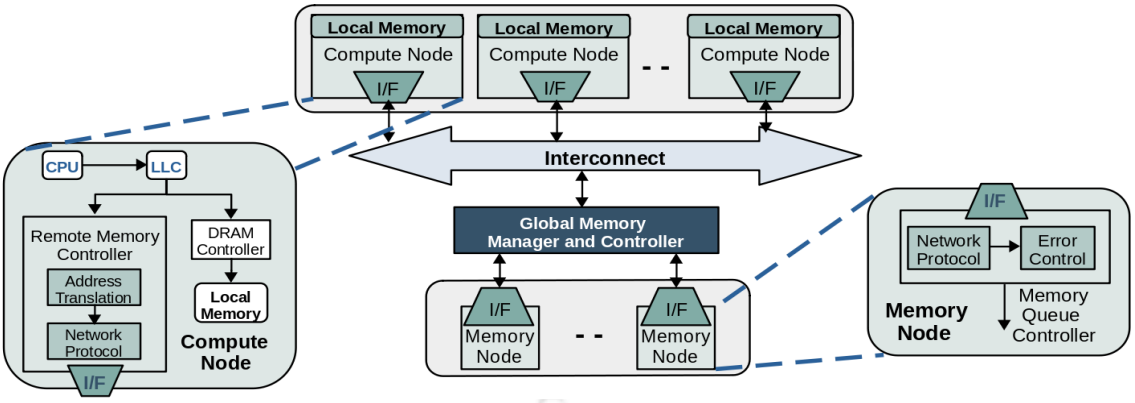


FIGURE 3.1: Overview of DMS and its Interface with Host Compute Nodes

more compute nodes, memory access traffic will generate congestion on the network and contention in queues at remote memory pools. The interconnect latency and bandwidth will become crucial for application performance. It will also allow for a deeper study of memory management in a scaled-up environment. To our knowledge, no other simulator model fulfills such requirements, which is the primary reason behind developing *DRackSim*.

3.3 Baseline Hardware Disaggregated Memory Systems

Fig. 3.1 shows the abstract view of a hardware DMS consisting of compute nodes, memory nodes, interconnect, and a global memory manager/controller. *Compute nodes* are the primary focus for performance evaluation, have a small amount of local memory, and rely on remote memory for most application requirements. *Remote memory pools/nodes* are the memory-only nodes from which a central memory manager allocates memory on demand to the compute nodes. As shown in Fig. 3.1 *Global memory manager* is an in-network centralized remote memory manager and controller that performs memory-related activities, such as memory allocation, revocation, etc., in the remote memory address space. An *Interconnect* is the binding fabric that supports coherent memory access from the compute nodes in allocated remote memory address space. The compute nodes are interfaced to the network interconnect through a remote memory controller, an addressable hardware module similar to a local DRAM controller. The LLC misses that belong to the remote memory addresses are forwarded to the remote memory controller, which then performs address translation and implements network protocol in hardware before sending it to the physical layer interface (root port in CXL). A similar controller is present at the memory nodes to decode the network memory request packets and send back the responses to CPU nodes.

3.3.1 Remote Memory Organization

The scalability in hardware disaggregated memory will largely depend on the remote memory organization schemes and the way remote memory is exposed to the system at compute nodes. Remote memory can be organized using a shared memory or as a distributed approach, as shown in Fig. 3.2a and 3.2b, respectively. In the shared memory approach, all the remote memory address space is visible to the OS at compute nodes with a single global address. The application can be allocated a page at any address while supporting page sharing between nodes, with the owner node acting as a home agent for that page. Such memory organization will still generate excessive memory coherence traffic (to memory nodes and other compute nodes) in the network due to shared pages, limiting the system's scalability. The compute nodes will also require frequent communication with a central authority (global memory manager) to prevent an address conflict during a remote page allocation, creating a bottleneck in the global memory manager while serving page allocation requests. However, the advantages are that the applications can also span multiple servers to meet the computing requirements.

On the other hand, hardware DMS does not need to span application threads across multiple nodes. Modern high-end multi-processor systems support tens of cores, and most data center applications can fulfill their computing requirements within a single node. Even if the workflows get spanned across multiple nodes, they run independently with occasional communication. The distributed memory organization can fulfill such requirements without an allocation bottleneck. In this organization, remote memory address space is not initially visible to compute nodes. The remote memory can be reserved in large-sized chunks, say 4MB-16MB (easy to evict and deallocate chunks), whenever a node requests. This approach will also require a global memory manager to reserve remote memory for a compute node. However, allocation in larger chunks will not create a bottleneck because of the lesser frequency of requests. Considering these benefits, *DRackSim* models a distributed approach and uses extra hardware for address mapping at the compute node's remote memory controller, as shown in Fig. 3.3. This mapping is required for translating the local physical addresses to the remote physical address for the allocated remote memory chunks, which differs from virtual to physical address translation at TLBs. Frequently used addresses can be cached in the remote memory controller, adding a few extra cycles on each remote memory access. The newly allocated remote memory can be added to the compute node's address space at run-time using the hot (un)plug service available in Linux OS. Once initialized, the new memory is available as an extension of the local address space that can be used for regular page allocation.

With the distributed approach, the compute node will have exclusive access to remote memory chunks allocated to it. This allows coherency traffic to get limited to a single

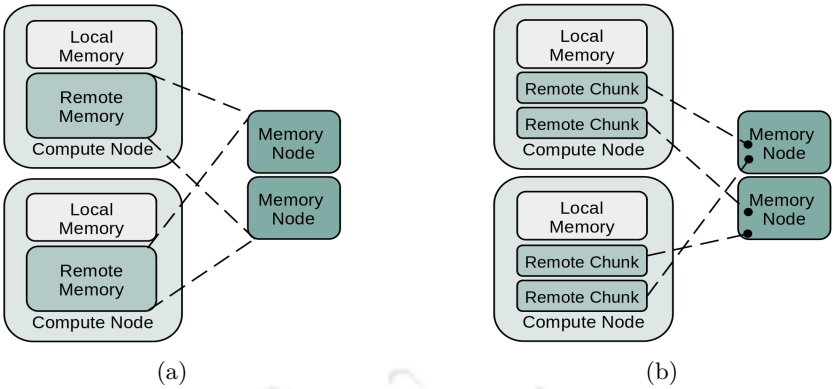


FIGURE 3.2: Remote Memory Exposure to Compute Nodes (a) Shared Organization (b) Distributed Organization

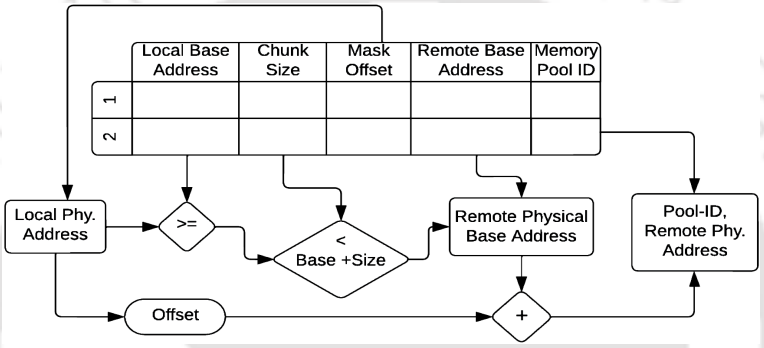


FIGURE 3.3: Address Translation at Remote Memory Controller

compute node domain and to its allocated remote memory. This eliminates the need to maintain inter-compute node coherence while enhancing scalability. Further, the major part of the compute node's cache capacity will consist of remote memory data, and the types of caches (write allocate/no-allocate, write-through/write-back) are crucial for the extent of coherence traffic between compute and memory nodes. The outward coherency traffic to memory pools can be minimized using write-back caches at compute nodes.

3.4 DRackSim Design and Operations

Fig. 3.4 gives an overview of the complete simulation process in *DRackSim* with its two-phase design: a front-end and a back-end. A Pin-based front-end performs the application analysis, whose output is fed to the back end for scalable multi-node simulation with disaggregated memory. *DRackSim* supports two different modes of operation: a trace-based simulation model and a cycle-level detailed simulation model.

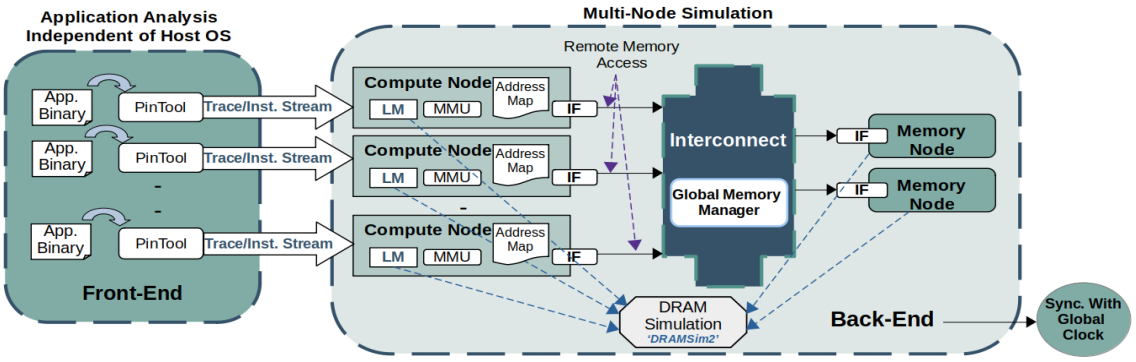
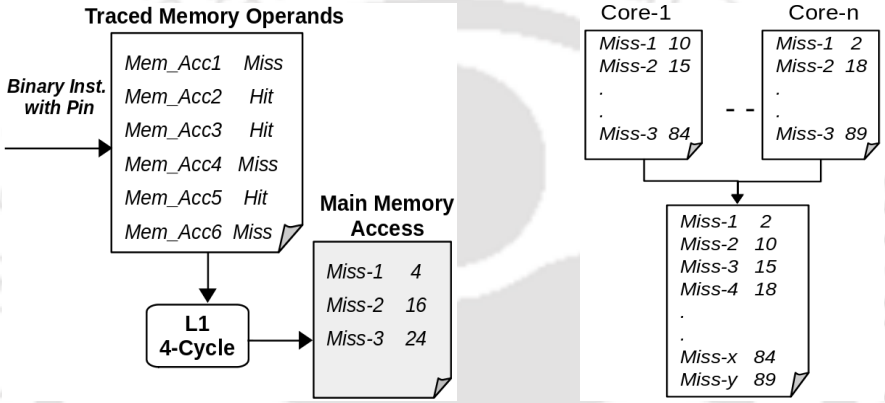


FIGURE 3.4: *DRackSim* infrastructure overview; "LM" Local memory, "MMU" Memory management unit, "IF" Interface



(a) (b)

FIGURE 3.5: Trace Generation (a) Recording Main-memory access (b) Final Multi-Threaded Trace

3.4.1 Trace-Based Model

The trace-based simulation is faster than the CPU-driven memory model and can quickly evaluate disaggregated systems on a large scale for billions of local and remote memory accesses. Pin provides with its package a tool named *Allcache* that performs a functional simulation of the single-core cache hierarchy. We extend this tool to support multi-core TLB and a 3-level cache hierarchy (private I/D-L1, L2, and shared L3) and add support to instrument multi-threaded workloads. The instrumentation is done at the instruction-level granularity, and each thread is mapped to one of the cores based on its thread ID. The tool generates memory references whenever an instrumented instruction has a memory operand. The memory references are passed through the TLB/cache model to generate an approximate cycle number for each LLC miss. With the help of access latency at each cache level and the hit/miss status of all the memory accesses, an aggregate counter is maintained to determine the clock cycle number. If the memory access is a miss at LLC, it is recorded in a trace file with the aggregate counter. Fig. 3.5a shows an example of generating main memory traces on a single cache level with

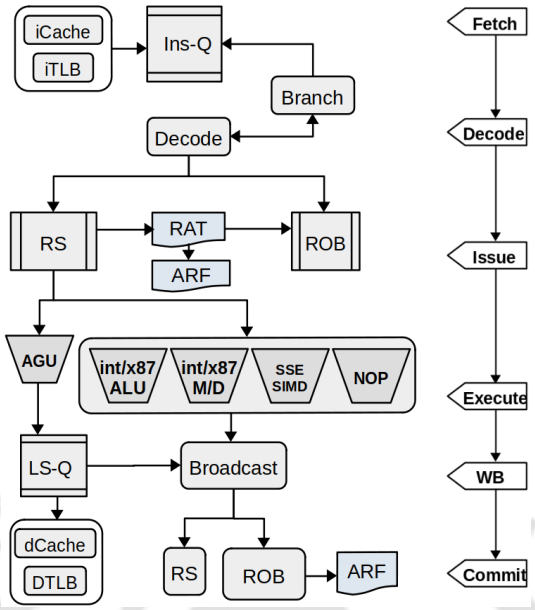


FIGURE 3.6: Out-of-Order Core Modeling Subsystem

4-cycle latency. Each memory access (hit or miss) takes 4-cycles for cache lookup, and a counter keeps track of the aggregate cycle number. All the cache misses are recorded in a trace file with the thread ID, address, access type, and the value of the aggregate counter representing the cycle number of main memory access generation. A similar process is followed to collect LLC misses at different cores, which are merged and sorted to generate a single trace file. The file represents the workload's main memory accesses while maintaining its multi-threaded nature, as shown in Fig. 3.5b. Even though main memory traces are less accurate than a real CPU model driving the memory model, they are convenient to use and consume significantly less disk space than CPU-generated memory traces. However, traces are static and do not allow system-level optimizations like hot-page migration, page-swapping systems, cache prefetching, etc., that change the state of TLBs and caches during the simulation.

3.4.2 Cycle-Level Simulation Model

Due to the above-mentioned limitations in the trace-based model, we present a cycle-level simulation model for the multi-core out-of-order x86 processor architecture at the compute node. In this mode, the pintool scope is restricted to producing an instruction stream by intercepting each executed instruction in the workload. The trace consists of instruction type (int/x87 ALU, int/x87 mul/div, SSE (vector), branch, nop, etc.), instruction address, memory operand address/size, and register dependencies for each instruction. The core and cache subsystem modeling is handled at the back end in this simulation mode.

Out-of-Order Core Modeling: Fig. 3.6 shows the details of the OOO pipeline core modeling subsystem. The core model implements multiple pipeline stages (fetch, decode, issue, execute, write-back (WB) (WB), and commit) at a higher level of abstraction with detailed modeling of hardware structures such as instruction queue (Ins-Q), reservation stations (RS), re-order buffer (ROB), architecture register file (ARF), register-alias table (RAT), and load-store queue (LS-Q), as shown in Fig. 3.6. The instructions are read from the instruction stream generated by Pin, after which a fetch unit simulates the instruction fetch for multiple instructions within a cache line rather than fetching them individually. This is also pointed out in the previous work as many other academic simulators fetch each instruction separately [108, 109]. The decode unit decodes the instruction, puts it into a buffer, and checks for the branch and its prediction result. The instruction waits in the decode buffer until the hardware resources are allocated. A limitation of performing binary instrumentation is that the execution of a process is decoupled, and it never goes down the wrong branch path in simulation. But the Pin API can tell whether an instruction is a branch. A branch predictor matches the prediction result with the information passed by the pin, and a penalty is added in case of misprediction, during which the CPU remains stalled. The issue unit allocates an entry for the instruction in the RS and ROB or stalls it if no free entry is available. The incoming instruction in the RS clears its register dependencies by accessing the ARF or from the RAT, which points to an ROB entry. The instruction waits if some previous instruction does not free a register yet. The memory read operands (if present) are sent to an address generation unit (AGU) to simulate effective address calculation and forward the address to the load-store queue for memory access. If the same load address is already present in the queue as a store in LS-Q, it is directly forwarded to the waiting instruction in the RS without adding a new load in the LS-Q. Once the dependencies are clear, instructions are moved to a ready queue. The dispatch unit selects and allocates execution units based on the instruction type and opcode. The execution latency can simply be configured for each type of instruction and its operation based on the number of cycles it takes to execute in the target processor model. Finally, the result gets broadcast among all the hardware structures: the waiting RS entries clear their memory or register dependency, instruction status changes to 'executed' in ROB, and a write-back is performed to memory if there is a write operand. Only in the commit stage is the ROB entry released, and updates to the register file are performed to make it available globally. *DRackSim* allows the user to configure all the CPU parameters (CPU width, Ins-Q, Decode buffer size, ROB size, LS-Q size, etc.) to simulate target hardware.

Cache Modeling: The cache model comprises a multi-level hierarchy with private L1 I-D, L2, and shared L3 cache. The non-blocking caches support multiple outstanding misses using miss-status handling registers (MSHRs) with a configurable number of entries. The memory access for instruction fetch or load/store queue starts at the TLB for virtual to physical address translation and uses 4KB fixed-size pages. Once the

memory access reaches LLC MSHR, it is queued for the main memory access and DRAM simulation. The caches can be configured to be either write-allocate or no-write-allocate. Finally, the cache subsystem notifies the corresponding entry in the load/store queue on completing a memory request, which is then broadcast to clear the memory dependencies of the waiting instructions in the RS.

3.4.3 Back-end Modeling

The back-end of *DRackSim* simulates an environment similar to a large-scale DMS with multiple compute nodes running simultaneously on different simulation threads. The memory accesses produced by the compute nodes drive the DRAM simulation at local or remote memory. The local memory requests are directly simulated at the node's local memory, and remote memory requests are passed through an interconnect model before being simulated for memory access at the remote memory pool. We explain here all the simulated components to model disaggregated memory behavior.

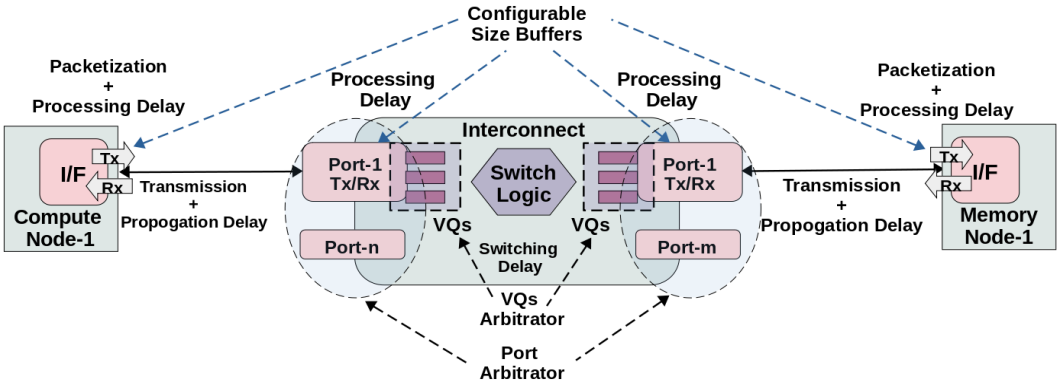
Compute Nodes: Besides CPU and cache simulation, compute nodes simulate a local memory unit and a memory manager to make memory allocation decisions and manage address space. The memory manager is an abstraction of processor MMU for address translation and an OS memory manager for address space management and memory allocation at the compute node. A memory request reaches MMU on a TLB miss and performs a page-table walk with a defined latency. If the page is not in memory, the request is forwarded to the page-fault handler for memory allocation and creates a page-table entry (PTE). *DRackSim* models 4-level page tables for mapping virtual addresses to the physical pages. The memory manager allocates a new page in local or remote memory based on the allocation policy and availability of free memory space. The page-fault service stalls the CPU and incurs fixed latency, which can be configured to model the OS page-fault latency. The page allocation in disaggregated memory is crucial, and the memory manager should carefully decide the footprint ratio in local and remote memory for different applications running at the compute node. The response time of latency-sensitive applications can be significantly impacted compared to latency-insensitivity applications. An uninformed page allocation policy that does not consider the sensitivities of different applications while allocating the memory pages in local/remote memory can result in high average memory latency, impacting application performance. The memory allocation at the compute node in *DRackSim* can be configured to allocate memory pages in any ratio from local and remote memory. The modeling of these components allows for exploring memory management policies that can make such decisions.

Global Memory Manager: The global memory manager (and controller) takes care of the remote memory address space in all the memory pools and reserves remote memory

from one of the pools on receiving a request from compute nodes. The global manager handles conflicts during remote memory reservations to different nodes and acts as a load balancer while choosing a memory pool for allocation. The remote memory is allocated in large granularity chunks rather than single pages (possible in distributed memory organization) and hence does not introduce allocation bottlenecks. The remote memory allocation requests are initiated by the compute nodes' page faults and then resolved by the global memory manager at the switch by allocating remote memory chunks. Therefore, the remote page fault handling latency latency is configured larger than the local page faults. The maximum number of simultaneous requests is defined by the MSHR at the dTLB. On reservation of a memory chunk, it will share chunk details (pool-id, remote base address,s size, etc.) with the requesting compute node, creating an entry in its mapping table as shown in section 3.3. The memory pools are bound to face contention in their queues when several compute nodes access the same pool simultaneously. To avoid contention and tail latency, memory pool selection should be done so that all pools face almost similar amounts of memory request traffic. It should also ensure QoS to different applications running on compute nodes with different sensitivities towards memory latency (compute vs memory-bound applications) using request scheduling mechanisms. *DRackSim* follows a round-robin pool selection while reserving a memory chunk. It allows further exploration of similar pool selection policies for memory performance evaluation impacted by contention when memory requests (lead by allocation) are unequally distributed among pools.

Interconnect Model: The interconnect model in *DRackSim* is based on a queue simulation that simulates the behavior of memory-semantic fabrics such as CXL or other similar interconnects proposed for disaggregated memory. The interface (similar to NIC) at the compute nodes (memory requestor) and memory pools (memory responder) allows access to remote memory through the network. The on-chip integration of the fabric and lightweight network protocol implementation in the hardware allows low-latency cache line access from remote memory during an LLC miss. The remote memory accesses from multiple compute nodes pass through a central switch before accessing the pooled memory. The bandwidth and latency can be configured to match the interconnect.

DRackSim simulates both cache line and paged access from the compute nodes to remote memory pools for design space exploration. If an LLC miss refers to remote memory address space, it accesses the local-to-remote address map (discussed above) at the compute node's remote memory controller. The memory access is encapsulated into a network packet containing the destination memory pool-id and its remote physical address, as shown in Fig. 3.7b. The model uses fixed 64-byte packets for a memory request, as the payload consists of only a memory address. The packet is then pushed into the queue structure at the controller's network interface after adding a delay for packetization. While the packet transmits from the compute node to the switch, it incurs



(a)

```
struct remote_memory_access
{
    uint64_t remote_phy_addr;
    int source;//compute node
    int dest; //Memory Pool ID
    uint64_t trans_id;
    bool isWrite_block_req;
    bool isPAGE;
};
```

(b)

FIGURE 3.7: (a) Interconnect Simulation Model detail (b) Packet Structure for Remote Memory Access

transmission and propagation delays based on configured bandwidths at the nodes and hop distance. It is then added to the switch input port queue and faces a processing delay. The interconnect model implements a crossbar topology that supports single-hop communication between compute and memory nodes. It supports virtual queues at switch ports to avoid head-of-line blocking, and a 2-stage switch arbitrator selects the packet for forwarding in each cycle. The first stage arbitrator selects one of the input ports, and the second stage selects from one of the virtual queues at the selected input port. The packet is added to the buffer at the destination output port after adding a switching delay. Finally, it reaches the network interface of the destination memory pool for DRAM simulation. A similar way is followed at the memory pool to send back the response to a compute node using the source address of the memory access packet. The response packet holds a cache line of data as a payload for block accesses, and its size can be configured accordingly based on cache block size. Similarly, write-backs from compute nodes to remote memory also use a packet size capable of storing a cache line.

Further, page requests from compute nodes to remote memory are also simulated. Memory pools can differentiate between block or page request packets through the additional information present in the header. The response from the memory pool can be sent as single or multiple small-sized packets that the user can configure. The reassembly logic collects all the response packets at the compute node to form a memory page and notify

on receiving a complete page. This functionality can be used to implement hot-page migration systems to reduce average memory latency. However, poorly scheduled page access can starve the critical block accesses and should only be used to supplement cache line accesses for utilizing locality in hot pages. Further, the network interface and the ports at the switch support configurable size buffers at both ends and implement a reliable network with back-pressure flow control in case a buffer gets full. The interconnect model in *DRackSim* can be configured with different latency and bandwidth to simulate target hardware (CXL1.0/3.0) for cache line and page transfer. Fig. 3.7a shows a complete view of the interconnect simulation model in *DRackSim*.

Some memory systems might have an optimal memory fetch size bigger than a cache block but smaller than a complete page. For those cases, *DRackSim* allows configuring the memory fetch size by specifying the address range within a page. However, we leave it to future work to explore such optimizations and implement new hardware structures (e.g., bulk prefetching into buffers). The baseline system without extra hardware only fetches a single cache line from memory pools on an LLC miss.

Memory Simulation: We integrate cycle-accurate DRAMSim2 [] for DDR4 simulation of local and remote memory. We initialize multiple instances of DRAMSim2 memory units using its *Memory System* interface, each representing either the local memory at a compute node or remote memory at a memory pool. DRAMSim2 provides a *callback* functionality to notify the CPU driving the memory model on completing every memory access. We modify the *MemorySystem* interface and *callback* functionality so that each memory unit (at a node or remote pool) can have a separate identity. We further modify the *addtransaction* function, which adds a memory request to a memory unit, to include a node-id of the requester, a transaction-id, and other metadata for the collection of statistics. The modifications allow tracking the completion of memory accesses at each memory unit separately and correctly sending back a response to the requesting node.

Simulator Implementation and Clock Management: The back end of *DRackSim* handles the scalable disaggregated memory simulations of multiple compute and memory nodes. In the trace-based model, multiple traces are collected (one for each node) to be parsed in parallel to perform memory and network simulation representing a multi-node environment. The memory requests are split across local/remote memory and in-between different memory pools based on their access address and the location of respective pages. In the cycle-level simulation, multiple instruction streams are generated simultaneously by separate instances of Pintool (one for each node). All the instruction streams are then continuously fed to the back end in parallel for multi-node disaggregated memory simulation. We also add support for instrumenting multiple applications to create workload mixes at a single compute node. The user can skip any number of instructions to jump to the region of interest in a workload.

At the back-end, *DRackSim* model consists of multiple independent components such as compute nodes, memory pools, interconnect, etc., which are synchronized with a single global clock. This is necessary to maintain the time-ordering of global events, such as simultaneous network and remote memory accesses from different compute nodes. The number of compute nodes, memory pools, and the frequencies of individual components (compute/memory nodes and interconnect) can be configured separately, and the global clock only provides a common reference time for the functional correctness of the simulation. The simulation of each compute node is performed using a separate thread, and *thread-barriers* are utilized for synchronization and controlling the simulation flow of multiple nodes. This allows simulation of large-scale DMS without much slowdown.

3.5 Validation

It is important to cover different validation aspects while developing a new architectural simulator. The first one is the functional correctness of the program. In our case, application functionality is decoupled from the actual simulation process, where Pin runs it natively. Although Pin can change application functionality and data flow (e.g., by changing the register values), our Pintool is restricted to adding only instrumentation primitives to the source binary for generating program traces. We verify the functional correctness of the Splash-3 benchmark suite, which DRackSim correctly maintains. The second aspect is the accuracy of performance metrics. While validating the simulator with actual hardware is important, the availability of a full-scale disaggregated system is an issue. The integrated on-chip interconnects, such as CXL, are not yet commercially unavailable, and similar interconnects have only been tested with small-scale prototypes using FPGAs [13, 15, 110]. The next generations of Intel Xeon [29] and AMD EPYC [30] processors are expected to come with CXL v1.1 support, but the functionality of CXL memory controllers and remote memory managers is not clear yet. Further, CXL v1.1 does not support a switch for connecting multiple CXL devices (memory pooling). On the other hand, the performance of RDMA-based memory-sharing servers significantly rely on the software implementation and usage of RDMA constructs in the application. Moreover, RDMA does not support cache coherence, which makes it a non-candidate for validation.

Due to the unavailability of hardware, we choose to validate *DRackSim* with a standard open-source simulator and incrementally validate the individual components. As a best-effort approach, we validate the core and cache subsystems of DRackSim against *Gem5* system emulation (SE) mode. We validate the network interconnect against the closest hardware prototype and also show the performance impact with a wide set of experiments in the next section. We set the processor width and instructions latency for

TABLE 3.1: Validation Parameters

Element	Parameter
CPU	3.6GHz, 8-width, 64-InsQ, 64-ResvStation, 192-ROB, 128-LSQ
L1 Cache	32KB(I/D), 8-Way, 2-Cyc, 64B block
L2 Cache	256KB, 4-Way, 20-Cyc, 64B block
L3 Cache	2MB per core shared, 16-Way, 32-Cyc, 64B block
Cache Type	Write-Back/Write-Allocate, Round-Robin

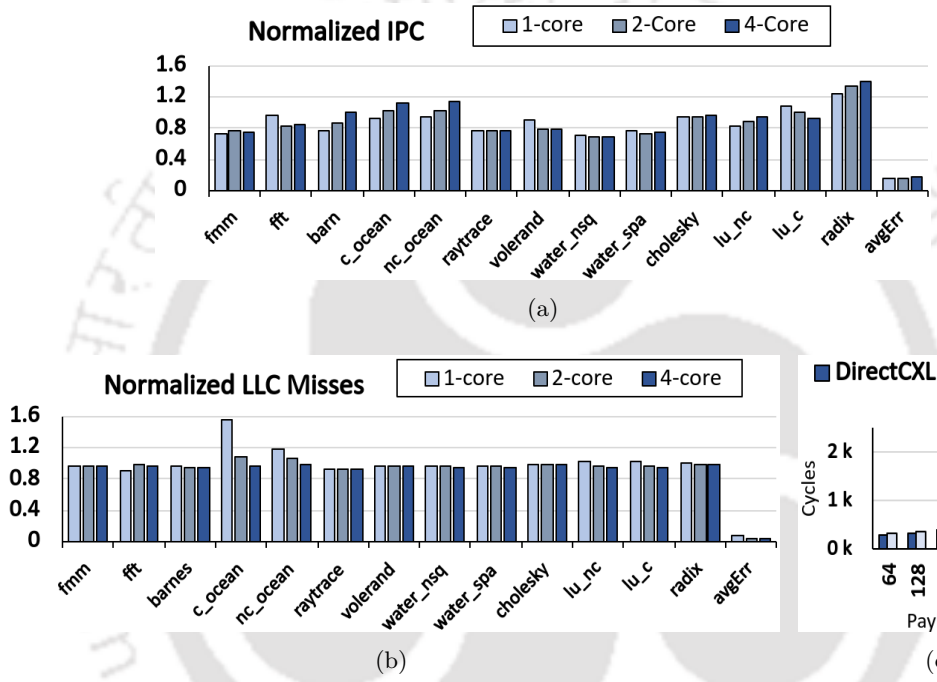


FIGURE 3.8: Validation on *Splash-3* benchmarks (a) Normalized IPC (b) Normalized LLC Misses (c) *DRackSim* vs DirectCXL

each instruction type to the same values for calibration and used the same size structures for all the hardware resources (such as InsQ, RS, ROB, and LSQ) in *Gem5* and *DRacKSim*. We further fix our simulator's page fault and TLB-miss latency as per *Gem5*, which only adds 1 or 2 cycles on each such event in SE mode. Table 3.1 shows the system configuration for CPU validation.

We perform the CPU validation for 1, 2, and 4 cores over Splash-3[111] benchmarks by spanning the same number of threads as the number of cores. Figure 3.8a shows the CPU validation results with normalized IPC values of our simulator compared to the IPC values of Gem5. The IPC numbers of *DRackSim* are close to Gem5 IPC for most of the benchmarks and show a mean absolute percentage error of 12% across all core configurations. Similarly, we validated the cache subsystem using the LLC misses, which can represent the behavior of the complete cache hierarchy and is a standard approach

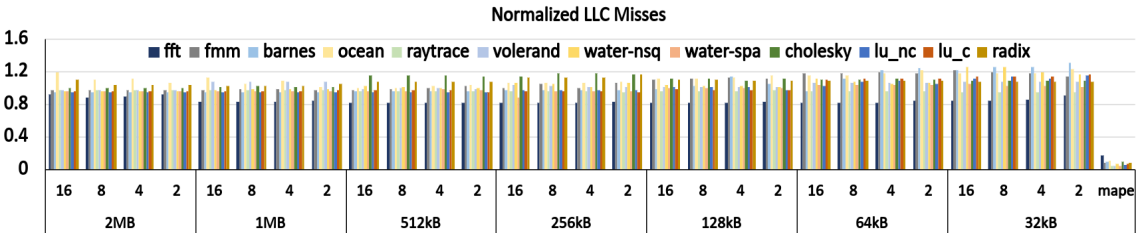


FIGURE 3.9: Last Level Cache Misses normalized against Gem5 over different cache configurations

for cache validation [108]. LLC is an interaction point between the multi-level caches and memory hierarchy and is a suitable candidate for overall cache validation by measuring the number of misses or the miss rate at LLC. Figure 3.8b shows the LLC validation results for all workloads. We only observe an unexpectedly large error in the case of *contiguous_ocean* with a 1-core CPU. Besides this, the mean absolute percentage error is around 3% across all benchmarks on 1, 2, or 4 cores CPU. Finally, we validate *DRackSim* for raw memory latency against that of a Direct-CXL prototype [85]. We simulate the same cache latency (L1:4, L24), LLC MSHR (16), CPU frequency (100MHz), and CXL switch frequency (250MHz) as mentioned in [78]. The micro-benchmark only consists of instructions that fetch a small amount of data (64B to 4KB) from CXL-connected DDR4. Fig. 3.8c compares raw memory latency for different payload sizes in *DRackSim* against Direct-CXL.

We further perform an in-depth validation for the LLC misses over a range of L3 configurations on a 4-core CPU, shown in figure 3.9. We reduced the L2 size to 64KB to maximize the number of cache misses at LLC. Here also, we observe a slight variation in the LLC misses for DRackSim compared to Gem5. We observe a mean absolute percentage error of 7.5% for all workloads aggregated over all configurations. The LLC misses are slightly inflated or deflated in some configurations, which is due to the difference in implementation details of the cache hierarchy between *DRackSim* and *Gem5*. We do not implement a separate write buffer, so the LLC must evict a block during the write-backs. Another reason can be using separate load and store queues in Gem5, whereas DRackSim has a unified load/store queue that can create a small difference in the total number of non-redundant loads and stores. These differences can generate variation in cache accesses, and inaccuracies can accumulate from lower-level caches to LLC.

The variations observed in the results are common among different simulators, as shown for validation efforts in the past [108, 112]. Ayaz et al. surveyed major architectural simulators (Gem5, MARSSx86, Sniper, etc.) and observed significant variations in their IPC values and LLC misses compared to real x86 hardware. The main source of inaccuracies is due to the lack of support for fused micro-operations (μops), the pipeline depth, the lack of modeling for all hardware optimizations, and the abstraction of actions performed during branch miss-prediction. Further, Gem5 uses separate load/store

TABLE 3.2: Benchmarks

Benchmark name	**Domain**	**Input**	**LLC Misses**
Stream Cluster (SC) [113]	Data Mining	Pts:65536 Dim:256	3.67
Needleman Wunsch (NW) [113]	Bio-inform.	Rows/Col:4096 Pen:4	6.68
Block Tri-diagonal (BT) [114]	CFD	Class C	2.97
3D Fast Fourier (FT) [114]	CFD	Class C	28.63
High Perf. Conj. (HPCG) [115]	HPC	104 × 104 × 104	2.71
K-core Decomp. (KD) [116]	Graph Proc.	V:1M E:10M	0.95
K-means Clustering (KC) [113]	Data Mining	kdd_cup	0.62
Lulesh (LU) [117]	HPC	Cube Mesh Size:120	5.29
Multi-Grid (MG) [114]	CFD	Class C	37.95
miniFE (FE) [118]	HPC	140 × 140 ×140	14.89
PageRank (PR) [116]	Graph Proc.	V:1M E:10M	0.95
Particle Filter (PF) [113]	HPC	2K × 2K 20K Particles	33.13
Pennant (PEN) [119]	HPC	leblancx4.pnt	8.02
SimpleMOC (SM) [120]	HPC	small	1.75
SRAD (SR) [113]	Image Proc.	4K × 4K Data Points	3.19
XSBench (XSB) [121]	HPC	small	8.32

queues, whereas DRackSim has a unified queue that can create a small difference in the number of non-redundant loads/stores. These small inaccuracies can accumulate from lower-level caches to LLCs.

3.6 Evaluation

In this section, we demonstrate the working of *DRackSim* over various configurations and provide use case evaluation by modeling different data movement schemes in DMS. We also show the impact of the network by changing the latency and bandwidth parameters at the node's interface and the switch.

We evaluate the performance using various multi-threaded benchmarks shown in Table 3.2. We chose five workloads from the Rodinia heterogeneous benchmark suite: SC, NW, KC, PF, and SR, that simulate applications from different domains such as data mining, bio-informatics, image processing, etc. Two graph processing workloads, KD and PR, were chosen from the Liagra framework. Three workloads, BT, FT, and MG, are selected from the NASA parallel benchmark suite that mimics the computation and data movement in computational fluid dynamics applications. Further, we use six more multi-threaded workloads and mini-apps from the domain of HPC: HPCG, LU, FE, PEN,

Table 3.3: Simulation Parameters

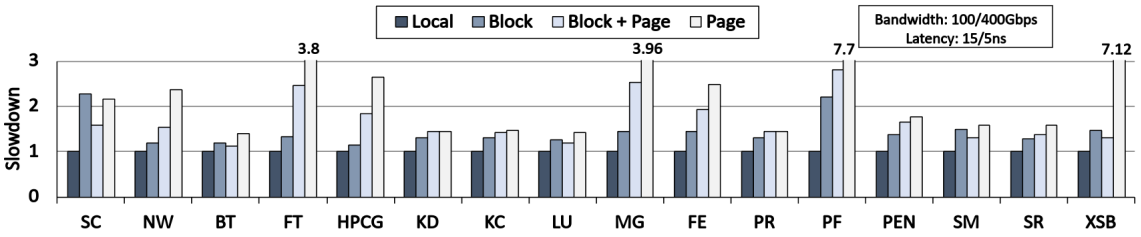
Element	**Parameter**
Page Fault	9µs
Global Memory Manager	15µs (Memory Chunk Allocation)
Memory (Loc/Rem)	2400MHz DRAM (19.2GB/s)
Switch	100/400Gbps, 4MB port Buffer, 5/10ns for Proc/Switching
Network Interface	40/100Gbps, 1MB buffer, 15/30ns for (De)Packetization/Proc
Packet Size	64B Req, 128B Resp, 128B Write-Backs, 4KB for Page Access

SM, and XSB. The selected workloads have a wide range of memory footprints, ranging from 38MB to 3.2GB, and vary in memory access patterns. We run openMP versions of all the workloads and only simulate the multi-threaded regions of the workloads (except PF, which has a significantly large single-threaded phase) using 4-threads, with each thread spanning one of the cores (no hyper-threading). The simulations were run for at least 400 million instructions for each multi-threaded workload and 100 million for PF, which was slow and took a long time. The last column of the table shows the total number of LLC misses or main memory accesses (in millions) for all workloads during the simulation period.

Finally, Table 3.3 shows the network and memory parameters for the simulation, while we use the same CPU and cache parameters mentioned earlier in section 3.5. We use a page fault handling and remote chunk allocation latency of 9µs and 16µs, respectively. We assume a 64B data packet for the remote memory request packet from the compute node. On the other hand, the response packet from the memory pool and remote memory write-backs will contain a cache block of data and have a packet size of 128B. The remote page request is also 64B, whereas the page access is performed by adding 64 cache block requests to the memory unit, and the response is sent as a single 4KB packet. For interconnect, we consider 100/400 Gbps bandwidth for the switch and 50/100 Gbps bandwidth at node interfaces. We also vary the latency, which is 15 or 30 ns at nodes for (de)packetization and processing and 5 or 10 ns for processing and switching at the switch. Each remote memory access will pass four times through the node interfaces (request/response at compute node and memory pool) and two times through the switch (request/response), bringing the total latency factor to 70 or 140 ns.

3.6.1 Design Space Exploration

To understand the impact of memory disaggregation with traditional server systems, we modeled four use case scenarios with *DRackSim*:



① **Block:** The first is our baseline hardware DMS in which all the remote memory accesses are made at the cache block granularity on an LLC miss. The pages are allocated alternatively in local and remote memory, each having 50% of memory footprint.

② **Page:** Next, we model a software disaggregated memory where the remote memory accesses are always made at page granularity. The page allocation is performed in the same manner with 50% of the workload footprint at remote memory and is migrated to local memory on the first reference to it.

③ **Block+Page:** Next, we model a very simplistic page migration on top of the baseline hardware disaggregated system that only migrates some of the hot pages in remote memory to local and utilize locality for future accesses. The memory requests to other remote pages are still made at cache block granularity. The page migration threshold is set through a simple training phase and is fixed at 20% access count of total accesses to predicted hot pages during training. Further, no special support is added for scheduling page migrations.

④ **Local:** Finally, we model a local-only system that assumes big enough local memory to fit the entire workload footprint. This system will have no remote page allocation, and all memory accesses are performed in local memory at block granularity.

The systems described in ② and ③ require an update to system page tables on every remote page migration and invalidate TLB entries on all the cores. This introduces long CPU stalls and may further take around $4\mu s - 13\mu s$ (depending on the core count) [22] for TLB-shootdown after every migration. Therefore, for a fair comparison, we perform page table updates in large batches of 1024 pages and use a remap table to delay page migrations by temporarily storing the new physical address of a migrated page.

We start our evaluation with a single compute node with one memory pool to show the performance impact of all three configurations compared to local. With a single compute node, all the network and remote memory bandwidth is available only for that node. In Fig. 3.10, we show the performance slowdown and increase in memory access cost for all the workloads at bandwidth and latency combination of 100/400 Gbps and 15/5 ns, respectively. As expected, we saw a significant performance slowdown that ranges

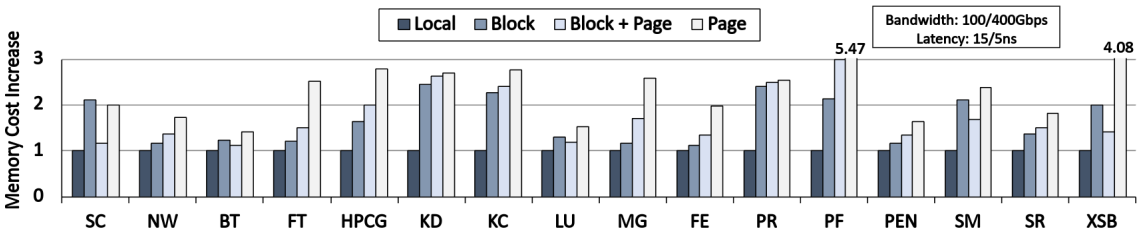


FIGURE 3.10: Impact on system performance (top) and memory cost (bottom) on all the workloads over all four configurations

from 10% to 120% for **Block** compared to **Local**. The variation in the slowdown for different workloads is due to changing memory access patterns, memory sensitivity, and the number of memory accesses. SC and PF are more sensitive to increased memory latency and face maximum slowdown. HPCG, KD, KC, and PR are less sensitive to memory latency and observe slowdowns between 5%-18% even after facing a significant increase in memory access cost. Although most workloads have similar average memory latency with **Block**, some show more variation when compared to **Local**. For instance, NW, BT, FT, MG, and FE face the least increase in memory cost, as these workloads already have large memory latency with **Local** (ranging between 70-100 ns) due to a streaming access pattern as the memory bandwidth is limited. With disaggregated memory, the requests are first distributed across two separate memory units (local and remote), thereby eliminating some contention due to more bandwidth. However, FT, MG, and FE still face a 25% to 30% performance slowdown due to a significantly high memory access count, unlike NW and BT, which have fewer memory accesses and did not face much impact from having 50% of memory footprint in remote memory. MG and PF have large memory accesses, but PF being single-threaded, could not use as much memory parallelism to hide memory latency and face a 2x slowdown.

With **Block+Page**, there was an expectation to observe a significant performance improvement compared to baseline **Block**. However, migrating remote pages to local memory shows no significant benefits. Only five out of all the evaluated workloads: SC, BT, LU, SM, and XSB, could get any benefits, if at all, and face 5% to 50% lesser slowdown than baseline. On the other hand, **Page** faces the maximum slowdowns compared to all the other scenarios. Firstly, the trend clearly explains the reason behind the performance slowdown with **Page**, which only accesses remote memory at page granularity and migrates a complete remote page on every LLC miss belonging to a remote address. Remote page accesses are costly as they read multiple cache blocks within a memory page (64 for 4KB page in our case) and add significant transmission delays at the network due to the large packet size (4KB).

In the case of **Block+Page**, only a few hot pages are migrated to local memory and do not see the same performance drop as in *Page*. Most of the memory accesses are still performed at block granularity. We observed that SC, BT, and SM all have small

memory footprints and fewer memory accesses, so it has only a few pages to migrate. It does not add severe delay to the critical block memory accesses to remote memory while accessing a page. It also allows completing a good percentage of memory accesses in local memory, utilizing the locality in hot pages. For LU and XSB, even though they migrate more pages, more than 90% of the memory accesses are completed in local memory as the result of migration and hence observe better performance. KD, KC, and PR have a few memory accesses and face a small slowdown due to remote page accesses compared to **Block**.

FT, MG, FE, and PF have enormous memory access traffic and migrate many pages due to their large memory footprints. The performance only gets impacted due to extra delays added by page accesses to regular demand remote memory accesses, which are in their critical path. Similar is the case for HPCG, which is more sensitive to page access delays, despite a small increase in average memory latency. NW and SR also could not benefit from migrating hot pages. Overall, the impact is lesser than **Page** for all workloads, as the regular block accesses are also performed simultaneously in remote memory. Further, KC, MG, PF, PEN, and SR also show less amount of spatial locality and could only get 20%, 15%, 25%, 23%, and 18% additional memory accesses in local memory even after migrating 66%, 65%, 48%, 38%, and 62% remote pages to local memory, respectively, out of the 50% of total pages that are placed in remote.

Page migration has also been widely used in DRAM-NVM hybrid memory systems; however, they do not present the same challenges as in DMS. A larger memory footprint on remote memory and the presence of an interconnection network make hot page migration more challenging in disaggregated environments to realize performance gains. We observed a few things that may help improve its performance with page migration. However, we leave it to future research work to explore optimized designs. The continuous access of a complete page for 64 blocks is problematic for two reasons. One is that the pending block memory requests to that page face long delays and are only served once the migration is complete. Two, the block memory requests to other pages in the same remote memory pool may get starved if multiple hot page requests are issued together. Further, optimal scheduling of page migrations becomes more critical due to the same reasons. An optimized hot page migration must consider all these issues for a viable solution.

In Daemon [37], authors explored bandwidth partitioning, link compression, and schedules page access only when the utilization of the block request queue is less. However, page access is still completed as a whole, adding a delay of around 1.2-1.5µs to subsequent block accesses even with an optimal scheduling policy. With truly disaggregated hardware and CXL interconnects, it is better to break down the page accesses into multiple cache line requests that can reduce most of the overhead with extra architectural support.

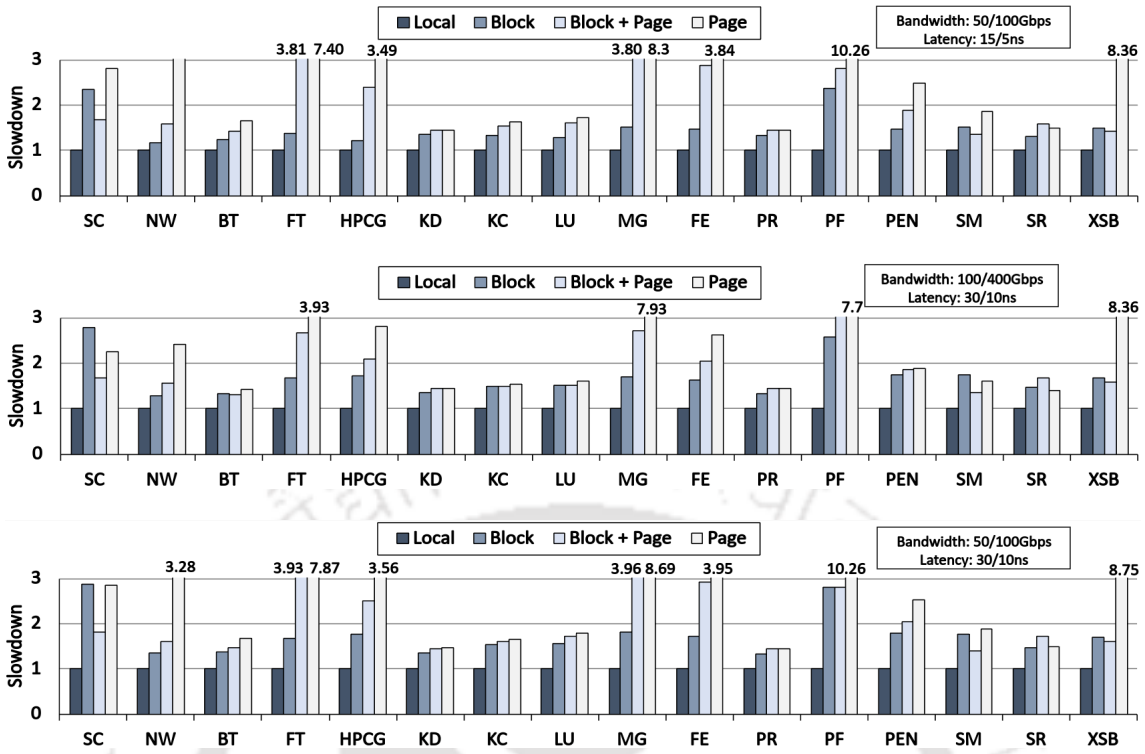
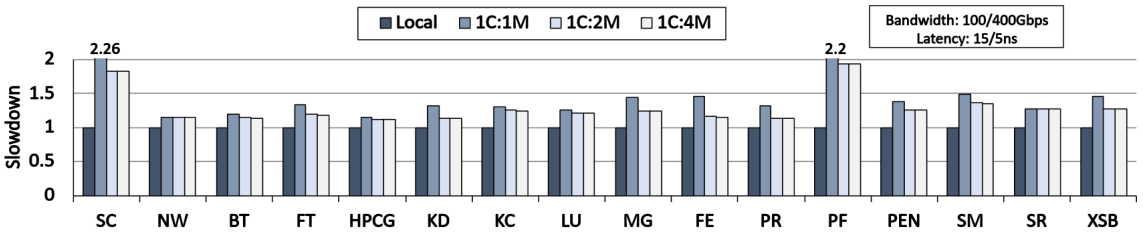


FIGURE 3.11: Impact on system performance for all workloads on changing the network latency and bandwidth parameters

Next, we show the performance impact over different network configurations described earlier. We reduced the node's interface bandwidth by 1/2 to 50Gbps and the switch bandwidth by 1/4 to 100Gbps. However, we kept the same latency at every point. The first graph in Fig. 3.11 shows that **Block+Page** and **Page** are significantly impacted by reducing the bandwidth. As both systems access remote memory pages with a response packet size of 4KB, therefore face additional transmission delays of around 575ns per page during a response by the memory pool. However, workloads show different sensitivity to the increased page access time. On the other hand, the performance of **Block** is more or less the same as in the previous network configuration, as it only accesses remote memory at 64B block granularity. Although each memory access takes a few nano-seconds more, the continuous memory accesses hide some of these latencies and have no significant drop in performance.

Next, we increase the latency parameter by 2x and use the same bandwidth of 100/400Gbps. The trend is the opposite in this case, as the packets with small sizes (block access) are most impacted and face almost two times remote memory latency for each access compared to initial network parameters. In contrast, page response packets get a slight increase in latency compared to their overall access time. **Page** is least impacted by increasing the latency, as each page access adds only 70ns extra for the request and response combined. The performance of **Block+Page** is also comparatively better in those cases where the number of remote memory accesses was significantly reduced due



to page migration. However, it still has to send many block accesses to remote memory, which increases the wait times. In the last case, we decrease both the latency and bandwidth, for which the performance can be seen in the bottom graph in Fig. 3.11. Overall, in all these cases, BT, KD, KC, LU, PR, and SR are the least impacted by memory disaggregation.

3.6.2 Multi-Node Disaggregated Memory Systems

As multiple memory pools are expected to be grouped with several compute nodes in a practical disaggregated environment, scalability remains a crucial aspect of DMS. Hence, we evaluate the performance impact of disaggregation in different compute node to memory pool configurations (xC:yM). We kept the memory footprint at a same ratio of 50:50 between local and remote in all the scalable configurations. The remote chunk allocation is done through round-robin pool selection whenever more than one memory pools are there.

Sensitivity to Multiple Memory Pools: Fig. 3.12 (Top) shows the performance improvement of each workload when a single node is configured with multiple memory pools. The performance is shown for the baseline configuration **Block**. With 2-memory pools, we observe a maximum performance improvement of around 25% for SC and FE, while it varies from 2% to 15% for most of the workloads and none for NW and SR. The improvement can relate to the increased memory bandwidth with more memory pools, as the remote memory accesses are now distributed across multiple memory units. Thus reducing the chances of contention in the queue and tail latency compared to a single memory pool. However, a further increase in memory pools from 2 to 4 has no additional benefits, as two memory pools could fulfill the maximum bandwidth requirements for all the workloads.

Sensitivity to Multiple Compute Nodes: In Fig. 3.12 (Bottom), we show the impact of increasing the number of nodes running the same workload with a single memory pool. SC, being more memory sensitive, is significantly impacted in all configurations with more than one compute node. FT, MG, FE, and PF, due to their extensive memory requests, observe high contention in the memory queues. However, with a 100Gbps interface and 400Gbps network bandwidth, and large enough buffers, interconnect could

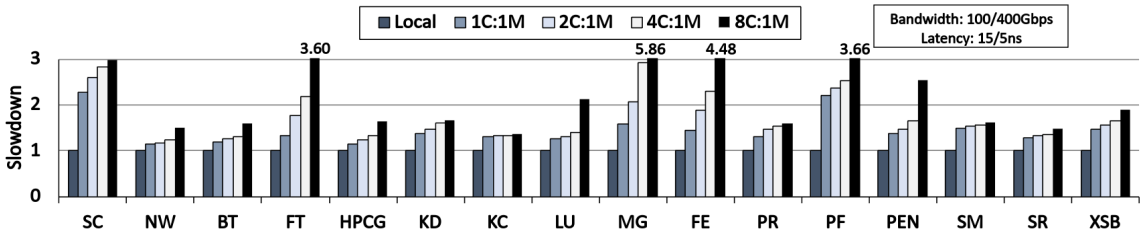


FIGURE 3.12: Impact on system performance for all the workloads on increasing the number of memory pools with a single compute pool (Top) or by increasing the number of compute nodes with a single memory pool (Bottom)

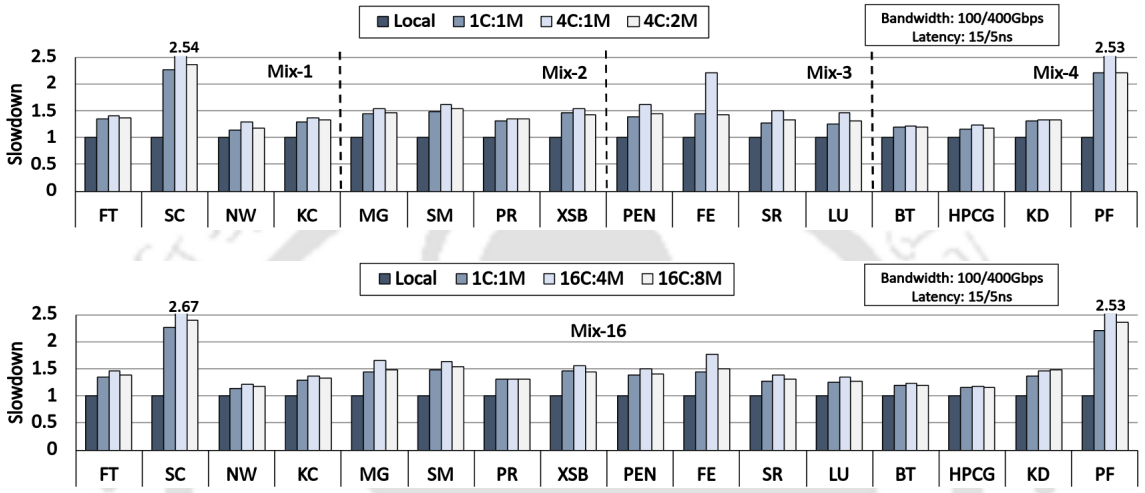


FIGURE 3.13: Impact on system performance with different compute-to-memory node configurations and workload combinations over 4 Compute Nodes (Top) and 16 Compute Nodes (Bottom)

still manage the high network traffic without much extra overhead. Finally, LU and PEN observed a server slowdown when the nodes were increased from 4 to 8.

Sensitivity to Multiple Compute and Memory Nodes: Next, we increase the number of compute nodes and memory pools to evaluate the performance for expected configurations in a practical environment. Fig. 3.13 (Top) shows the performance impact when four nodes run simultaneously over 1 or 2 memory pools. The performance is shown for four different workload mixes, with one workload running at each node. The workload mixes were created so the nodes have enough variation in their memory access rates. As expected, we observe a slowdown of 2% (PR, KC, KD) to 35% (FE) when 4-compute nodes are configured with only one memory node (4C:1M) compared to a single compute and memory node (1C:1M). Subsequently, this slowdown is 10% to 125% compared to a system using only local memory. However, the performance difference is negligible for 4C:2M compared to 1C:4M, making the case strong for compute-to-memory ratio 2:1 (under the given memory bandwidth parameters). We even observe a performance improvement for XSB in 4C:2M compared to 1C:1M, as the distribution of memory accesses suits its access patterns.

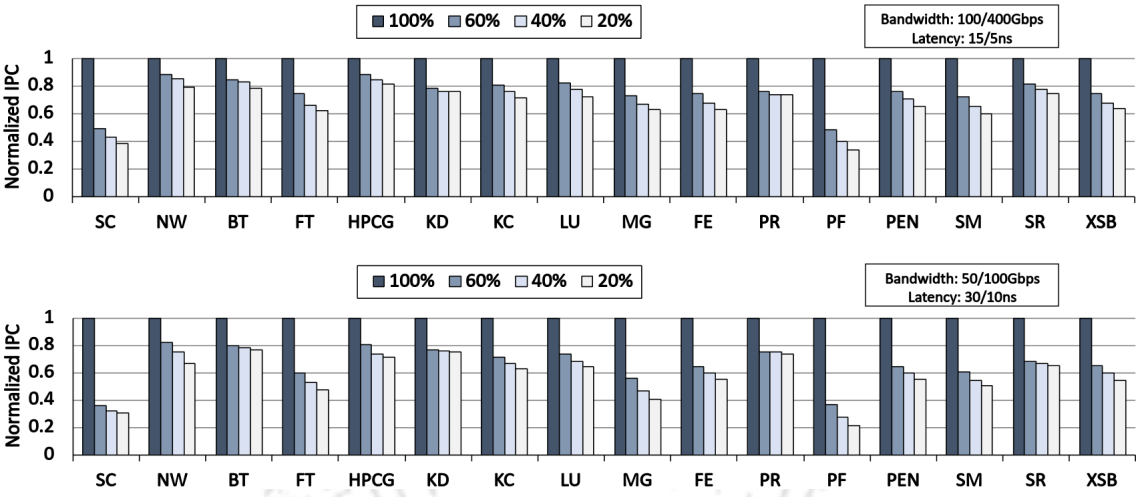
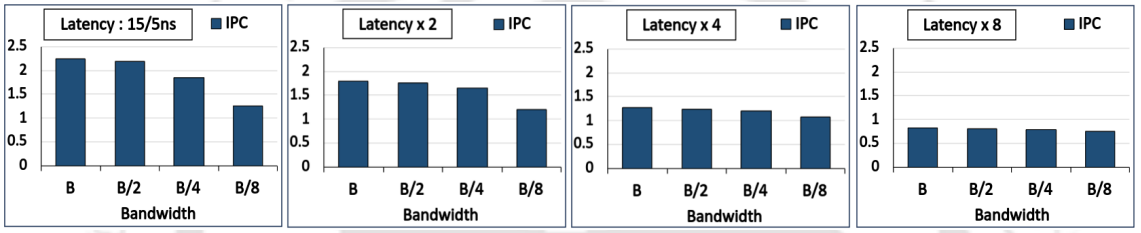


FIGURE 3.14: Impact on system performance on changing the local memory footprint



Similarly, we run all 16 workloads with an even bigger configuration by deploying them on 16 compute nodes (one workload on each node) with 4 or 8 memory nodes. Fig. 3.13 (bottom) shows the performance impact for all the workloads in these configurations. We observed that the performance is more or less the same compared to 4 node configurations over similar compute-to-memory node ratios (2:1), with a slight difference for SC and FE.

3.6.3 Sensitivity to Local Memory Footprint

In a practical setting, the page allocation ratio in local and remote will be crucial in deciding the impact of disaggregation, which varies for different applications, as seen in the above experiments. Therefore, we perform a sensitivity analysis on all the workloads with the changing local-to-remote memory footprint ratio. Fig. 3.14 shows the system IPC at two different network configurations with 60%, 40%, and 20% local memory footprint normalized against 100% local memory. The findings motivate the requirement of such algorithms, which can decide the local/remote ratio for applications running on the compute node. Further, multiple workloads will compete for local memory and provide the opportunity to explore page placement mechanisms.

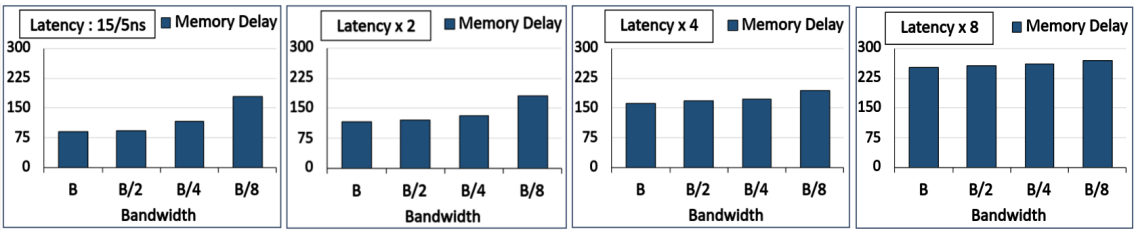


FIGURE 3.15: IPC (Top) and Average memory access delay (Bottom) for STREAM benchmark on changing the network bandwidth and latency

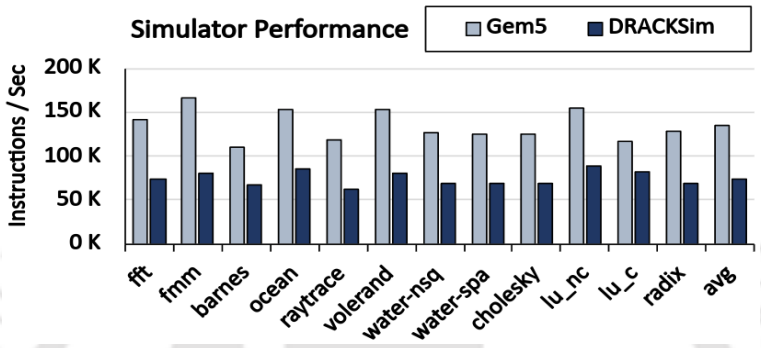


FIGURE 3.16: Simulator Performance Compared to Gem5

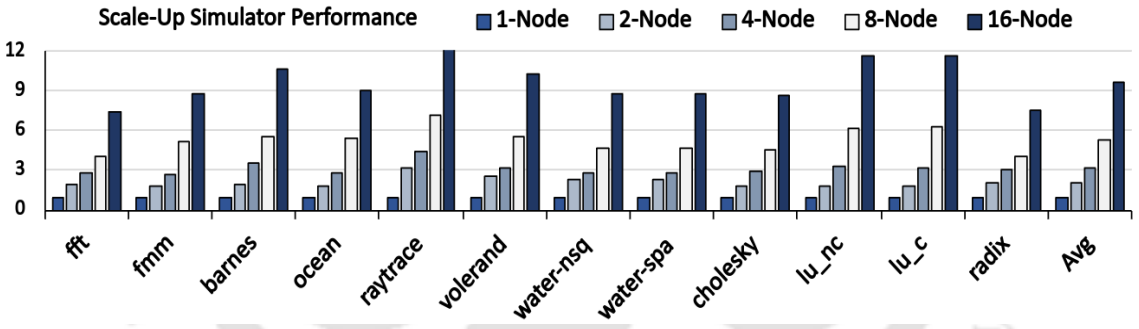


FIGURE 3.17: Increase in simulation time *DRackSim* for large-scale simulations for one million instructions per node (Normalized against the performance of a single node)

3.6.4 Network Latency and Bandwidth Test

Next, we rigorously test the interconnect model for its correctness using the STREAM benchmark with an input stream array size of 50 million. The simulation is also run for 400 million instructions on the multi-threaded region and has a memory footprint of around 1.14GB. We start with the interface and switch bandwidth of 100 and 400 Gbps, respectively, and reduce until up to one-eighth. Similarly, the initial latency is assumed as 15ns (nodes) and 5ns (switch) and increased up to 8 times. Fig. 3.15 shows the change in IPC and average memory access delay over a range of network bandwidth and latency parameters, respectively.

3.6.5 Simulator Performance

Finally, we measure the simulator performance against Gem5 and its scalability with multiple compute nodes and memory pools. Fig. 3.16 shows the number of instructions executed per sec in *DRackSim* on an AMD Ryzen9 machine. Although *DRackSim* is 1.5x to 2x slower than Gem5, we allocate 100% remote memory for performance testing, and the simulation time also includes a switch interconnect model, whereas Gem5 simulates memory accesses without passing through the switch model. The performance difference between the workloads is due to the amount of remote memory access traffic and instruction level dependencies. Figure 3.17 shows the increase in simulation time when scalable simulations are run with multiple nodes. Each compute node allocates 100% remote memory while a separate memory pool is used for each. Even with 16 nodes running one million instructions on each node, the slowdown is 7.5x to 12x. Most of the slowdown is due to the thread synchronization overhead and the queue model of interconnect, which does not scale well for HPC workloads.

3.7 Summary

DMS are being widely studied for their use in data centers due to their significant advantages over traditional server systems, such as improved memory utilization, scalability, and decoupling of memory. However, the commercial unavailability of these systems hinders path-breaking research ideas from the systems research community. Therefore, we propose a simulator *DRackSim* that models highly scalable DMS and supports a wide range of user-defined configurations. We perform rigorous validation of *DRackSim* against Gem5 and conduct a broad set of experiments to demonstrate the working of our proposed simulator. We have made the code [1] publicly available for the community and plan to support it with regular updates with more features and detailed documentation.

Adoption of DMS in data centers require multiple optimizations for reducing the performance impact of high remote memory latency. This can be primarily done by efficient implementation of hot page migration or cache prefetching systems that can hide memory latency. Other mechanisms can also be explored for eliminating remote memory allocation bottlenecks or reducing the queuing delays at the interconnect. In the next chapters, we explore some of these design ideas that significantly improves the system performance.

[1] https://github.com/Amit-P89/-DRackSim

Chapter 4

A Practical Approach For Workload-Aware Data Movement in Disaggregated Memory Systems

In this chapter, we present a hardware mechanism for workload-aware data movement between compute and memory pools [42] that significantly reduces the AMAT and improves system performance. We propose a novel mechanism for implementing a centralized epoch-based page migration for a large scale DMS. Our page migration mechanism first moves hot pages from remote memory to the centralized caches hosted at the switch until a group of hot pages is identified. This, in turn, reduces the data path for the intermediate remote memory accesses to those pages until they are moved to local memory. Further, our approach allows the remote memory pools to serve all the remote memory requests (cache miss/hot page) at cache block granularity by sharing the bandwidth between regular cache miss and hot page remote memory requests. This helps to eliminate long delays on critical cache miss requests introduced due to large granularity page accesses. The proposed design performs 10% to 100% better than traditional RDMA-based DMS that access remote memory at page granularity and 5% to 35% better than baseline DMS. Section 4.1 analyzes the complexities and overheads of accessing remote memory pages in a bandwidth-restricted large-scale DMS. Section 4.2 presents the background and motivation for the proposed design. In section 4.3,

we discuss the system design and architecture of the proposed mechanism. Section 4.4 presents the experimentation analysis, and we summarize the chapter in section 4.5.

4.1 Introduction

The large in-memory applications such as AI, big data, and video/graph analytics deployed in data centers will mostly rely on remote memory for most of their memory requirements. Due to this, the memory access cost will increase 2 to 3 times that of a completely local memory system. A commonly used software/virtual memory disaggregation implements page migration and always accesses remote memory at page-level granularity over an RDMA-enabled network. These systems swap out the memory pages to remote memory mapped from other server nodes rather than swapping to slow disks. The same can be done in hardware-based disaggregated systems, where local and remote memory (in memory pools) are organized linearly, and the memory-binding interconnects also allow block-level access to remote memory on a cache miss. Further, DMS features multi-tiered memory management where a global memory manager manages the address space of memory pools beside a memory manager at compute nodes.

The primary challenge in DMS is the high access cost to access remote memory blocks on a cache miss, significantly impacting performance. There is a scope for exploiting spatial locality in the workloads by migrating hot pages in slower memory to faster memory in addition to the regular cache misses to remote memory, as explained in section 2.4. Accessing a remote memory page requires more network/memory bandwidth that starves the subsequent cache line accesses in their critical path, significantly impacting the system's performance. Further, workloads show a range of friendliness to page migration [86]. Memory access patterns of the workloads define if they are benefited by migration or not. The page migration is more workload-friendly if its memory pages have good spatial and temporal locality. However, all the workloads do not show the same behavior, and it is crucial to set the migration threshold based on the memory access patterns of each workload running on different compute nodes. Therefore, a careful approach is required to exploit the benefits of hot page migration in large-scale DMS.

Prior techniques for page migration in a hybrid memory system (DRAM-NVAM, discussed in 2.4) do not work for large-scale DMS with shared second-tier memory (remote pools), making tracking pages difficult. Page migration has also been proposed for systems with software/virtual disaggregated memory, which allow only paged access and do not support cache block access to remote memory. These systems use RDMA to exploit the free memory in other server nodes but are not a replacement for hardware DMS, which supports multi-granularity remote memory access. Komareddy et al. [38]

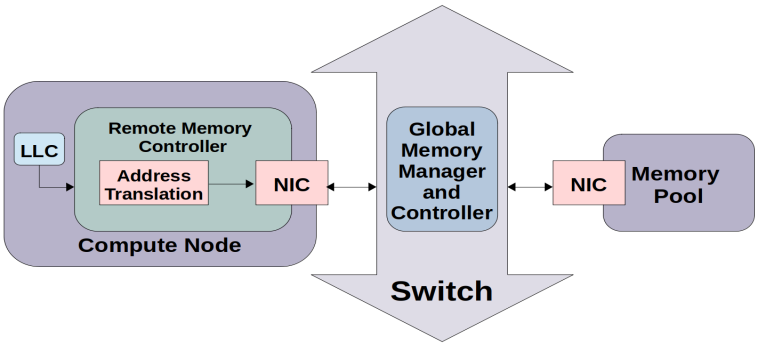


FIGURE 4.1: Baseline Hardware Disaggregated Memory System

proposed a global memory controller hosted at the rack switch. Although page migration in disaggregated memory was proposed for the first time, the controller delays the migrations to miss out on the potential benefits and uses fixed parameters for migration without any intelligence. There is little opportunity to translate the available designs for page migration in hybrid memory or software disaggregated systems to fully DMS.

This work proposes a hardware mechanism for workload-aware data movement in a DMS that implements an epoch-based hot page migration to reduce AMAT. Our approach is based on a centralized page migration system that supports rack-level disaggregation, where multiple compute nodes run simultaneously, each running a workload with distinct memory access patterns. The hardware support comes through an in-network global memory manager [28, 122], which is extended to perform workload-aware page migration. The modified memory manager is implemented at a programmable central switch that utilizes new caching structures at the switch to store metadata as also has been proposed in the past for DMS [28, 122–126], shown in Fig. 4.1. The proposed mechanism works for both shared and distributed remote memory organizations. However, our experimentation assumes no page sharing among multiple nodes.

Even though a page migration brings multiple cache lines to fast memory, accessing a remote page (1.2-1.5µs) is 5 to 7 times more than a block access latency. Also, reading a remote memory page (assuming a 4KB page) and sending its response back to the compute node increases the delays to the critical cache line accesses and starves them of memory/network bandwidth. Our mechanism for remote page access ensures that the starvation to block accesses is eliminated while also reducing the response time for the page access, which is now completed at block-level granularity rather than accessing a complete page altogether. Further, page migration systems (epoch-based [38] and on-the-fly [22]) have their limitations that may even degrade the performance of a DMS. Therefore, we add hardware support that implements an epoch-based approach but gets the advantage of on-the-fly migration by reducing the data path for pages waiting for migration. Finally, we offer software support for centralized page migration that requires carefully selecting the migration parameters for each compute node based on

its memory access patterns. A learning-based page migration policy is implemented that initially learns the workload behavior for each node and fixes the migration parameters. Further, considering all the trade-offs of page migration in DMS discussed in section 2.4, choosing migration parameters wisely for an epoch-based page migration is essential. The centralized switch may only allow small caches to track pages and can not afford to track all the remote memory pages due to hardware and architectural restrictions.

We summarize our contribution to this chapter as follows:

- We propose a novel centralized hardware mechanism for workload-aware data movement in rack-level DMS that reduces the data path for the remote memory accesses until the hot pages are not migrated.
- We analyze the major hurdles of multi-granularity remote memory access in a DMS and implement an approach to neutralize the extra overheads and delays due to page access and migrations.
- Finally, we evaluate our proposed mechanism over various multi-threaded HPC workloads and mini-applications and show a significant improvement in the system performance.

4.2 System Design

Overview: This section discusses the proposed hardware structures to support a centralized page-migration with workload-aware data movement that eliminates the bandwidth and starvation issues. Firstly, the central switch differentiates the memory accesses of individual nodes by reading the access packets and passing this information to the global memory controller. This allows the controller to characterize the access pattern for each node separately and fix the page migration parameters and access priorities. The global memory controller holds multiple new hardware structures whose sizes can be scaled to support any number of nodes. However, a limited number of compute nodes (C) and memory pools (M) are expected to be grouped inside a rack with specific configurations (say, at a fixed ratio of 1C:2M or 1C:4M). These configurations are unknown, as continuous research is being done in disaggregated memory space, while we assume support for a maximum of 16 nodes. The global memory controller's design overview can be seen in Fig. 4.2, which identifies hot pages and schedules their movement along with cache line accesses after bandwidth partitioning. The new hardware structures are 1) Hot-Page Tracker, 2) Access Controller, 3) Pending Blocks Accesses queues, and 4) Page Buffers to store accessed memory pages.

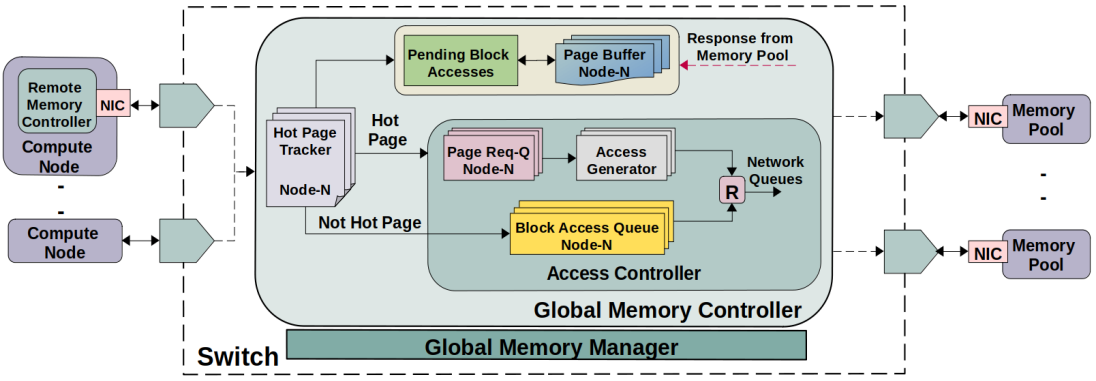
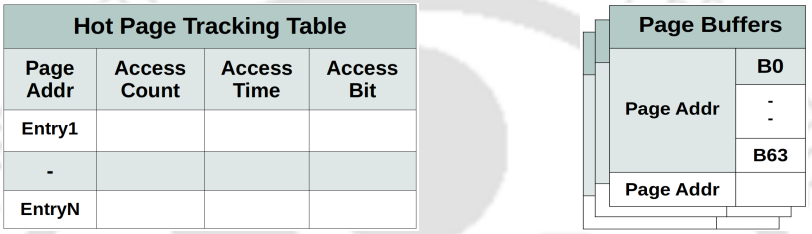


FIGURE 4.2: Centralized Page Migration Support with Workload-Aware Efficient Data Movement, 'R' represents a Request Selector



(a) (b)

FIGURE 4.3: (a) Hot Page Tracking Table structure (b) Page Buffer structure

4.2.1 Hot Page Tracking

The controller supports predicting hot pages by tracking access count and frequency to remote pages. Figure 4.3a shows the hot-page tracker (HPT), which consists of a table to store information on the most recently accessed pages with a maximum of 100 entries per node (also limits the number of pages migrated together). Each entry consists of the *page address* (32-bit), *access count* (16-bit), *first access time* (32-bit), and *Access_bit* (1-bit). However, there could be more active pages at a time that will not fit in the cached table. In that case, a new entry will be created by replacing the old one with minimum access count and oldest access time. The evicted entry is kept in a similar table at the switch DRAM and loaded back when that page is re-accessed. A page is identified as hot when it crosses the hotness threshold (explained later) and is based on access count and reuse frequency. The reuse frequency can be calculated using the page's access count and first access time. On identifying a hot page, a request is added to the page request queue (inside the Access Control block) with its page address, and the *Access_bit* is set to 1 and remains set until it is migrated to the compute node. The future memory accesses to the same page are not sent to the memory pool and are completed at the global memory controller (described in subsection 4.2.4). However, if the page is not hot yet and *Access_bit* is '0', the block request is added to the block access queues (shown in yellow in Fig. 4.2).

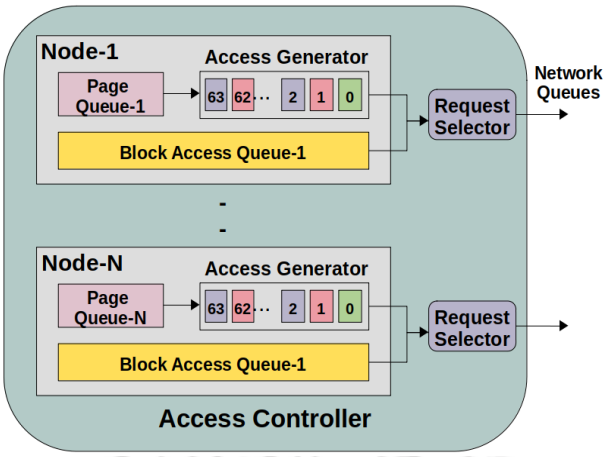


FIGURE 4.4: Access Controller to control Multi-granularity Access

4.2.2 Performing Migration and Using Page Buffers

Once the number of hot pages in the HPT becomes equal to NPM, the migration will be performed by swapping the same number of local victim pages at the compute node to the address of the migrated remote pages. One limitation of the epoch-based page migration is that many benefits are lost until the batch of hot pages is ready, especially for workloads with a high temporal locality. This can be eliminated with on-the-fly page migration, which instantly migrates a page as it is identified as hot but has high overhead due to frequent TLB shoot-downs. We take a middle path by using page buffers (as cache) at the global memory controller with space for 100 pages per node, shown in Fig. 4.3b and 4.2. The page is instantly accessed as it becomes hot and kept in this cache until the whole batch is ready to migrate. When memory access to any of these pages arrives, it is completed at the controller itself through page buffers (costing less than half of a remote memory access cost), getting the best of two techniques. Further, keeping this buffer at a central controller also allows clean access to shared pages between nodes (in the case of a shared memory approach).

4.2.3 Access Controller

Handling Page Access: Once a page is identified as hot, its request is added to the Page Request Queue but is not sent to the memory queue as it is. Firstly, remote page access latency is high, which delays the pending block-level accesses to the same page. Secondly, page access occupies the available memory/network bandwidth and obstructs subsequent block accesses to other pages, adding long delays again. We overcome this problem by servicing page requests at a finer granularity and responding as soon as a block request within a page completes at the memory pool. Fig. 4.4 shows the detailed mechanism by which the Access Controller handles the page and block-level requests. A

dedicated set of queues is present for each connected node to handle page and cache block requests. The access generator breaks the page request into multiple cache line requests. A page request eventually accesses 64 contiguous memory blocks (4KB page with 64B block). Rather than completing page access in one go, it is accessed block-by-block. The access generator will pick a page address from the front of the page request queue and generate 64 block accesses to that page from block-0 to block-63 (using a fixed-size queue at the access generator). Access control has separate hardware structures for each node where its request selector forwards one of the requests from either the block access queue or the access generator.

Handling Block Accesses and Bandwidth Allocation: When block accesses do not belong to a page request (the page is not hot), they are treated as regular requests and kept in block access queues (one for each node). Like page queues, each node has separate block access queues. We implement bandwidth partitioning between the page and block-level accesses to eliminate starvation and reduce waiting times for all types of access. In each cycle, the request selector will choose one of the requests either from the block access queue (for regular block access) or from the access generator queue (part of page access). Further, each queue can be allocated different priority levels to prioritize one type of request over another. The controller adds extra information to the packet header of selected requests before forwarding them to network queues to differentiate between regular block requests and those belonging to the page access. The response packet from the memory pool also includes the same information in its header, allowing the controller to take appropriate action when a response is received. If the response packet belongs to page access, it stores the block in the appropriate page buffer by matching the destination compute node and page address. If the response belongs to regular block accesses, it is directly forwarded to the destination compute node.

4.2.4 Pending Block Accesses

Once a page access request is sent and the page is undergoing access or is present in the page buffer, all the block accesses to that page are halted at the controller and directed to Pending Block Access queues, as shown in Fig. 4.2. The response is instantly sent to the compute node if the block is available in the page buffer. However, the block request waits in the queues if it is not there. The queues have separate buffers for storing reads and writes. Reads queues store page address and page offset, whereas write queues also have space to store a block of data. When a new block arrives in the page buffer, it checks for a pending block access with the same address and completes the access by sending a response back to the source node. In case of a pending write request, the data block inside the page buffer (fetched from the memory pool and is dirty now) is updated with the data in pending write queues.

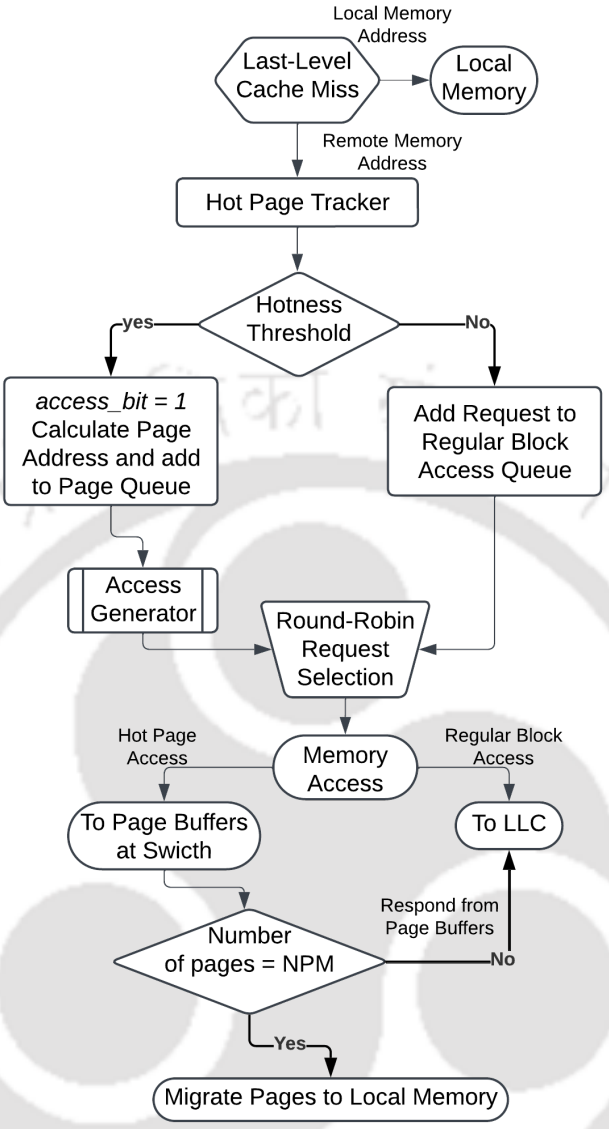


FIGURE 4.5: Flowchart representing the data path for remote memory access

4.3 Remote Memory Access Data Path

Figure 4.5 shows the data path for the remote memory access for regular memory access and hot page migration. Whenever a block request from a compute node arrives at the global memory controller, it will update the page tracking parameters in the hot page tracker. If the page crosses the hotness threshold, a page access request is created using access generators, and *access_bit* is set, while the current block request is also added to pending block queues. However, the request is added to regular block access queues if the *access_bit* is '0'. From here on, page requests will be partitioned into multiple block-sized requests using an access generator, and a round-robin arbitrator select requests from one of the queues. The regular memory accesses will be completed

as usual by sending their response to the requesting compute node. On the other hand, if the response packet belonging to the page request arrives (identified by the packet header), the response is stored in the page buffer of the respective node buffer at an entry matching the page address. The pending block accesses to that page are also served if the page address matches. However, the pages are only migrated when a whole batch of hot pages is available in the page buffer. The global memory controller will then notify the respective compute node to perform page migration, for which the OS at the compute node evicts an equal number of local pages. The eviction can be performed using basic memory page replacement policies such as LRU, clock replacement, or finding cold pages using access counters.

4.4 Hardware Overheads

We discuss the overhead of proposed hardware cache structures at the global memory controller to support a maximum of 16 nodes in any node-to-memory pool configuration. The HPT has 100 entries per node with 71 bits for each entry, approximately 14KB for 16 nodes. The page request queue inside the Access Controller has only four entries per node, which is the maximum number of on-flight page requests. Each entry stores a page address (32-bit). The access generators have 64 entries (one for each page block) with a 32-bit page address and a 6-bit block address. The access controller requires 5KB in total for 16 nodes. The size of regular and pending block access queues will depend on the MSHR size of the LLC at the nodes. The memory requests will also be distributed among these queues, many of which are served instantly without delay. For a node with a 32-core system and 256 entries in MSHR, we assume 96 entries in the regular block access queue and only 32 entries for the pending block access queue (equally divided for pending reads and writes), as only a few on-flight page requests can be there. Each entry stores the page address (32-bit), block address (6-bit), and source node-id (4-bit) for memory access. The pending block queues additionally have 64B of data for write requests. The total size of all these queues is around 26.5KB for 16 nodes for both of these queues. Finally, the page buffers store 100 pages of size 4KB per node with its 32-bit address. This will require a slightly bigger cache of around 6.25MB but provides significant benefits by using positives from both on-the-fly and epoch-based migration.

4.5 Characterizing Workloads with Training

Setting Migration Parameters: We analyze the memory access pattern for each compute node to set the hotness threshold. An epoch-based page migration policy requires setting three migration parameters. However, if the number of hot pages and

hotness threshold are known, then a fixed epoch length is not required, as the system will reach a stage when the other two conditions are met. We also do not fix the NPM parameter and start migration with a small NPM (say 25) while changing it dynamically based on the feedback from the compute node, which is a more practical approach than fixed values. The hotness threshold is set using the collected information during training. When a process starts, page migration is kept off initially for a few million cycles, during which the global memory controller collects the access count and reuse frequency of all the touched pages in its DRAM (in the same way as during hot-page tracking). At the end of this phase, the pages are sorted by access count, and filtration is performed to remove less significant entries. The page entries with an access count lesser than the mean are removed. The filtration may be repeated to set a more conservative threshold until the list does not get too small (20-30% of the initial size). Finally, threshold parameters are calculated using the mean of access count and reuse frequency of the leftover page entries.

Migration Feedback: The OS at the compute node can run a daemon program in the background to evaluate the benefits and the overheads of page migration. We track accesses to migrated and victim pages for each migration batch and evaluate it after every few batches. Based on this evaluation, a feedback score is generated and shared with the global memory controller. System overheads not only include the TLB shoot-down time (T_{inv}) but also the time to copy pages (T_{copy_pages}), time to re-access invalidated TLB entries (T_{tlb_miss}, which is the number of pages migrated multiplied by TLB miss time), and time to access victim pages in remote memory (T_{Vpage_acc}). Eq. 4.1 shows the calculation of total overhead. The benefits are calculated by multiplying total memory accesses (Acc_Count) to migrated pages with the difference in remote and local memory access time ($AMAT_{Remote\ -\ Local}$), in eq. 4.2. Finally, a migration score ($Score_{mig}$) is produced using eq. 4.3, normalized on a scale of -100 to 100 and sent to the global controller. The controller modifies the NPM (NPM_{new}) of a compute node based on its feedback score, which is either positive or negative according to the overheads and benefits of page migration. If the overhead is negative continuously or NPM falls below a certain threshold, re-initialization is done to reset parameters.

$$Overhead_{mig} = T_{inv} + T_{tlb\_miss} + T_{copy\_pages} + T_{Vpage\_acc} \tag{4.1}$$

$$Benefit_{mig} = Acc\_Count \times AMAT_{Remote\ -\ Local} \tag{4.2}$$

$$\underset{\lim\ (-100 \to 100)}{\%Score_{mig}} = \frac{(Benefit_{mig}\ -\ Overhead_{mig})}{Overhead_{mig}} \times 100 \tag{4.3}$$

$$NPM_{new} = NPM + NPM \times \frac{Score_{mig}}{100} \tag{4.4}$$

4.6 Experimental Analysis

We use various HPC applications and workloads that mimic different scientific applications and have a variety of memory access patterns and footprints (ranging from 50MB to 830MB). Table 4.2 mentions all the selected workloads with their functionality. We skip the initial single-threaded regions for each workload and simulate 200 million instructions only for the multi-threaded region. Table 4.1 mentions all the simulation parameters at the compute nodes, memory pools, and the interconnect. We evaluate our proposed design over two network configurations, one with 100Gbps and 400Gbps bandwidth at NIC (compute node and memory pool) and switch, respectively. The other one uses 40Gbps and 100Gbps of bandwidth. The experiments were conducted using the cycle-level simulation mode of *DRackSim*.

Table 4.1: Simulation Parameters

CPU	3.6GHz, 4-core, 2-width 64-InsQ, 64-RS, 192-ROB, 128-LSQ
L1 Cache	32KB(I/D), 8-Way, 2-Cyc
L2 Cache	256KB, 4-Way, 20-Cyc
L3 Cache	2MB per core shared, 16-Way, 40-Cyc
Cache Type	Write-Back/Write-Allocate, Round-Robin
Memory (Local/Remote)	1200x2MHz DDR4 DRAM (19.2GB/s)
Switch	100/400Gbps, 5ns for processing/switching
Network Interface (Nodes)	40/100Gbps, 10ns (de)packetization/processing

Table 4.2: Benchmarks

SimpleMOC(s) [120]	Light Water Reactor Simulation
miniFE [118]	Unstructured Implicit Finite Element Codes
Lulesh [117]	Unstructured Hydrodynamics
XSBench(l) [121]	Monte Carlo Neutron Transport Kernel
Testdfft [127]	Fast-Fourier Transform for HACC
Pennant [119]	Lagrangian staggered-grid hydrodynamics
NPB (bt, dc, ft, mg) [114]	Computational fluid dynamics

4.6.1 Results

We evaluate our page migration system with other data movement scenarios.

- **Page** represents a traditional DMS where data is only transferred at page-level granularity. However, these systems are inherently slow compared to hardware

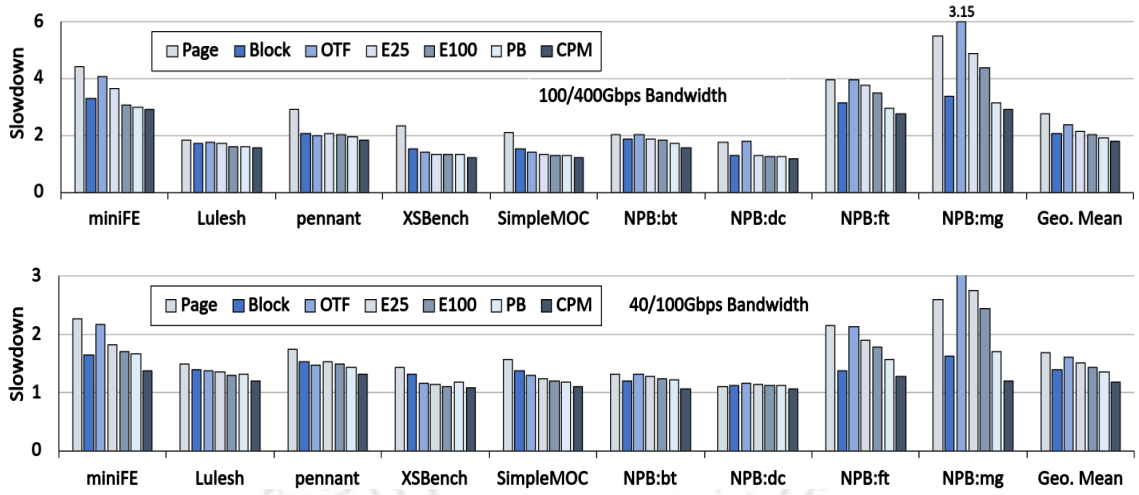


FIGURE 4.6: Performance Slowdown for all the workloads with different data movement policies

DMS. So, we use similar page buffers (as in our design) to delay the page-table updates.

- **Block** represents a baseline hardware memory disaggregation with all the remote memory accesses at block-level granularity.
- **OTF** is an on-the-fly page migration on the same system without extra hardware support.
- **E25** and **E100** represent epoch-based page migration in batches of 25 and 100 pages, respectively.
- **PB** is the same as our proposed design, but the remote page access is made all together without any bandwidth partitioning, and the response is sent as a 4KB packet. Also, there are no pending memory access queues for in-flight page requests at the global memory controller.
- Finally, **CPM** is our proposed centralized hot page migration system with all its features enabled.

Further, we use the same hot-page identification mechanism for OTF, E25, and E100. The memory page allocations are performed across local and remote memory at a fixed ratio of 50:50 using a round-robin policy (unless mentioned otherwise). To keep the memory ratio constant, we pre-evict the same number of victim pages from local memory for every page migration using a clock-replacement policy.

Impact on System Performance: We first evaluate the slowdown in system performance compared to a system using entirely local memory. We run each workload in a single node configuration using one remote memory pool. As shown in Fig. 4.6, CPM

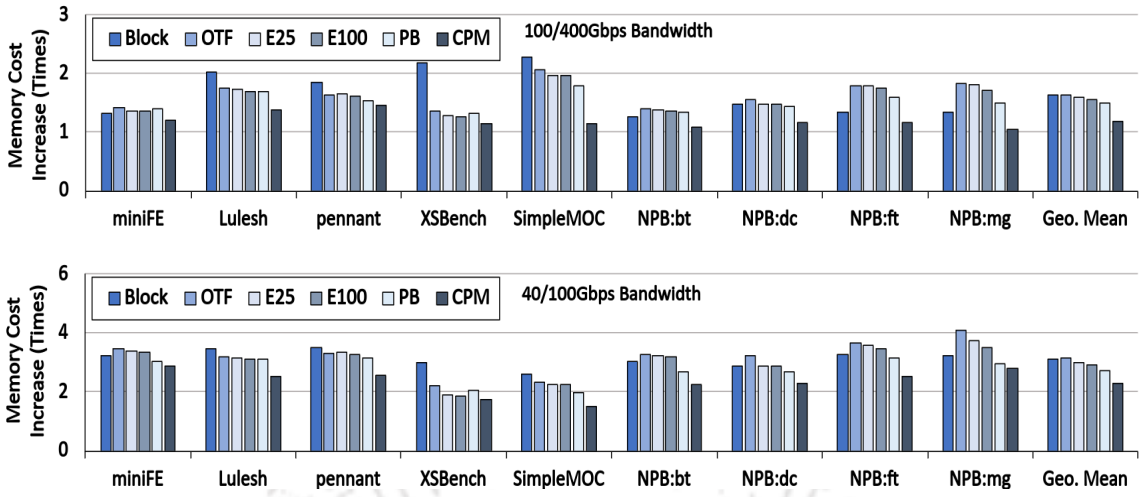


FIGURE 4.7: Increase in memory access cost for all the workloads with different data movement policies

experiences the minimum slowdown among all the data movement schemes for all workloads at both network configurations. With 100/400Gbps bandwidth, the performance for *XSBench*, *NPB:bt*, and *NPB:dc* is very close to the system with 100% local memory even with 50% of memory footprint at local memory. As expected, PAGE suffers the maximum slowdown as it has to access all the remote pages at 4KB granularity, which increases the waiting times of pending memory accesses to those pages. Epoch-based page migration, such as E25, and E100, improved performance compared to PAGE but does not always perform better than the baseline BLOCK, as it misses out on many benefits due to the long waiting time before a batch of pages is ready to migrate. Only in a few cases, when a workload has good spatial locality, epoch-based migrations perform better than the baseline. On the other hand, OTF suffers severe slowdowns in some cases (miniFE, NPB:ft, and NPB:mg), when more pages are migrated. As there are no page buffers with OTF, it performs worse or equivalent to PAGE in these cases due to regular CPU stalls during TLB shoot-downs. PB could eliminate the CPU stalls by using page buffers, not miss out on the migration benefits due to instant migration, and improve the performance for all workloads compared to baseline BLOCK. Finally, CPM further improves the performance of PB by proper bandwidth allocation to page and block requests and eliminates starvation. Further, CPM managed good enough performance even with 40/100Gbps for most workloads except miniFE, NPB:ft and NPB:mg, as they have the maximum number of cache misses.

Impact on Memory Access Cost: Fig. 4.7 shows the increase in memory access cost for all the above data movement schemes over two network configurations. As depicted by the system performance, CPM has the lowest increase in overall memory latency and averages around 1.25 times compared to local-only memory latency over a 100/400Gbps network. In the case of a 40/100Gbps network, the average increase in memory cost is around 2.25 times the local memory access latency. The baseline BLOCK and OTF

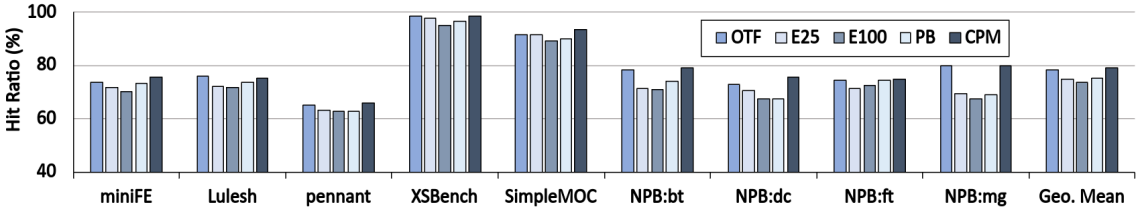


FIGURE 4.8: Percentage of memory access at local memory due to migration of pages

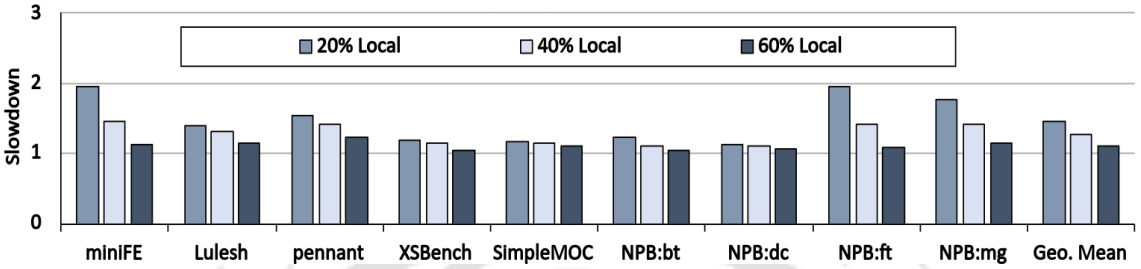


FIGURE 4.9: Performance slowdown on changing the local memory footprint

suffer the most in their memory access latency. However, memory latency does not reveal the performance slowdown for page migration systems, especially for OTF, as it hides the long CPU stalls after the migration. We do not show the results for PAGE because page requests are queued up at remote memory due to significantly high page access times that could not correctly represent the waiting times for LLC misses.

Impact on Hit-ratio at Local Memory: Fig. 4.8 shows the percentage of memory accesses completed at local memory as the consequences of page migration. For all the workloads, CPM has most of the memory accesses at local memory. The results for PB and CPM also include a few percentages of the pending memory accesses completed at the global memory controller using page buffers until a batch of pages gets ready to migrate. OTF experiences a similar percentage of local memory accesses compared to CPM, but the overheads did not allow them to experience similar performance gains. Further, CPM and PB have a significant difference in local memory hit ratio, that is because the pages are accessed at 4KB granularity in PB, which takes more time, and many block accesses to those pages are completed at remote memory before the migration.

4.6.2 Sensitivity Analysis

We further experiment by changing the memory-related parameters and different deployment configurations by changing the number of compute nodes and memory pools.

Local-to-Remote Memory Allocation Percentage: First, we change the memory allocation percentages at the local and remote memory by allocating pages in the same percentage (1 out of every 5-pages is allocated local memory to maintain 20% local

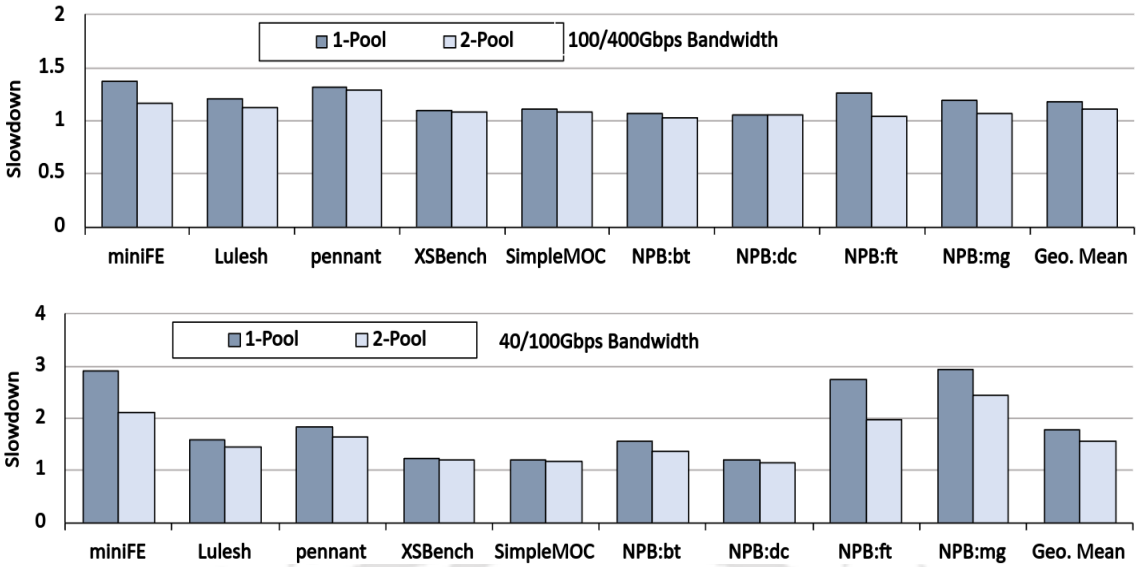


FIGURE 4.10: Performance slowdown on using multiple memory pools over different network configurations

footprint). Fig. 4.9 shows the performance slowdown with 20%, 40%, and 60% of local memory compared to a system with completely local memory. For XSBench, SimpleMOC, NPB:bt, and NPB:dc, the performance with even 20% of the local memory is around 80% of the local-only systems due to the high spatial locality in these workloads. Once the pages are migrated, most of the memory accesses are completed in local memory. With the novel hardware mechanism of CPM, even during page access and migrations, the overall impact of using remote memory is minimal. On the other hand, miniFE, NPB:ft, and NPB:mg faces more slowdown due to a decrease in application footprint on the local memory.

Multiple Memory Pools: Next, we evaluate the performance improvement when a compute node uses multiple memory pools rather than a single one. This results in an overall increase in the memory bandwidth and improves the memory access latency by reducing contention at the remote memory queues. Fig. 4.10 shows the performance slowdown for each workload compared to the local-only system when the remote memory pages are spread across multiple memory pools. As we can see, in both network configurations, the workloads face fewer slowdowns when their pages are allocated across two memory pools than a single. The slowdown is more significant in the case of a 40/100Gbps network.

Multiple Compute Nodes and Memory Pools: Finally, we evaluate different configurations of multiple compute nodes and memory pools, which is the expected way of deployment for the hardware DMS. We consider 8-compute nodes and configure them in a 4:1 or 2:1 ratio with memory pools. The local-to-remote memory allocation ratio is fixed at 50:50, and the memory pool selection is done using a round-robin policy (to allocate a 4MB chunk on each request by the compute nodes). Fig. 4.11 shows the

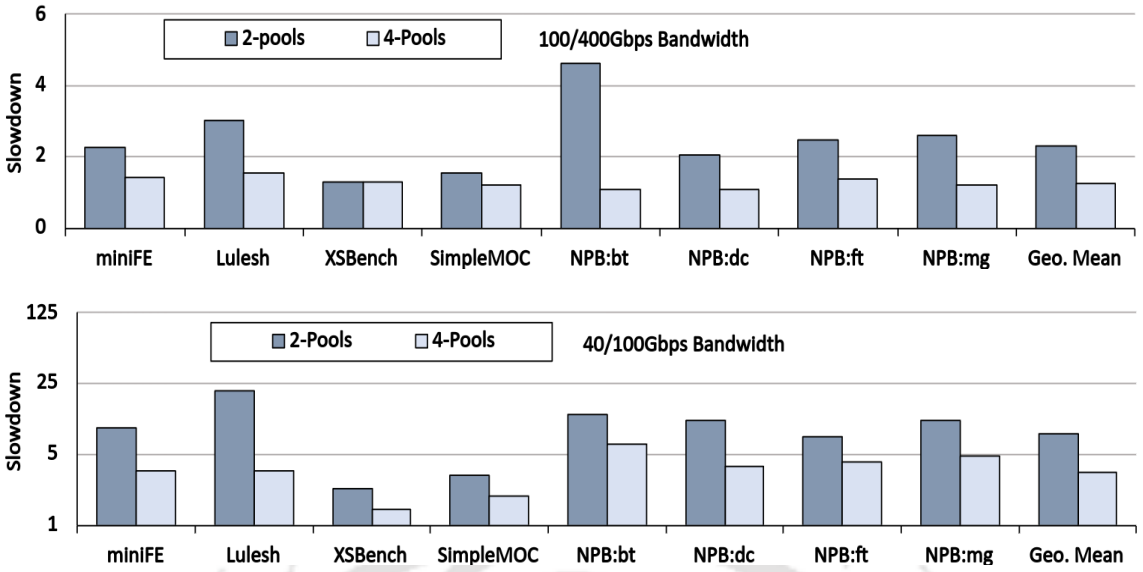


FIGURE 4.11: Performance Slowdown with Multiple Compute and Memory Nodes over different network configurations

performance slowdown for all the workloads (one on each node) running together with different network configurations and node-to-pool ratios. As we can see, the performance impact is minimal with four memory pools, making it an optimal choice for a node-to-pool configuration ratio. Whereas, over a slower network and a 4:1 ratio, the slowdown is significant and is around 9.6x of the local-only system.

4.7 Summary

Although coherent interconnects in DMS allow cache-based access to remote memory, the high memory latency in such a system is the real concern that significantly impacts performance. A page-migration system may bring down the latency but has multiple issues and cannot be implemented as such in DMS. This chapter proposes a centralized hot-page migration system that eliminates those issues by accessing remote memory at block granularity for both page and cache line requests. This allowed fine-grained bandwidth partition between different types of requests. This significantly reduces the waiting times for regular block accesses caused by bandwidth-hungry hot page accesses. In a non-optimal system, page access would occupy both the memory and the network bandwidth and add long delays to critical memory accesses. Further, the data path of remote memory accesses to the pages waiting for migration is reduced significantly by using small hardware caching structures in the central switch. Our proposed design improves the system performance between 10% to 100% compared to traditional disaggregated systems (page sharing with RDMA) and 5% to 35% compared to the baseline hardware DMS.

In this work, even though the page access was broken down into cache line accesses for control over bandwidth partition, the order of serving cache lines within hot pages is fixed (0 to 63). It does not consider the order in which workloads access the block offsets within the pages. Further, there is no distinction while scheduling the hot remote page access for different workloads. However, workloads can have significantly different access frequencies for the page access. In the next chapter, we explore the possibility of further improvements in the proposed design and removing unwanted delays in the data path by implementing on-the-fly page migration.

Chapter 5

CosMoS: Architectural Support for Cost-Effective Data Movement in a Scalable Disaggregated Memory Systems

In the previous chapter, we proposed a system with centralized epoch-based page migration, which could get benefits of on-the-fly page migration from the cached page data at the switch. Although the memory latency is reduced with the reduced remote memory access path, the scalability can be an issue. The size of cached hardware structures at the switch cannot be changed once manufactured. This limits the number of compute nodes utilizing the caching hardware for implementing page migration. Further, there is scope for further decreasing the remote memory latency with an on-the-fly mechanism if the system overhead of TLB shoot-down is taken care of.

In this chapter, we discuss an architectural solution *CosMoS* for *Cos*t-effective data *Mo*vement in a *S*calable DMS using on-the-fly page migration that can predict, schedule, and optimize the movement of hot pages between local and remote memory. We start by performing a workload characterization to show the limitations of existing mechanisms and to make rational choices in our proposed design. We observe that different workloads show a range of access frequency within the pages. Further, we observe that an access pattern exists for all workloads while accessing different offsets within most of the pages. We utilized these observations to design a page migration mechanism that schedules pages based on their access frequency. Further, the pages are accessed in an order such

that the offsets that are required by the CPU first are accessed before. Our results show 20% performance improvement with *CosMoS* compared to state-of-the-art design and 86% improvement compared to baseline for a large-scale DMS. Section 5.1 gives an overview of the proposed mechanism. Section 5.2 discusses the background and workload characterization with the limitations of existing techniques. We discuss the architecture and design of *CosMoS* in section 5.3 and present an experiment analysis in section 5.4. Section 5.5 summarizes this chapter.

5.1 Introduction

As discussed in the previous chapters, the remote memory address space and the compute node's local memory are organized in a flat address structure over a cache-coherent interconnect. As local memory only holds a small portion of the application's footprint, DMS requires data movement optimizations such as hot page migration to maintain application-level performance. The frequent cache-based access to the remote memory can be supplemented by occasionally bringing a few hot remote pages to local memory. However, the data movement cost and other overheads of page access present a unique design challenge for the hardware DMS. Firstly, remote page access obstructs cache-based access to other remote pages for a long time. The cache misses are on its critical path and any delay will impact the application performance. Secondly, multiple compute nodes access the remote memory in a large-scale DMS. This will further increase the delays when multiple page accesses are scheduled on a memory node ahead of serving a cache miss.

This work builds upon the learning gained in the previous chapter and presents new architectural support for an on-the-fly page migration system with cost-effective data movement in a large-scale DMS. We present a workload characterization to understand the requirements while performing remote page access. Like in the previous chapter, the approach proposed in this chapter is also hardware-based. In addition to the reasons mentioned earlier, the software approaches also have less prediction accuracy. The hot page prediction accuracy is important to avoid bandwidth wastage and ping-pong movement of pages between local and remote memory. Further, the existing solution does not schedule page movement, which is necessary for a large-scale DMS involving multiple nodes that run workloads with different requirements. We extend the hot-page prediction mechanism to assist in scheduling remote page accesses and eliminate unwanted delays due to less critical pages getting scheduled earlier than the critical ones. Our mechanism also supports accessing the offsets that are required first by the CPU using an early response mechanism.

We summarize our contributions in this work as follows:

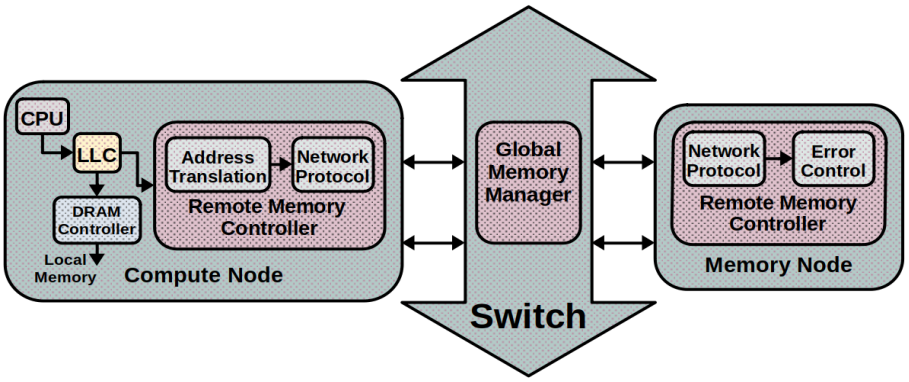


FIGURE 5.1: Memory Controller at Compute and Memory Node

- We present a workload characterization on a wide range of applications to first study the limitations of typical page migration mechanisms and the rationality behind proposing a new hardware-based solution.
- We propose *CosMoS* to exploit the multi-granular remote memory access in a DMS to support an efficient hot-page movement that takes care of bandwidth limitations, performs rightful scheduling and does not disrupt the critical cache-line regular accesses to remote memory.
- Finally, we evaluate *CosMoS* on a large range of workloads representing data center and HPC applications with a custom-built simulator for scalable DMS to study the performance impact in all the possible configurations.

5.2 Background and Motivation

In the last work, our approach was to design a centralized page migration mechanism that also included a central hot-page detection at the switch. However, the approach is not scalable if the compute nodes increase due to the fixed hardware caching structure at the switch. The approach was suitable for both shared and distributed organization of remote memory nodes (or pools). As discussed in previous chapters, a shared approach is only beneficial if workloads span across multiple compute nodes. However, with the high core count in modern processors, there is no scarcity of computing resources within a node, and the distributed approach seems much more suitable. In this work, we consider a distributed approach with no page-sharing across compute nodes and explore the opportunity of on-the-fly page migration where the hot-page identification is performed at the compute node itself. As discussed before, epoch-based page migration misses the benefits until the group of hot pages is available. This chapter proposes a mechanism based on on-the-fly page migration. Although it introduces large migration overhead, it can be eliminated by using extra hardware to store the remapping of page addresses

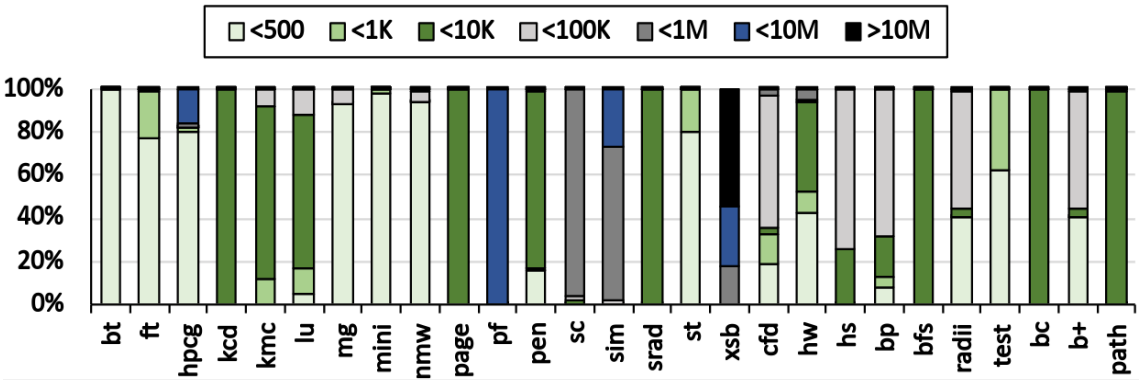


FIGURE 5.2: %age of pages with different access frequency for different workloads

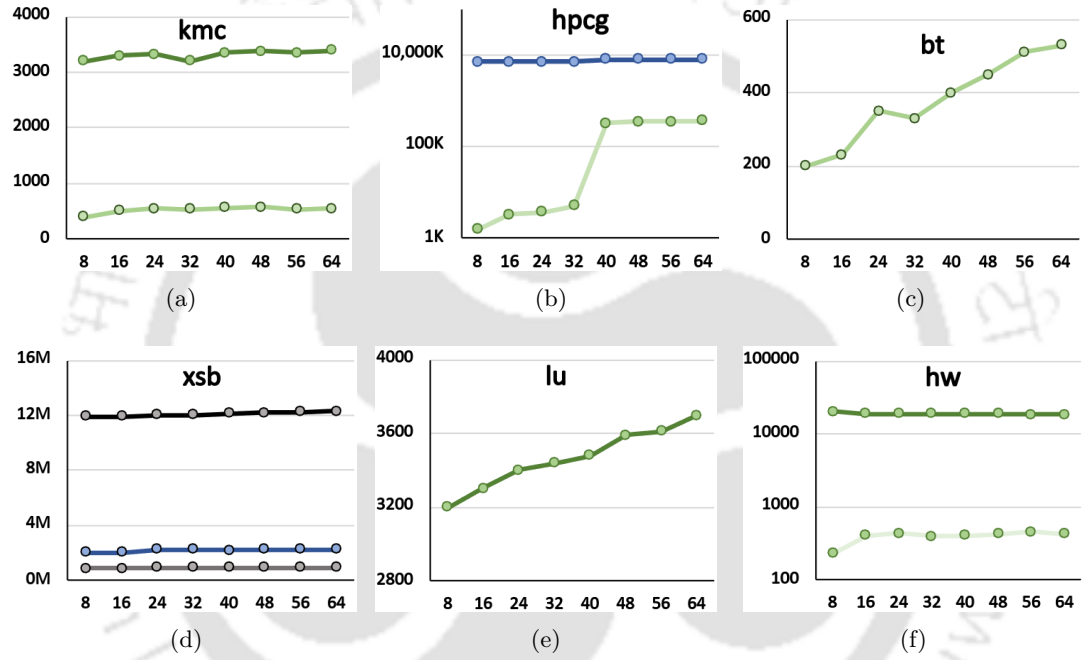


FIGURE 5.3: Rolling Average Access Frequency plot for majority of Pages in different Workloads (same colors as depicted in Fig. 5.2), X-axis represents n^{th} access to a Page, Y-Axis represents the number of CPU Cycles

temporarily [22]. The baseline architecture with respective memory controllers is shown in Fig. 5.1, including the global memory manager at the switch. In this approach, we utilize the respective memory controllers at the compute and the memory node for cost-effective data movement.

5.2.1 Workload Characterization

Page Access Frequency: We characterize a range of workloads from different domains (mentioned in section 5.4) for their page access frequencies. We trace all the memory accesses during the execution of 1 billion instructions on a cycle-level simulator and collect time stamps of individual accesses within each page. Fig. 5.2 shows the percentage

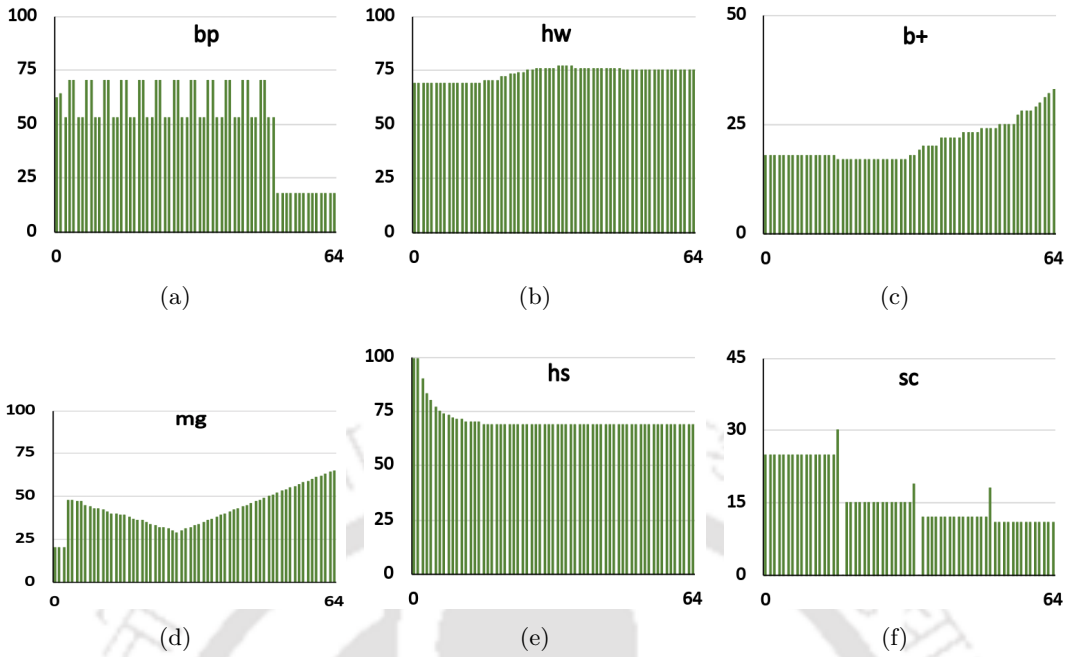


FIGURE 5.4: Access order within the pages in different workloads (using page offset), X-axis represents nth access to a Page, Y-Axis represents the percentage of the occurrence of the majority offset

of pages with different access frequencies in different workloads (the light green specifies the between 0 to 1K CPU cycles, the parrot green range is 1K to 10K CPU cycles, and so on). We have made the following important observations based on it: ① Pages within a workload can have a variety of access frequencies. ② Different workloads can have a significant variation in their page access frequencies. In a large-scale DMS with multiple workloads running on different compute nodes, the large range of access frequencies will allow treating pages differently during migration. Next, in Fig. 5.3, we study the rolling average access frequencies of the majority categories of pages (from Fig. 5.2) for an understanding of the memory access patterns (results shown for a few workloads, although trends remain the same for others). We observe that the rolling average remains the same during future accesses to a page, except in *hpcg*, where it suddenly rises after an initial few accesses for a large number of pages. Further, a good amount of spatial locality exists regardless of the page access frequency.

Page-offsets Access Sequence in Main Memory: Next, we observe that the pages are mostly accessed in a particular order of their offsets in the main memory. We recorded the page-wise offset access sequence to all the pages and found that a particular offset is always accessed the most at every nth access and especially during the initial few accesses to a page. Fig. 5.4 shows the percentage of the most accessed page offsets at every nth access in all the pages. The trend is the same for all the workloads, while some workloads consistently access the memory pages in a fixed order and have 95-100% of common access sequence. However, there is an exception with *xsb*, which does not have

any common access sequence. We utilize all the above observations for accelerated data movement between local and remote memory in a scalable DMS.

5.2.2 Analysis and Limitations

Several other software approaches have been proposed for page migration on multi-tiered memory with unified address structures similar to DMS, except for the bandwidth constraint. The software approaches have certain limitations in precisely identifying hot pages. Other hardware approaches do not consider any scheduling for page movement and treat all hot pages as the same based on their access count. In DMS, multiple layers of memory management control the local and remote memory address space. Further, the interconnect and remote memory bandwidth have to be shared by a large number of compute nodes. These requirements put additional constraints on a DMS compared to any other monolithic multi-tiered system such as DRAM-NVM.

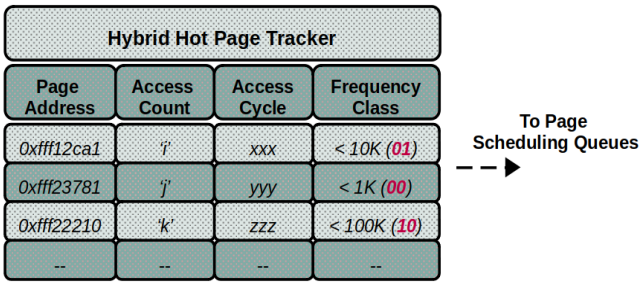
Memory Access Tracking: The software-based page migration [33, 91, 128] mechanisms rely on memory access tracking for hot page identification, which is done by scanning the access bit in the page table entry. However, the overhead associated is very high, and precision is limited due to scanning frequency. Another way is to use hardware-based counters to track events such as LLC miss using precise event-based sampling (PEBS). Firstly, these methods do not track writes, read-for-ownership, and prefetch reads. Further, sampling hardware events and analyzing their traces in software has a trade-off between the overhead and prediction accuracy. These mechanisms usually set very low targets for sampling rates to meet the CPU overhead of 3% to 5%, and the sampling is only performed on a small subset of the memory due to memory overhead constraints. Further, the hotness thresholds are set lightly, e.g., 2 in TPP[33] and 1 in TMTS [91]. These mechanisms work well for NUMA nodes with smaller memory segments but not for DMS that expand to terabytes. Tracking memory access to a possible hot page will be missed between the sampling instances. It also misleads on hotness for pages just accessed during the sampled interval. Software approaches can work fine while targeting super pages for migration as they remain hot for a long time. However, moving super or huge pages in bandwidth-constrained DMS requires special hardware support for fine-granular data migration [23, 88]. Otherwise, it can become a burden and introduce significantly high remote memory latency. *CosMoS* overcomes this issue by using dedicated hardware for hot page identification at each compute node. The cache structure can keep track of multiple pages and remove an entry as soon as the hotness threshold is reached. Although the approach is similar to what was used in the last chapter, however, the mechanism is shifted at the compute node from the switch due to scalability issues.

Dealing with Page Access Frequency: The existing hardware mechanism focuses on hotness prediction accuracy and ignores the scheduling. UImigrate [129] and RHPM [130] measure the relative hotness of predicted hot pages in slow memory with the cold eviction candidates from the fast memory before migrating them. Another focus was correctly predicting hot regions in the super-pages [23, 88] and implementing lightweight page migration only for hot regions. However, all the past works lack any scheduling for page movement and treat all the pages as the same. Pages in a workload or across workloads are accessed at different frequencies in real-time (as shown above 5.2.1) and can be treated differently. The pages accessed more frequently should be migrated faster, while the ones with lesser access frequency can be served slowly. Existing methods either do not perform scheduling or completely ignore the hot pages with lesser access frequency. *CosMoS* implements a multi-queue structure to schedule pages with different access frequency.

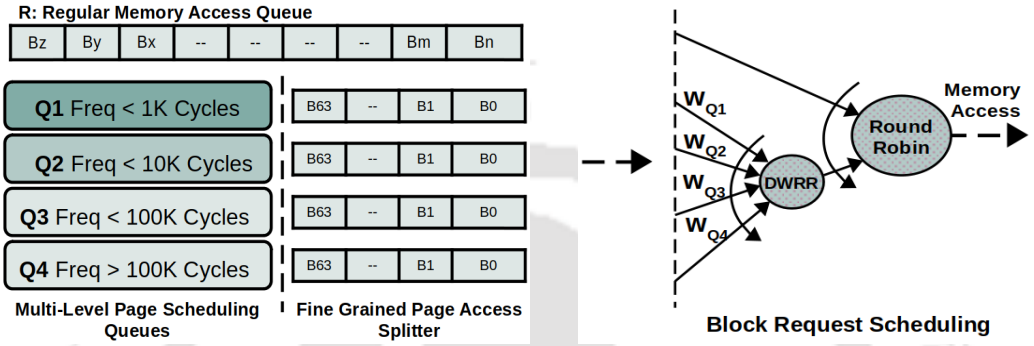
Bandwidth Control: Only scheduling the page accesses is insufficient and requires bandwidth management for remote memory and interconnect sharing, as discussed in the last chapter. As the memory address space is unified, regular cache line access must share the interconnect and memory bandwidth with the hot-page accesses, as multi-granularity accesses are performed simultaneously. However, every page access involves multiple reads from the memory, e.g., 64 cache lines of size 64B for every 4KB page. Due to this, response to the cache misses on their critical path will be delayed, causing a significant performance impact. The delays will increase when multiple hot pages are scheduled for access by different compute hosts on a memory node in a large-scale DMS. On the other hand, all the cache lines within a hot page might not be instantly required and can be delayed a bit in favor of critical regular cache misses. Giannoula et al. implemented bandwidth partition in Daemon [37], bandwidth is divided in a way such that a whole page is still accessed in one go, e.g., for a bandwidth partition ratio of 25:75, a 4Kb page access is followed by a 22 regular cache line accesses ($1/3^{rd}$ of total cache lines in a page). The critical cache misses still have to wait if a page is already under access until it is completed. More delays are encountered at the interconnect as the response packet is sent to the compute node, e.g., a 4KB packet will have a transmission delay of 328 *ns* on a 100Gbps connection. *CosMoS* implements a fine-grained bandwidth partition that responds to both page and cache line remote memory accesses in cache line granularity. Further, *CosMoS* implements a dynamic weighted round-robin to divide the bandwidth among page scheduling queues.

5.3 *CosMoS* Architecture

In this section, we first discuss the important design modules of *CosMoS* and their interaction with each other. We then present the overall design with its data path.



(a) Hybrid Page Tracker



(b) Multi-Queue Page Access Scheduler (c) Bandwidth Controller

FIGURE 5.5: Design Components in *CosMoS*

5.3.1 Design Modules

CosMoS consists of three basic design ideas to implement cost-effective data movement: 1) Hybrid Hot Page Tracker, 2) Page Access Scheduler, and 3) Dynamic Bandwidth Controller as shown in Fig. 5.5. All the modules are integrated to control the flow of remote memory requests and improve the memory response time. We additionally use temporary buffers to support the data movement. We discuss the modules in detail below:

Hybrid Hot Page Tracker: To identify hot pages, the system only needs to track a small set of currently active pages. This set of pages can change after every small time frame, and the tracking system should maintain a large enough access history of the working set. Our proposed hot page tracker works completely in hardware to track only the currently active pages in remote memory with a small hardware overhead. It does not require software involvement other than modifying page tables once the migration is complete. Fig. 5.5a shows the metadata structure stored for each active page at the tracker. Although the cached table only stores a few entries, we extend this table by storing more entries in the local memory in a hash table (hash calculation using page address). In a real system, a small amount of memory (1MB in our case) can be used at a fixed location in DRAM, which will be unavailable to the OS. Once the cached buffer is full, a new entry will replace the entry that is least recently used and is written to the local memory table. This will be brought back if there is access to that page in the

future. Similarly, entries from the local memory table are eliminated using LRU once it is full. The page tracking in hardware has no CPU overhead, and table values are updated in the background. This process has a small memory overhead while looking for existing entries in the memory hash table.

Each table entry stores the following metadata for an active page: the page address, access count, access time (the time stamp of first access to that page), and the frequency class. The frequency class is a 2-bit value that defines how a page will be scheduled for access by the Page Access Scheduler (explained next) once it crosses the hotness threshold. The requirement of 2-bit is based on the number of scheduling classes in the scheduler and is determined through the page access frequency. The page access frequency is calculated by dividing the access count by the time difference between the current and first access.

Page Access Scheduler: Although multiple pages are identified as hot in every small time frame, they all can have different access frequencies. They can be brought to local memory from the remote at dissimilar rates. Our Page Access Scheduler is a multi-queue structure that schedules the migration of identified hot pages at different rates. As shown in Fig. 5.5b, the scheduler has 4-queues, each serving the pages with a particular access frequency. The pages with an access frequency of less than 1K cycles will go to *Q1*, those with an access frequency greater than 1K but less than 10K cycles will go to *Q2*, similarly to *Q3* between 10K and 100K and the rest of them to *Q4*. Once a page is predicted as hot, a page request is added to one of the queues based on its frequency class. The scheduler module, coupled with the bandwidth controller, divides the channel access between multiple queues in a certain ratio to treat them at the desired urgency. The choice of 4 scheduling queues is based on empirical results and the access frequency range shown by most pages in different workloads.

Another design goal of the scheduler is to make it possible for the bandwidth controller to have fine-grained control over the sharing of the network channel for different memory request queues. Apart from 4-page queues, there is a regular memory access queue R through which the LLC misses are directly forwarded to the remote memory for cache block access, in Fig. 5.5b. While the hot pages are accessed, page and cache block requests are interleaved. To eliminate long delays on a critical block request due to one or more leading page requests (as explained in 5.2.2), the page request arriving from any page queue is split into multiple cache block size requests (e.g., 64 requests for a 4KB page with 64B block) by the page access splitter. This will allow bandwidth partitioning at the block level by the controller (explained next) among page and regular memory queues.

Dynamic Fine-grained Bandwidth Controller: As page requests are split into multiple block requests, they are accessed block by block in an interleaved fashion with

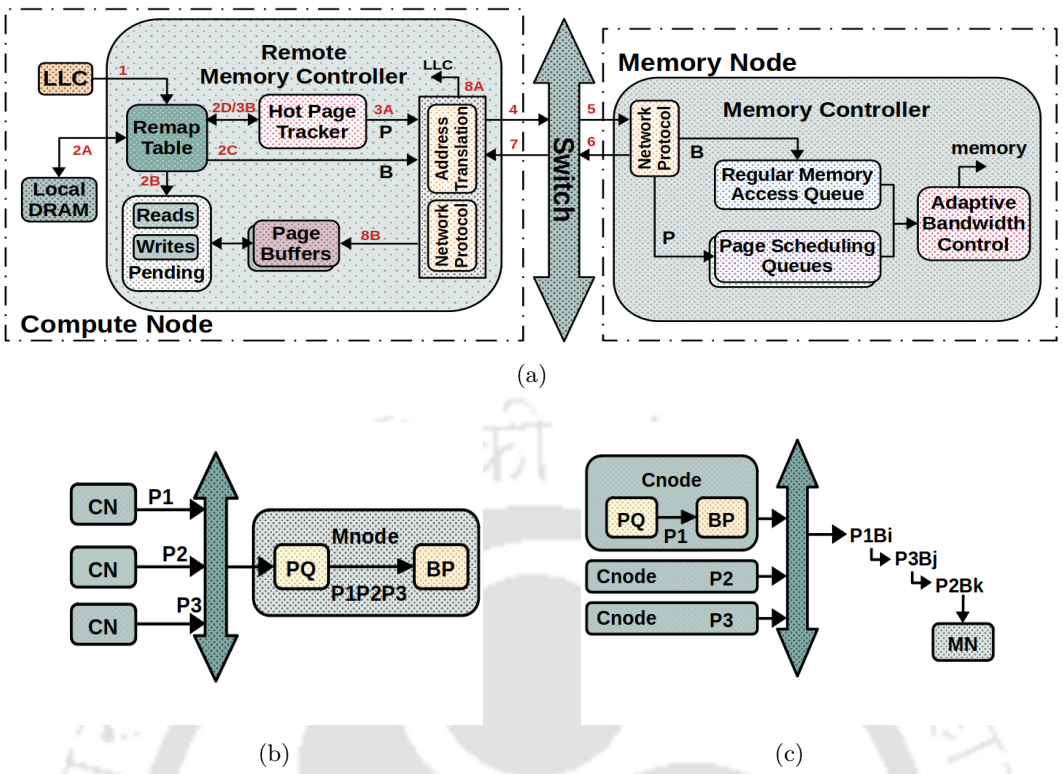


FIGURE 5.6: (a) Complete Design of Memory Controllers in CosMoS, 'P': Page Request, 'B': Block Request | (b),(c) Possible Placement of Scheduling Page Queues (PQ) and Bandwidth Partition Controller (BP) – (b) at the Memory Node (Mnode, MN) (c) at the Compute Node (Cnode, CN)

memory requests from other queues rather than in one go. The interleaving of memory requests from different queues does not add any significant delays to the subsequent cache miss requests in R. The bandwidth is partitioned among these queues to manage delay-sensitive requests and versatile resource allocation. First, it is equally shared between queue R and all the page queues combined using a round-robin. Next, a dynamic weighted round-robin (DWRR) is applied between different page queues and adjusting to changing request rates. Each page queue is given a first chance to send its memory request for the number of cycles equivalent to its weight in every complete round. If a queue is empty during its turn, a request from the next queue is taken. Finally, the weights are updated after a time interval, which is done using a statically allocated weight W_S and queue input rate.

As an example, in every cycle, a memory request is selected from either R or one of the page queues. The W_S allocation for the page queues is user-defined. We allocate it in the ratio of 100:50:10:5 based on the criticality of memory requests in each queue. Assuming that the request rate is the same for all the queues except *Q3*, whose rate is twice of others, the W_D will become 100:50:20:5 for *Q1*, *Q2*, *Q3*, and *Q4*, respectively.

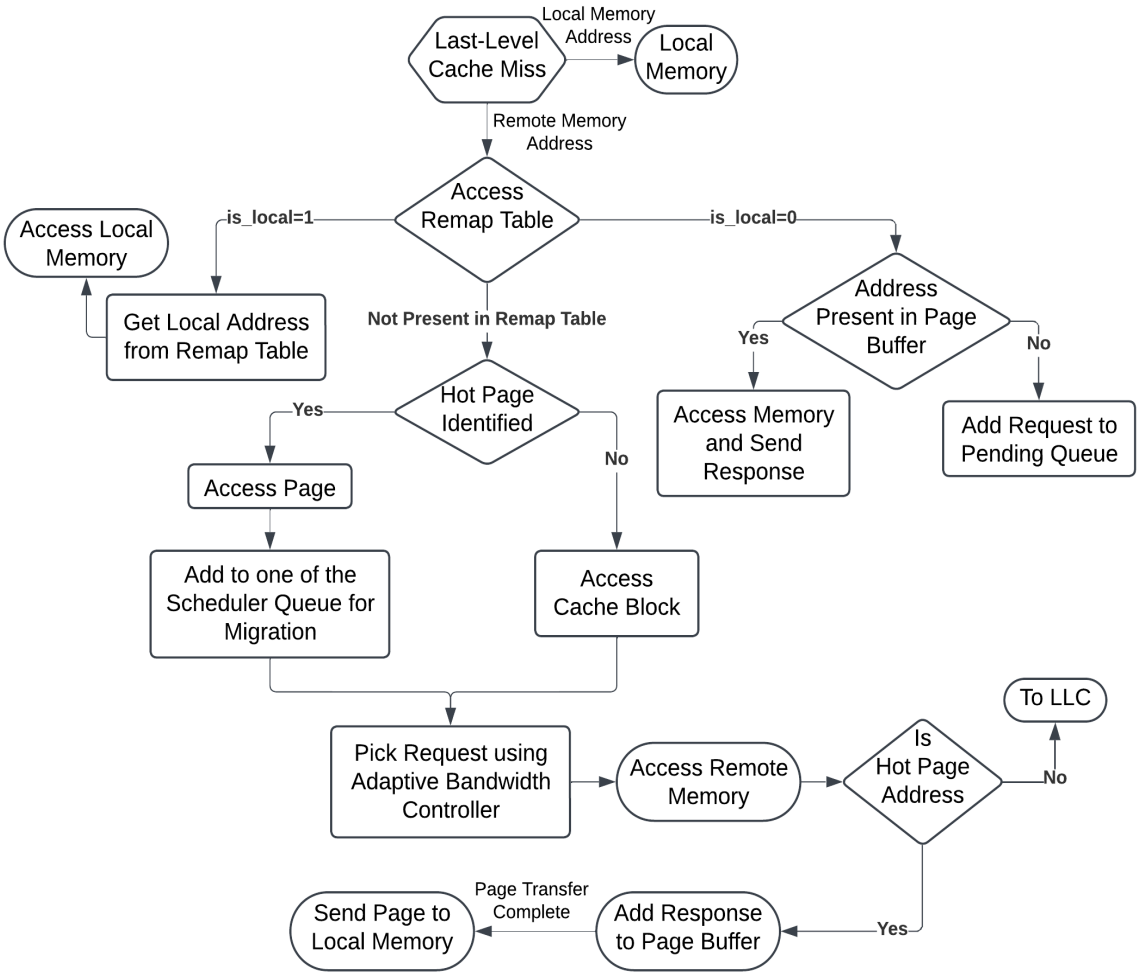


FIGURE 5.7: Flowchart representing the data path for remote memory access

5.3.2 CosMoS Complete Design

We now discuss the complete design with its data path and the placement of design modules. Fig. 5.6a shows the complete design of *CosMos* with all the proposed modules and supporting data structure. A remap table has been proposed in the past [22] for page migration systems to temporarily delay the page table updates. It temporarily stores the address translation of migrated pages with its old and new addresses. We use the remap table and extend it to include *is_local* bit in each entry. Next, We use the page buffers for storing partial page data that is now received as blocks. Lastly, the pending reads (address) and writes (address and data) store the cache misses belonging to in-flight hot pages.

Data Path: The request path is shown using a detailed flowchart in Fig. 5.7. When an LLC miss belongs to a remote memory address, it is first sent to the remap table, where multiple scenarios can occur. ① The page entry is present in the remap table, which

means it is either migrated to local memory (*is_local = 1*) or has been scheduled for migration (*is_local = 0*): (a) if *is_local* is 1, the new physical address is accessed, and the request is forwarded to local DRAM (b) if *is_local* is 0, the request goes towards pending read/writes, where it checks if the block has arrived in page buffers. If present, the response is instantly sent to LLC. Otherwise, it waits in the pending read/write queue until the block arrives (early response mechanism). ② The page entry is not in the remap table and is not even scheduled for migration. In this case, the request is sent for remote memory access as packet 'B', and the access is recorded in the hot page tracker for the future by creating a new entry or updating the previous entry for the page address. The hot page tracker issues a page request 'P' whenever it crosses the hotness threshold and removes the entry while creating a new entry in the remap table. The request types 'P' or 'B' can be determined at remote memory using a bit in the packet header.

From there on, the request goes to remote memory, and its response is received and identified through the packet header. It is directly sent to the LLC if it belongs to a regular memory request queue. If a response belongs to a page, it is sent to page buffers and completes any pending reads/writes to that page. The coherency is also correctly maintained at the page buffers. If a page buffer gets full, it means all the blocks of that page have arrived, and the page is sent to local memory after creating an entry in the remap table. Finally, the page table is updated using re-mappings when the remap table is full and is cleared for future access.

Early Response Mechanism: Normally, the cache misses to a page undergoing the migration would wait at the compute node (introduce long delays until the complete page is accessed and transferred) or can be redundantly issued for memory access (introduce extra network/memory traffic). In both these cases, there is an additional overhead. In *CosMoS*, the partial memory responses belonging to the hot pages can instantly be transferred to the compute node in small packets rather than waiting for complete page access to be sent as a single large packet. These memory responses are stored in a cached page buffer until the whole page arrives (full data path in section 5.3.2). Our approach allows an early response to the cache misses using the partial page data available in the cached buffers.

Further, *CosMoS* eliminates the redundancy in remote memory requests through a separate wait queue for cache misses belonging to in-flight pages. The cache misses are kept pending in the wait queue rather than being re-issued for accessing the data that is already being accessed through pages. The expectation is that the required cache block will soon arrive or is already present in the page buffer, reducing the delays in the critical path. To accelerate this process, each compute node learns the most common offset access sequence within the pages (the most re-occurring access sequence during

the learning, as described in section 5.3.1). The access splitter (in Fig. 5.5b) now generates the block request sequence for the requested page in a particular order (node and workload-specific) rather than blindly generating it from 0 to 63. the overall response time improves for the pending cache misses as the page offsets that are expected to be requested before are served first. The mechanism results in a significant improvement in the local hit ratio without extra overheads.

Placement of Design Modules: CXL 3.0 protocol supports multiple CXL devices (compute and memory nodes) through a switch and is expected to be deployed in large-scale DMS configurations. If the scheduling queues and bandwidth controller are placed at compute nodes, page requests from different compute nodes can be interleaved at the block level, as shown in Fig. 4.3b. This would result in smaller delays in accessing data from pages scheduled within the same queue. In this example, assuming page requests P1, P2, and P3 are scheduled in a same queue, then blocks Bi, Bj, and Bk can be served individually, reducing the page access response time. However, this choice will fail the motive of page migration, as it does not bring multiple cache lines with a single page request and increases the network traffic with more remote memory requests. Contrary, if these modules are put on the memory nodes, similar page requests can not interleave at the block level if they are in the same queue, as in Fig. 5.6c. This can be costly if multiple page requests from different compute nodes are lined up in a queue, especially 'Q1'. However, this arrangement will be less problematic when there are multiple memory nodes (expected in scalable DMS), as the page requests will also be distributed among all of them, keeping the delays minimal. Therefore, it is logical to only perform hot page prediction at the compute nodes and the rest at memory nodes to utilize the page migration mechanism properly.

Super Page Support and Cold Page Prediction: Most modern OS can support huge pages of size 2MB or 1GB; however, we do not recommend directly scheduling huge pages for migration, as it will introduce long delays. The existing mechanism proposes hierarchical hotness identification [23], first to predict a hot super page and then to find hot regions within a super page. With *CosMoS*, the super pages can be first predicted in software, and 4KB pages within hot super pages can be tracked with our hot-page tracker. Further, existing LRU-based cold page eviction works well with page access bits and can be decoupled from the migration [33].

5.4 Experiment Analysis

We choose a variety of multi-threaded workloads from multiple domains, such as Graph processing, bioinformatics, Fluid Dynamics, Machine Learning, Image Processing, and so on, to evaluate CosMoS, shown in Table 5.1. The selected workloads also have a wide

TABLE 5.1: Benchmarks

Rodinia Suite [113]	Srad, BFS, HeartWell(hw) , CFD, BackProp(bp) HotSpot(ht), PathFinder(path), KmeanCluster(kmc), NeedleMan(nmw), ParticleFilter(pf), B+tree (b+), stream-cluster(sc)
Ligra-Graph [116]	KcoreDecomp.(kd), PageRank(pr), BC, Radii
NPB Suite[114]	BlockTriDiagonal(bt), 3DFFT(ft), MultiGrid(mg)
Others	Lulesh(lu) [117], miniFE(mini) [118], HPCG [115], XSBench(xsb) [121], stream(st) [131], TestDFFT [127], SimpleMOC(sim) [120], Pennent(pen) [119]

TABLE 5.2: Simulation Parameters

CPU	64-IQ, 64-RS, 192-ROB, 128-LSQ, width-2, 4-core, 3.6GHz
L1	64B block, 2-Cyc, 8-Way, 32KB(I/D)
L2	64B block, 20-Cyc, 4-Way, 256KB
L3	64B block, 32-Cyc, 16-Way, 2MB/core (shared)
Caches	Write-Back/Write-Allocate, Round-Robin
Memory	1200x2MHz DDR4 (19.2GB/s) (Local/Remote)
Switch	10Gbps, 4MB Port Buffer, 5ns for Proc/Switching
NIC	50Gbps, 15ns for (De)-Packetization/Proc

range of MPKI and memory footprints (30MB to 3.5GB for 1billion instructions). We use cycle-level simulation mode and simulate compute nodes with 4-core CPUs running at 3.6GHz with private L1-(I/D) and L2 cache with a shared last-level cache. The complete system configuration is mentioned in Table 5.2. The bandwidth at the end (compute/memory) nodes is 10Gbps, while it is 50Gbps for the switch. A fixed latency of 15ns is added at end nodes for (de)packetization/NIC_Processing and 5ns at the switch for Packet_processing/switching. Finally, the propagation delay from an end node to a switch is assumed to be 5ns.

5.4.1 Results

We performed extensive experimentation for performance evaluation of *CosMos* in multiple configurations and compared it with two state-of-the-art mechanisms: *Daemon* [37] and *TPP* [33]. We use the same hot page prediction for both and set the hotness threshold at ten accesses. *Daemon* also performs LZ77 page compression before sending the page on the network. It uses four parallel engines, each operating on 256 bytes of data with a total latency of 256 cycles for de(compression) on a 4KB page. The compression

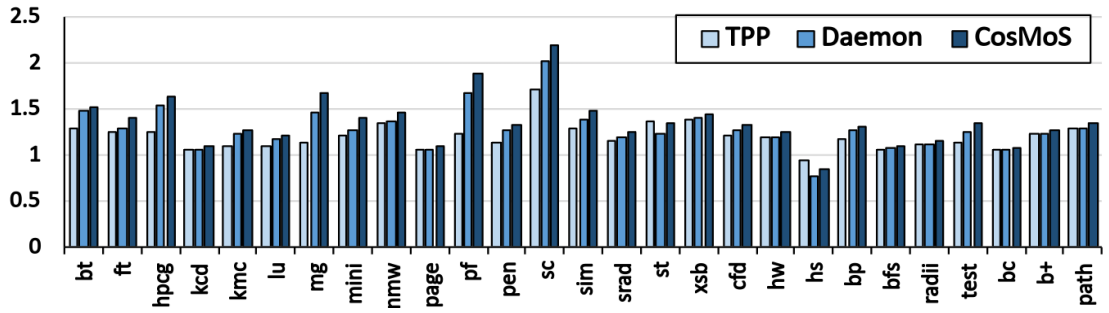


FIGURE 5.8: Performance Speed-Up with *CosMos* vs *Daemon* vs *TPP*, Normalized to Baseline

ratio varies for different workloads and is less when performed on small-sized data (4KB pages) compared to large files [132]. We use a fixed compression ratio of 2:1 on all the transferred pages as an average for all the workloads. Next, we set the memory access sampling at a frequency of 1 out of 100 for *TPP* and '2' as the hotness threshold. We also use the remap table with *Daemon* and *TPP* for a fair comparison.

Single Node Performance: First, we show single-node performance for all the workloads with all the network/memory bandwidth exclusively available to the node. The compute nodes are configured to store only 20% of the workload footprint in local memory, while the 80% is allocated remotely. This is maintained by allocating one out of every five pages in local memory.

SpeedUp: Figure. 5.8 compares the performance of *CosMos* with *Daemon*, and *TPP*. The IPC is normalized to the baseline system, which accesses remote memory only at the cache line. For all workloads, *CosMoS* performs better mainly because of the page access splitting and fine-grained bandwidth partition. In this configuration, the role of scheduling queues is limited, and only those workloads having variation in their page access frequency (*hpcg, lu, pen, cfd, hw, hs, bp, radii, b+*) get additional benefits of multi-queue scheduling. Although most workloads have good spatial locality, the performance benefits for *kcd, page, bfs, radii, b+* are limited due to the sparsity in memory access and lesser MPKI. Other workloads like (bt, ft, mg, mini, st, hs) have a very high MPKI, and their local memory latency increases compared to baseline (although lesser than remote) once the hot pages are brought to it due to limited memory bandwidth (single channel 19.8GB/s). Therefore, these workloads are expected to observe more performance gain in a multi-channel local memory system. The performance of *hs* suffers for the same reason as it issues bursts of memory accesses, and average memory latency increases with page migration due to more number of memory accesses in small time frames.

Daemon, gets most of its performance benefits due to the page compression, which reduces the network time. However, (de)-compression also adds additional latency, and the page access time is still non-trivial. Many cache misses are still served from the remote memory for workloads with high page access frequency until the page is migrated. This

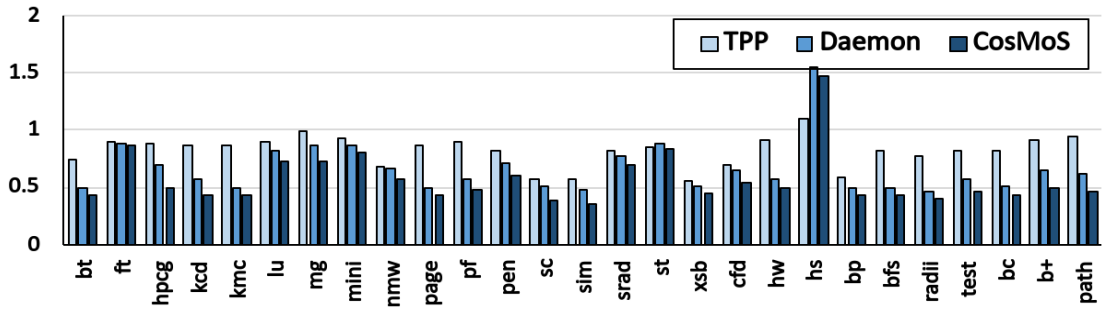


FIGURE 5.9: Improvement in Memory Latency for *CosMos* vs *Daemon* vs *TPP*, Normalized to Baseline

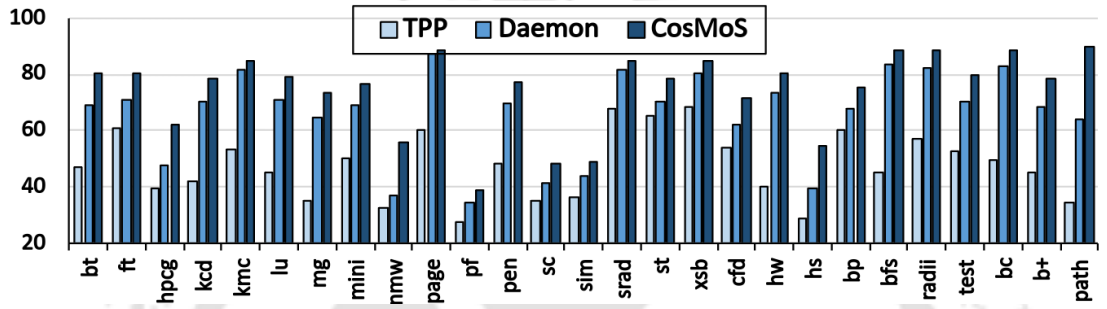
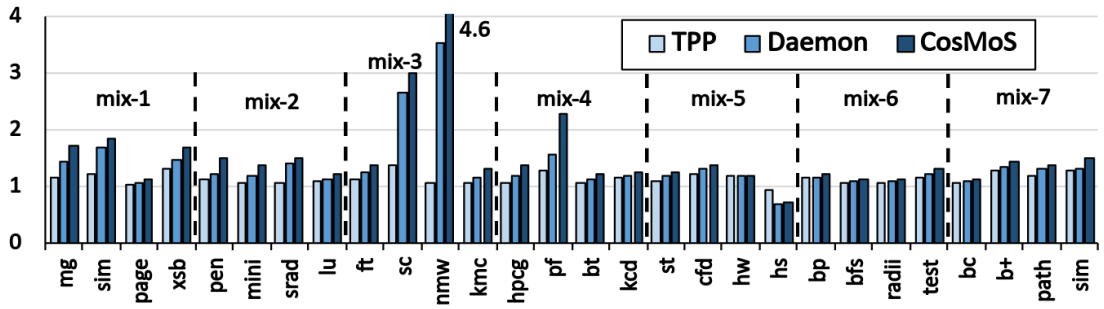


FIGURE 5.10: Local Hit-Ratio with *CosMos* vs *Daemon* vs *TPP*, Normalized to Baseline

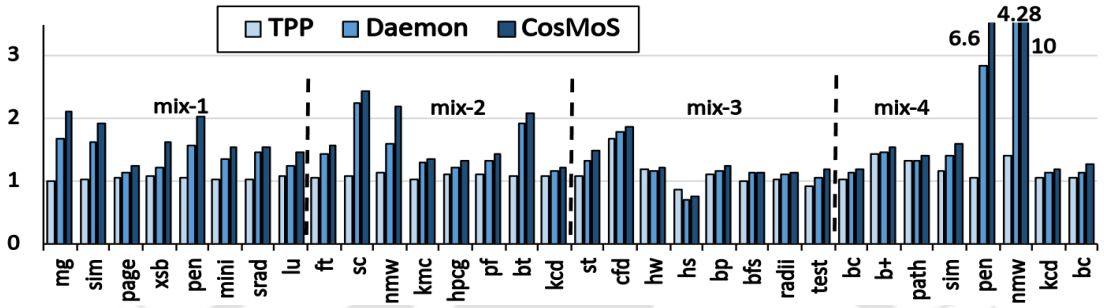
introduces redundancy in memory access and increases average memory latency. Further, large packet sizes in the network also become a bottleneck as it incurs more latency and occupies extra network buffer preventing the critical cache-based access to complete timely network access. *CosMoS* reduces the redundancy with an *Early Response Mechanism* and improves the hit ratio in local memory or the page buffers. Finally, *TPP*, due to constraints on the memory sampling frequency, migrates fewer pages than the other two and has limited performance improvement. The average performance improvement for all workloads is 15%, 23%, and 35% for *TPP*, *Daemon*, and *CosMoS*, respectively.

Memory Latency: Figure. 5.9 shows a decrease in memory latency for all the workloads compared to the baseline system. Except *hs*, we observe a significant decrease for all other workloads. Even though *Daemon* reduces the network time for page access, the effective memory access latency is not reduced to the same extent. With *CosMoS*, effective memory latency is reduced to a relatively larger extent as many of the cache misses are served from the partial page movements in the page buffers. We observe an average reduction of 19%, 34%, and 57% for *TPP*, *Daemon*, and *CosMoS*, respectively.

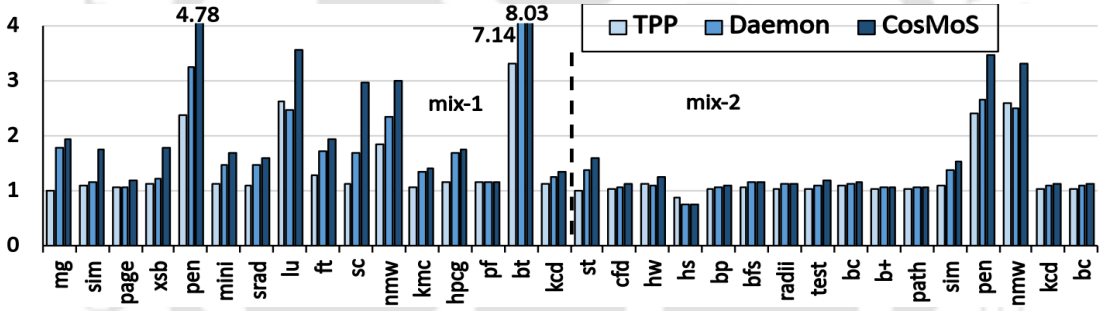
Hit-Ratio Figure. 5.10 shows the percentage of memory accesses completed locally with all three mechanisms. Firstly, the observed speedup is not proportional to the hit ratio. This is because workloads show different sensitivities to memory latency. Further, there are overheads involved in page migration that limit the performance gains. In almost all workloads, *CosMoS* have the best hit ratio with just 20% of the local memory. Due



(a) 4-Compute Nodes and 2-Memory Nodes



(b) 8-Compute Nodes and 4-Memory Nodes



(c) 16-Compute Nodes and 8-Memory Nodes

FIGURE 5.11: *CosMoS* vs *Daemon* vs *TPP* in Large-Scale Configurations

to early responses for page accesses in *CosMoS*, it observes additional hits in the page buffers. Whereas *Daemon* has to wait until the complete page arrives. However, all three mechanisms are still ahead of the baseline, showing a hit ratio of around 20% in local memory.

Large-Scale Performance: Next, we evaluate the performance in large-scale configurations where multiple compute nodes run a workload simultaneously and utilize multiple remote memory nodes. The compute-to-memory node ratio is always kept at 2:1 so as to avail enough memory bandwidth. While allocating a remote memory chunk, the global memory manager selects a memory node in a round-robin manner, and the same memory node can hold chunks for multiple compute nodes. We create workload mixes that run together in a large-scale configuration. The workloads in each 'mix' have a variety of MPKI that represent a real-time scenario. Finally, the remote memory ratio is maintained at 60% for all large-scale configurations from here on (unless explicitly

mentioned), as simulation time grows considerably larger with 80% remote memory in multi-node configuration.

Figure. 5.11 show the performance for 4 (Top), 8 (Middle), and 16 (Bottom) compute nodes, respectively. We observe that *CosMoS* performs even better than the competitors in multi-node configurations. This is because the page scheduling queues, coupled with fine-grained page access and bandwidth partitioning, eliminate longer delays during the page access at remote memory. As the memory nodes are now shared among multiple compute nodes, they will receive simultaneous page requests from compute nodes. With *Daemon* and *TPP*, the page migration delays increase when multiple pages are queued, eventually reducing the local memory hit ratio. However, the page movement mechanism in *CosMoS* allows simultaneous partial access from pages in different queues at a proportionate access rate. Therefore improving the local hit ratio and reducing the redundancy in memory accesses. On average, *CosMoS* performs 1.14x better than *Daemon* in 4-node configurations and 1.20x better in 8/16-node configurations, while it was 1.10x for a single node. Compared to the baseline, the performance gap also widens in large-scale configuration with *CosMoS*. As an example, in the 8-node configuration, it is 86% for *Daemon*, 56% for *Daemon*, and 12% for *TPP* (reduced).

Senstivity to Multi-Queue Scheduling and Fine-grained Bandwidth Partition: To demonstrate the performance impact of various design features in *CosMoS*, we turned off certain components and compared the performance with all features turned on. Figure. 5.12 shows the performance impact of different features, where PQ represents that only a single page queue holds all the page requests with a fine-grained access splitter and round-robin request service from the page queue and regular memory queue. BP represents that the page access splitter is not used, and a fine-grained bandwidth partition is not implemented. However, multiple page queues are there from which pages are served in a round-robin fashion. We can see that BP shows a significant performance hit as splitting page access into multiple block requests allowed bandwidth partitioning at a fine granularity. This mechanism ensured that the regular memory accesses and the page request data were served simultaneously with minimal delays with an early response mechanism. With BP, each page access is exclusive, during which memory bandwidth is not shared with regular memory requests, as done in *Daemon* (but reduces the delays with page compression). On the other hand, PQ has a smaller performance impact comparatively. However, multiple-page queues with extra logic for proper request arbitration can increase the performance.

Senstivity to Node/Switch Bandwidth: Next, we measure the performance impact for *CosMoS* on changing the bandwidth parameters and compare it with *Daemon*. As pages are moved as a single large packet in *Daemon*, it is supposed to show some differences in performance impact at reduced bandwidth compared to *CosMoS*. We configured the end-node (compute/memory) bandwidth to 40Gbps (0.4x of the previous

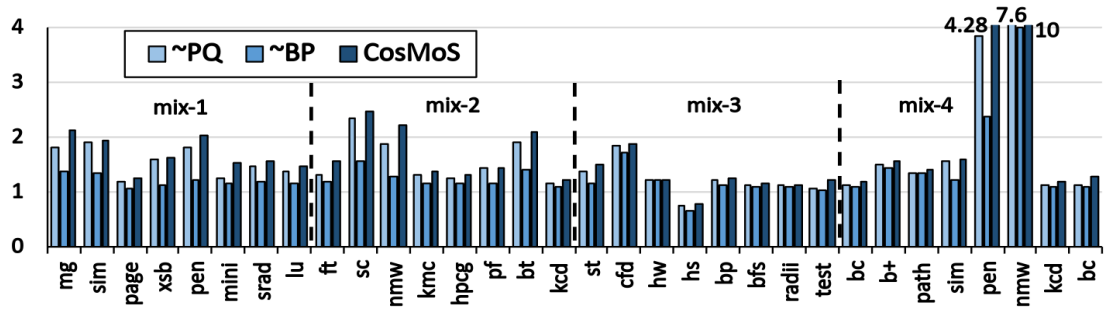


FIGURE 5.12: Performance Impact of Page Scheduling and Bandwidth Partitioning

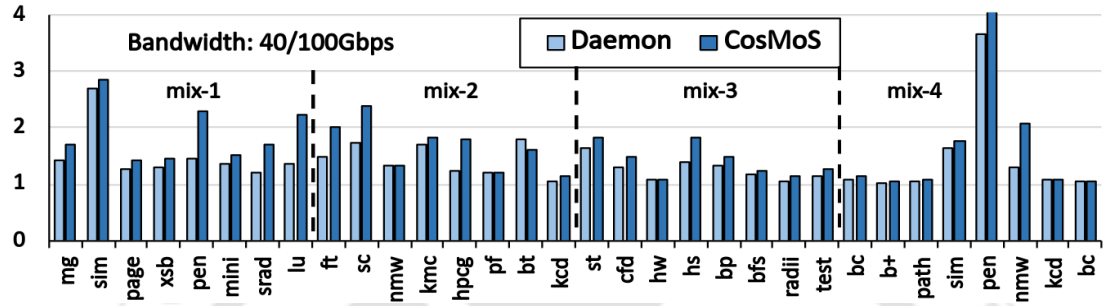


FIGURE 5.13: Impact of changing Bandwidth Parameters

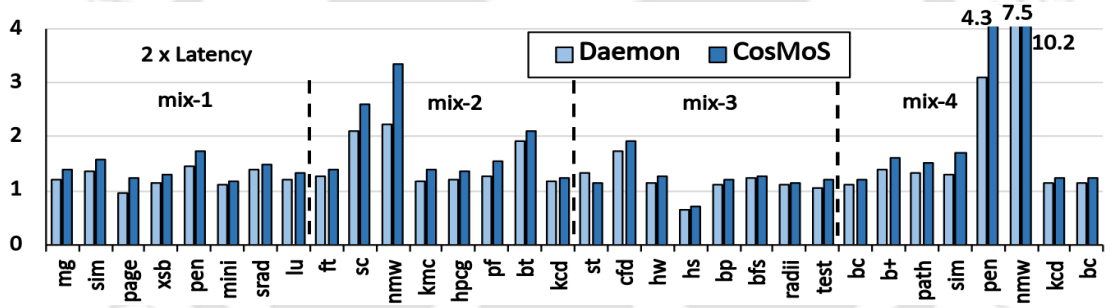


FIGURE 5.14: Impact of changing Latency Parameters

bandwidth) and kept the switch bandwidth at 100Gbps (0.25x of the previous bandwidth). Figure 5.13 shows the performance of *CosMoS* and *Daemon* compared to the baseline in this configuration. With lesser bandwidth, page access time will increase, and *Daemon* observes this impact relatively more. With *CosMoS*, the responses are sent back at cache line granularity, which are small packets of 128B and are less impacted by a change in bandwidth than a page. Therefore, the performance gap increases in favor of *CosMoS* compared to *Daemon*, especially for workloads where a large number of pages get migrated, such as in *pen (mix-1), lu, ft, sc, hs, nmw (mix-4)*. The average performance increase compared to baseline is 69% for *CosMoS* and 42% for *Daemon*. This is a decrease compared to the previous bandwidth configuration. This is due to the extra page transmission time, resulting in a lesser local hit ratio.

Senstivity to Node/Switch Latency: In this case, we increase the packet processing latency at end nodes (30ns) and port processing delays (10ns) at the switch to twice the

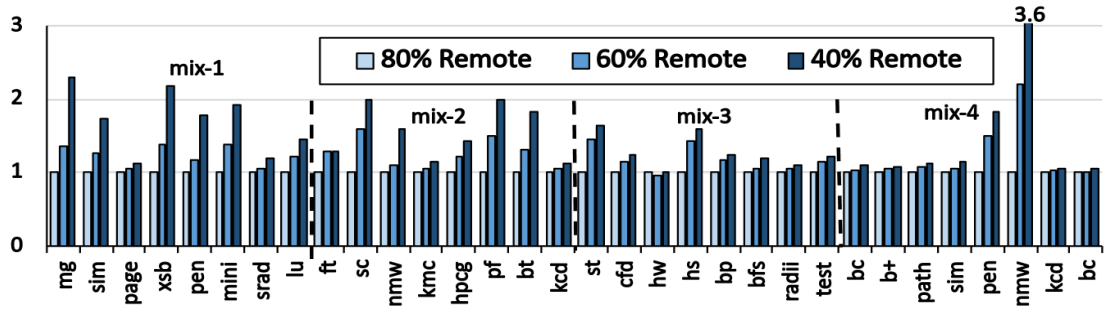


FIGURE 5.15: Impact of Changing Remote Memory Footprint

initial latency parameters. Figure. 5.14 shows the performance of *CosMoS* and *Daemon* compared to the baseline with increased latency. However, the performance gap widens in this case compared to the initial configuration. The average performance improvement with respect to baseline is 84% and 59% for *CosMoS* and *Daemon*, respectively. With increased latency, all the cache line accesses (small packets) are most impacted, and the mechanism with large page accesses (*Daemon*) suffers less impact. However, *CosMoS* still performs better than *Daemon* due to the reason explained before, but the performance improvement is 1.15x, which is lesser than that of the previous case with reduced bandwidth (1.19x) and from initial configuration (1.20x).

Senstivity to Local/Remote Memory Ratio: Lastly, we show the speedup with *CosMoS* on changing the local/remote memory allocation ratio for an 8-node configuration. Figure. 5.15 shows the performance improvement on using 60% and 40% of the remote memory compared to the 80% remote (performance normalized). As expected, the performance is improved for most workloads as more pages can be kept in the local memory at any time. Further, it also reduces the number of pages that must be migrated from remote to local. However, for some workloads, the performance difference is marginal due to lesser memory access or less memory sensitivity of the workloads.

5.5 Summary

The remote AMAT is significantly larger than local DRAM in DMS due to interconnect and network/memory bandwidth sharing in large-scale configurations. In this work, we propose a novel hardware mechanism *CosMoS* that reduces the AMAT in DMS and involves remote memory access with on-the-fly hot-page migration. Our design includes a hybrid hot-page tracker to overcome the limitations in present systems for tracking remote memory accesses. A multi-queue structure for page scheduling keeps the pages with different access frequencies in separate queues. The fine-grained bandwidth partitioning uses a page access splitter logic and an early response mechanism to reduce the delays during a hot page access. We also use a remap table to support on-the-fly page migration by delaying the page-table updates and a few page buffers to temporarily store

the partial page data until the whole page arrives. Finally, we measure the performance through vast experimentation using a variety of workloads on our cycle-accurate simulator. We validate that *CosMoS* significantly improves the performance against existing state-of-the-art mechanisms involving page migration in DMS.

We observe that *CosMoS* improves the performance in all scenarios with up to 20% improvement over *Daemon* and 86% over the baseline DMS. We draw multiple conclusions from the results related to existing and proposed mechanisms. Firstly, the software mechanism for 4KB hot page prediction struggles due to inaccurate predictions due to the overhead involved in memory access tracking. Although compression significantly improves the performance in *Daemon*, the bandwidth partition at large page granularity did not help. Next, the page scheduling in *CosMoS* provides an opportunity to offer fairness to different workloads based on their memory access patterns. Multi-queue scheduling and early access mechanisms improve the response time to page and cache line access. Finally, the fine-grained bandwidth partition between different access queues could efficiently pass the packets from the network queues with minimum delays. The large page access packets in *Daemon* not only take more time but also add long delays to the subsequent packets waiting in the queues. The large packets also occupy more space in the network buffers, while the CPU does not require the whole page in one go and cannot make much use of the early availability of the hot page. Rather, it is better to interleave the responses of multiple critical memory requests like in *CosMoS*.

Although page migration significantly improves performance, the remote memory access traffic must be distributed equally across all the remote memory pools to utilize the available bandwidth properly. The page allocation must be done so that page and cache line accesses face minimum delays in the memory controller queues. In the next chapter, we explore the mechanism to load-balance the memory access traffic and introduce fairness.

Chapter 6

Design and Analysis of Memory Allocation Policies for Disaggregated Memory System

In a pooled memory system, the global memory manager performs remote memory allocation. When the memory manager receives a request from the compute node to allocate a remote memory chunk, it has to select a memory node to serve this request. However, sub-optimal selection can increase the remote memory latency and impact the application performance. In this chapter, we propose remote memory allocation policies for DMS [43] that could significantly improve memory latency compared to the conventional policies. In section 6.1, we introduce about the problem caused by sub-optimal pool selection. Section 6.2 discusses about the conventional and proposed pool allocation policies. We perform experiment analysis in section 6.3 with the proposed and compare it with the conventional allocation policies on various benchmarks with different memory access patterns. We finally conclude the chapter in section 6.4.

6.1 Introduction

DMS relies on a global memory manager to manage remote memory address space and can be hosted on a programmable rack switch. The memory manager performs the task of memory allocation, protection, and address translation for smooth access of remote memory by the compute nodes. The in-network memory managers had been

proposed earlier to manage remote memory connected over the network [28]. However, remote memory management is performed using conventional policies like round-robin or random selection, and no emphasis is given to understanding the impact of memory node selection in a large-scale remote memory system. A similar problem exists in a system with multiple NUMA nodes having their own local DRAM, which always aims to place memory pages in the nearest possible memory [133–135]. However, the distribution of memory pages to different NUMA nodes has little impact due to less NUMA node count. The memory pages can also be allocated to a node with lesser memory traffic to divide the memory traffic and effectively utilize the available bandwidth.

Similarly, In large-scale DMS, we observe that interconnect is not the only source of overheads during remote memory access. The contention in memory queues can also add significant overhead due to uneven memory traffic distribution across memory nodes. This, in turn, also increases the network congestion in some of the network lanes. In this work, we propose a two-phase memory allocation and reservation mechanism for using remote memory in scalable DMS. Different memory allocation policies show a variable impact on the latency performance based on its efficiency in load-balancing the memory requests across different memory nodes. We evaluate the average memory latency and application performance using our simulator in memory trace mode. The main contributions of this work are as follows:

- We study the overheads in remote memory performance in a large-scale DMS and determine the major factors impacting the performance.
- We propose two-phase memory management for DMS-connected compute hosts using local and global memory managers to implement allocation policies.
- We use a variety of benchmarks and workload mixes to evaluate the proposed policies, which show a significant improvement in the AMAT and application performance.

6.2 Memory Allocation Policies

Firstly, the local memory manager at the compute node should decide when to use the remote memory. The first way is to use all available local memory at the start and request the global memory manager once it is finished. This approach will cause a sudden performance slowdown when no more local memory is available. However, some cold pages can be migrated to remote memory in the background to give space for more local pages. Another way is to allocate pages alternatively in local and remote memory. This policy can manage a lower AMAT for a long duration as it gives more scope for moving cold pages to remote. It extends the overall utilization of the local

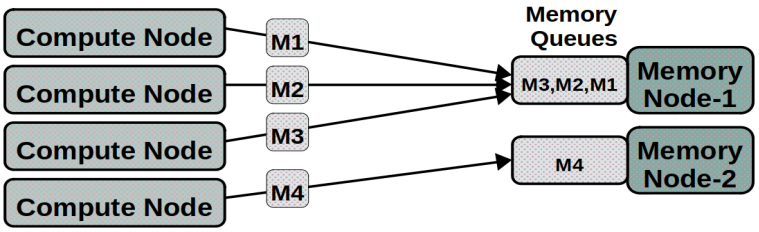


FIGURE 6.1: Contention at remote memory node-1 due to an imbalance in the number of memory accesses across memory nodes

memory but does not get the initial low latency benefit from it. The applications will observe different performance impacts of local/remote allocation based on their memory allocation rate and access patterns.

When the local memory manager chooses to allocate pages from remote memory, it first reserves the remote memory from memory nodes in large chunks, as discussed in section 3.3.1. At this step, the global memory manager must select one of the memory nodes. A good allocation policy will do this allocation in such a way that the memory requests are balanced across all the memory nodes. This will reduce queuing delays at memory nodes and in the network. Without proper load balancing, the memory nodes getting more traffic will observe contention at the memory queues, as illustrated in Fig. 6.1. Any of the memory nodes can face this situation for $1/100^{th}$ of a second, increasing the tail latency significantly. Similarly, some network lanes will observe more traffic and become bottlenecks. The unwanted delays increase the AMAT due to non-optimal memory node selection, significantly impacting the application performance.

6.2.1 Conventional Allocation Policy

We first analyze the memory performance of workload *WL-Mix1* (described in section 6.3) with a random node selection policy. The global memory manager chooses one of the memory nodes randomly for serving the chunk allocation request. The page allocation is done alternatively between local and remote memory. As shown in Fig. 6.2a, the average memory access latency is of the order of microseconds, way beyond what is expected. We observe that congestion at the remote memory queues is significant, which is due to the random node selection. Fig, 6.2b shows the average memory latency (across memory nodes), excluding the network delay. We found that the *2%* of the remote memory accesses suffered access latency of *1000ns* or more and were responsible for larger latency, shown in Fig. 6.2c. This high tail latency can be handled by load-balancing the memory requests across different memory nodes. Fig. 6.2d shows the variation in the number of remote memory accesses sent to different memory nodes periodically after every *1.5* million simulation cycles. The variation is calculated by subtracting the count for the memory node with maximum and minimum memory requests during each

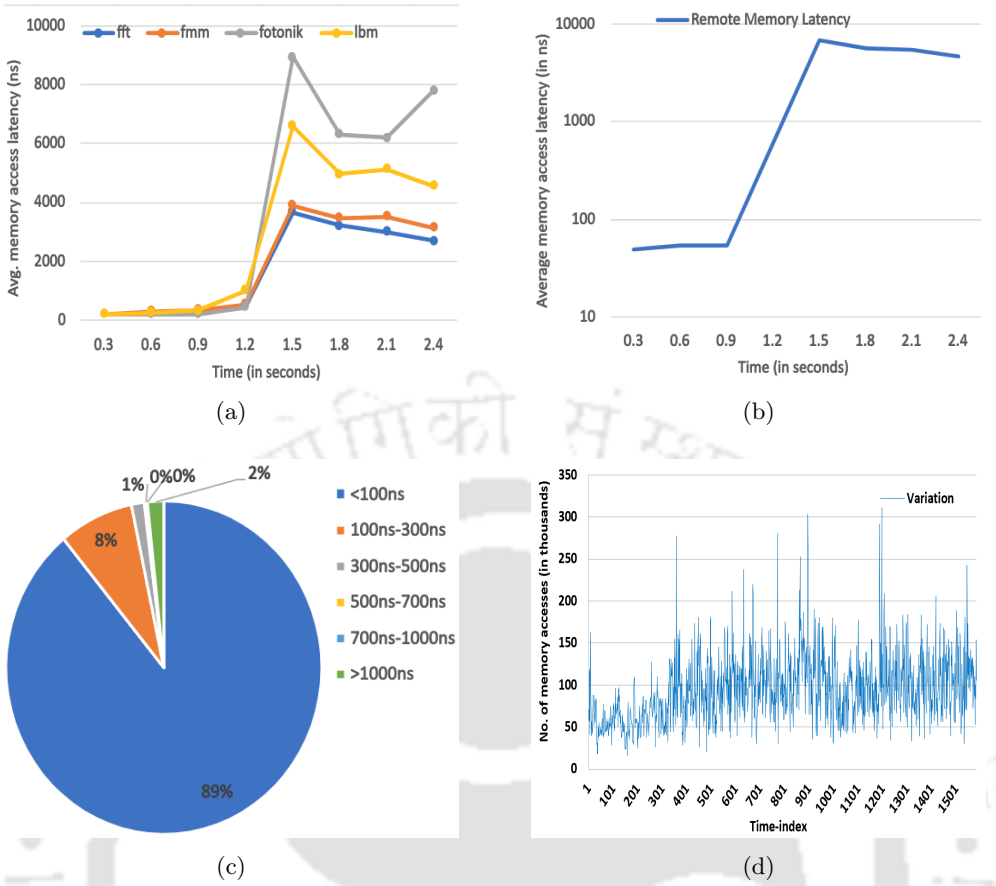


FIGURE 6.2: Random selection with alternate local-remote page allocation (a) Average memory access latency (b) Remote memory access latency (c) memory access latency distribution (d) Access variation in memory nodes

sampled period. We observe that the major reason behind high tail latency is due to the imbalance of memory accesses across the memory nodes.

Another conventional policy is to select the memory node in a round-robin fashion. Although round-robin selection performs better than a random selection, the chances of memory contention still exist. The compute nodes generate memory requests at different rates and the round-robin selection cannot ensure equal distribution of memory requests in every time period to all the memory nodes.

Next, we discuss the proposed pool selection policies for the scalable DMS that distribute the requests across the pools and reduce the tail latency.

6.2.2 Smart-idle Selection

The Smart-idle policy selects the most idle memory node to avoid sudden bursts of memory requests. The first step selects a small subset of memory nodes based on its recent request count. The memory nodes with the least traffic are less likely to face

Algorithm 1 Smart-Idle memory node selection

Input: Memory Access Count of last 4 windows

Notations:
$M_{aC}(i)$: Normalized Memory access count in window(i)
Af_{pool}: Access factor for a memory node
Step 1
for Pool = 1 to n **do**
    for each window i = 1 → 4 **do**
        $M_{aC}(i) = \frac{MemAccCount_{win(i)}}{i}$
    end for
    $Af_{pool} = \sum_{i=1}^{4} M_{aC}(i)$
end for
Step 2
Sort 'n' nodes in increasing order of their Af_{pool}
Select top 'm' nodes from sorted list,where
m = Ceil[log2(n)]
$Pool_{id}$ = Memory Node with least utilised memory
return $Pool_{id}$

contention and can be selected for new memory allocation. The memory controller with the global memory manager hosted on the rack switch keeps track of memory accesses to each memory node. It uses a few 64-bit counters per node to keep track of the memory access count during the most recent periods. The *window-based* mechanism is used to determine an access factor (Af) for each node. A *window* is the duration between two consecutive remote memory allocation requests to a memory node. The (Af) represents the degree of memory request traffic at each memory node. Memory requests are traced and counters are maintained for recent windows to determine the memory node's activity. The Af is calculated by adding the normalized memory access count (M_{aC}) of the last 4 windows. The M_{aC} is the total memory access counts *MemAccCount* in a window divided by its window number, where *window-1* is the most recent. This gives more weightage to recent memory accesses than those in the older windows. The complete process is discussed in algorithm 1. A lower value of Af indicates that a node has faced less memory traffic recently and can be selected for the next chunk allocation.

Directly selecting a memory node with the lowest access factor (Af) will result in an unequal amount of memory allocation across different memory nodes. Therefore, step two of the smart-idle policy chooses a lesser active node while also maintaining an even distribution of memory chunks across nodes. Assuming the total number of memory nodes to be n, the smart-idle policy will first select a set of m nodes, where m is calculated as m = Ceil[log2(n)]. The small set of nodes is selected by sorting *Afs* in

ascending order and selecting the top m nodes. The final node selection is made from this set having the least allocated memory. This condition ensures that even if a memory node recently faces fewer memory requests, a sudden burst of memory requests does not arrive from the memory allocated in the past. The choice of pool with the least allocated memory is less likely to face such sudden bursts.

6.2.3 Uniform Load Partitioning

In the next policy, the requests are uniformly distributed across the memory nodes based on certain heuristics. The request rate for each compute node is monitored and is defined by the number of remote memory requests per million cycles. It maps a remote memory node to each compute node so that all nodes face similar traffic. The memory request rate is monitored once at the start of each epoch, and the compute-to-memory node mapping is kept the same during the epoch. This mapping problem is similar to a *Set-Partition*, where the goal is to create small non-empty subsets of numbers from a large set such that each subset has the same sum. Practically, it may not be possible to partition larger sets into subsets with the same sum. So, we follow an approach where the subsets have approximately equal sums. A greedy approach is used to find a local optimal solution at each step, creating small subsets of compute nodes such that their total request rate is approximately the same. Each of these subsets is tagged to one of the memory nodes, and the compute nodes in this subset are always allocated memory chunks from the same node until the next epoch. The whole process is given in Algorithm 2.

Algorithm 2 Uniform load partition

Input: Set 'Z' of 'n' nodes with its memory request rate

Output: 'm' subsets of nodes, where 'm' is number of memory nodes

1: **Notations:**
2: min_set(setA, setB, ...setN): Set having minimum sum of elements
3: Sort set 'Z' of nodes in descending order of their memory request-rate
4: **while** set $'Z'$ has an element **do**
5: **for each** subset Y_i, $i = 1 \rightarrow m$ **do**
6: $Sum(Y_i) = \sum_{node=1}^{sizeof(Y_i)} mem\_req\_rate(node)$
7: **end for**
8: $Y_k = min\_set(Y_1, Y_2 ... Y_m)$
9: Add 0^{th} element of $'Z'$ to Y_k
10: remove 0^{th} element from $'Z'$
11: **end while**

TABLE 6.1: Benchmarks

Benchmark Name	Cache Miss-Rate	RAM Accesses (in Millions)	Footprint (in GB)	Label
lbm_s	12.49%	45.47	2.7	
fotonik3d_s	5.96%	11.92	0.57	*WL-Mix1*
fft	4.16%	15.81	1.06	
fmm	3.42%	12.5	3.20	
Lulesh	9.05%	5.2	0.31	
XSBench	2.01%	9.76	0.52	
miniFE	14.29%	15.7	0.92	
stream	2.95%	13.48	0.48	*WL-Mix2*
NAS:mg	14.91%	8.11	0.25	
NAS:ft	4.15%	19.53	1.13	
NAS:sp	1.16%	5.51	0.45	
NAS:bt	6.15%	5.27	0.43	

TABLE 6.2: Simulation Parameters

Element	Parameter
CPU	1.2GHz, 8-core
ITLB	128 Entries, 8-Way ITLB, 60-cycle latency
DTLB	64-Entries, 4-Way DTLB, 60-cycle latency
L1-Cache	32KB(I/D), 8-Way assoc, 4-cycle latency, 64B
L2-Cache	256KB, 4-Way assoc, 12-cycle latency, 64B
L3-Cache	2MB per core, 16-Way assoc, 41-cycle latency, 64B
Cache Type	Write-Back/Write-Allocate, Round-Robin
Memory	256MB Per Compute node, 32GB per Memory pool
Switch	400Gbps, 132MB Buffer, 20ns delay
NIC	100Gbps, 10ns Delay
Packet-Size	64B request, 128B response, 25ns Packet-Prep

6.3 Experimentation Methodology and Results

We use a trace-based simulation mode to rapidly simulate the memory accesses from multiple compute and memory nodes in a large-scale DMS. We selected four multi-threaded workloads with large memory footprints and high cache miss-rates from *Spec2017* [136] and *Splash-3x* [111]. This workload is shown as *WL-Mix1* in Table 6.1. We collected the traces for 200M instructions, and benchmarks significantly vary in the number of memory accesses made during the simulation. Further, we use another workload mix *WL-Mix2*, using traces of HPC mini-apps such as Lulesh [117], miniFE [118], XSBench

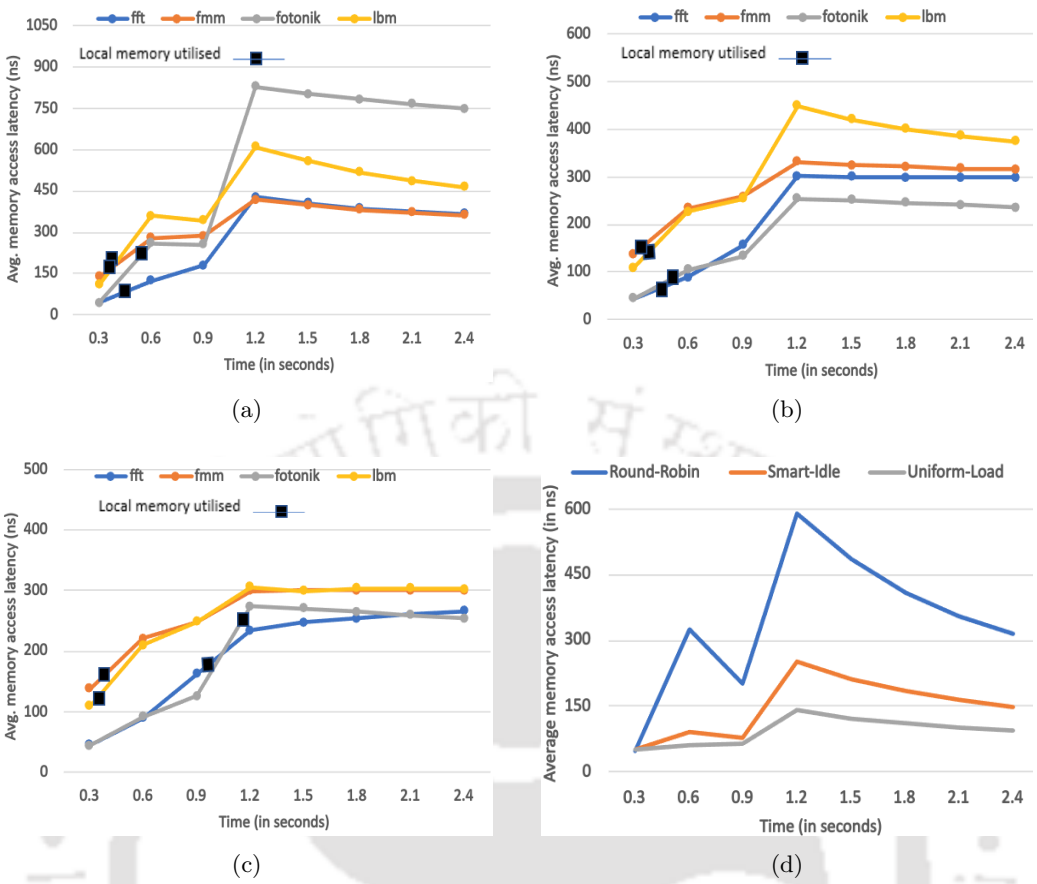


FIGURE 6.3: Average memory access latency with *Local-First* allocation over workload *WL-Mix1* (a) Round-Robin selection (b) Smart-Idle selection (c) Uniform load partition (d) Average remote memory latency

[121], and applications from NASA parallel benchmark suite [114]. Both the workload mixes are simulated in an environment similar to a rack-scale DMS with 16 compute nodes with 256MB local memory and 6 memory nodes. Each workload is deployed on 4 nodes for WL-mix1 and 2 nodes for WL-mix2. The performance is measured by taking the average over all the nodes running the same benchmark. We deliberately kept the number of memory pools on the lower side to test the memory/network bandwidth limits. In Table 6.2, we summarize the system parameters used for the simulations.

6.3.1 Impact on Memory Latency

We experimented with page allocation policies at local-memory manager: *local-first* and *alternate local-remote* and implemented a cold page migration to push the least recently accessed pages to remote memory. In conjunction with the local page allocation, we run different memory node selection policies: Round-Robin, Smart-idle, and Uniform load.

We start our discussion with *local-first* page allocation and round-robin memory node selection. Fig. 6.3a shows the average memory access latency at different simulation points, where the small black marks on the plot represent the time at which local memory is completely used up. Although the cold page eviction still works in the background, the local memory is not always available for a page allocation. The results show a substantial decrease in memory access latency for each benchmark compared to the random pool selection, it is still high for workloads like *lbm* and *fotonik*. This is because both these benchmarks have a high density of memory requests to the remote memory, and memory access latency is high enough to impact the application speed significantly. It is interesting to note that even though *lbm* generates more remote accesses than *fotonik*, the memory access latency in *fotonik* is very high compared to *lbm*. Due to a major burst of remote memory accesses during epoch 4, memory latency suddenly rose, but it is still lower for *lbm* compared to *fotonik*, as the contention is lesser.

On the other hand, the smart-idle policy significantly improves the average memory access latency compared to the round-robin policy for all the workloads, as shown in Fig. 6.3b. With local-first allocation, latency only increased gradually for all benchmarks due to load balancing. Fig. 6.3c shows that the uniform load partition policy further reduces the memory latency (compared to the smart-idle policy). Uniform load partitioning works well even during epoch4, where the other policies suffer from contention due to a sudden burst. With any workload, the average memory latency was below *300ns*, which is a big improvement compared to the other two policies. Fig. 6.3d shows the cumulative average memory access latency for all the memory nodes (without including network delays) for round-robin, smart-idle, and uniform load policies. In epoch4 and epoch5, the round-robin could not handle the sudden burst of requests across the memory nodes, and there was a large spike in the average memory latency. However, with smart-idle allocation and uniform load partition, chunk allocation was largely balanced, and the memory requests were distributed equally across all the pools.

Next, we study the performance of the memory manager at the node with *alternate local-remote* page allocation. Fig. 6.4 shows the results, where we see no sudden burst of memory accesses, and there is a gradual increase in memory access latency after a point when the local memory is exhausted. All three memory node selection policies perform well compared to that with the *local first* page allocation. Even though *lbm* and *fmm* have a large memory footprint, we still see low average memory latency. We observe that uniform load partition performs better among all, followed by smart idle and round-robin policies. Fig. 6.4c shows that the best results are obtained with uniform load partition pool selection combined with *alternate local-remote* page allocation.

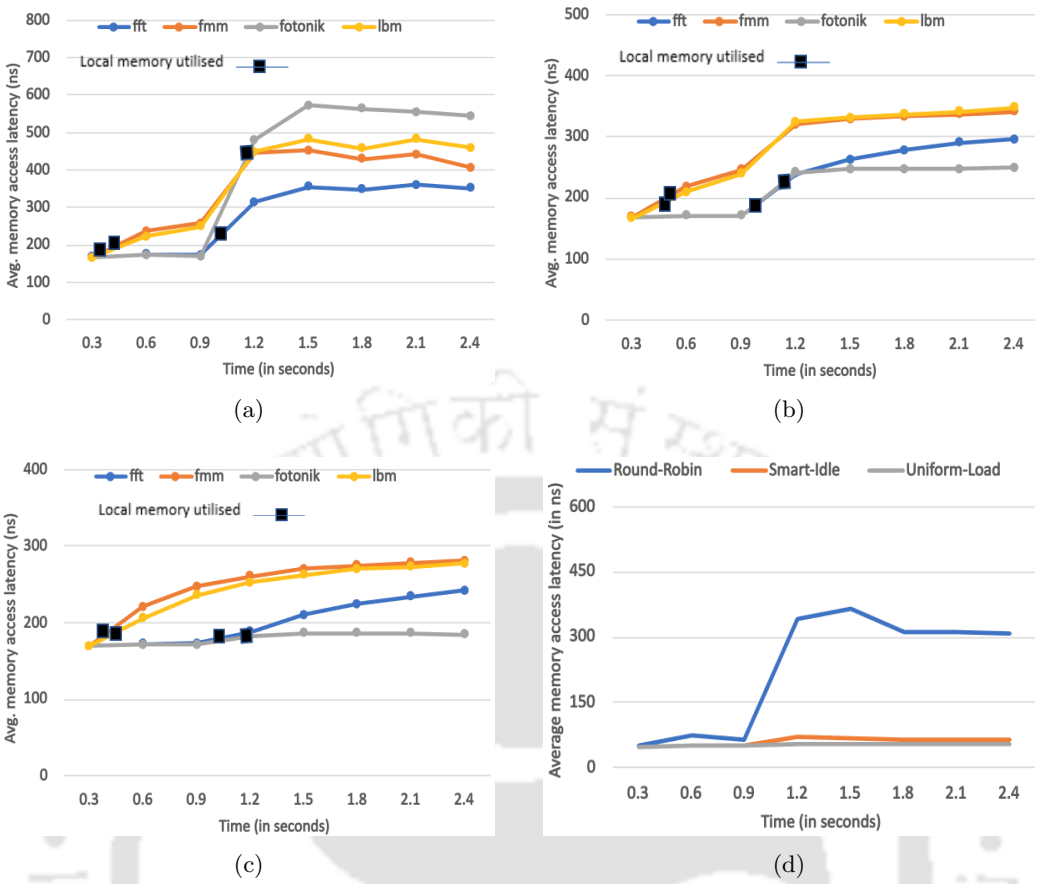


FIGURE 6.4: Average memory access latency with *Alternate Local-Remote* allocation over workload WL-Mix1 (a) Round-Robin selection (b) Smart-Idle pool selection (c) Uniform load partition (d) Average remote memory latency

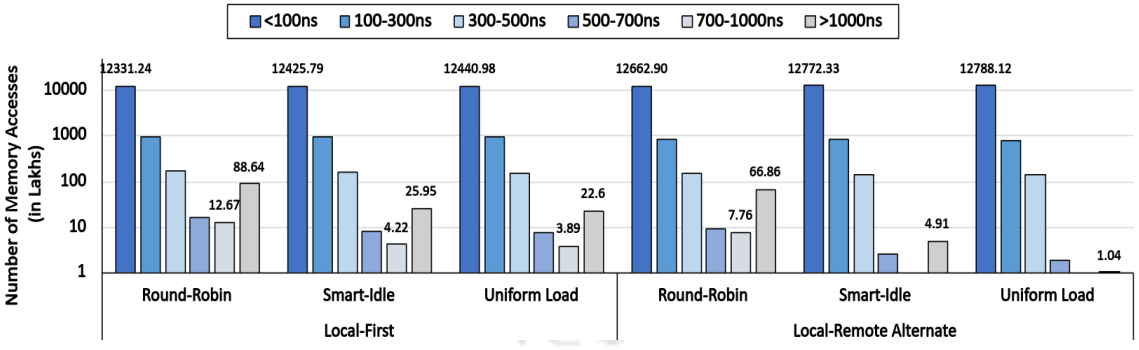


FIGURE 6.5: Distribution of remote memory accesses based on access latency

6.3.2 Impact on Tail Latency

We further analyze the memory access completion time for all the remote memory requests, shown in Fig. 6.5. The latency only includes the time taken at remote memory pools, not the time spent in the network. Different colored bars in the graph represent the number of memory accesses completed for each category based on its memory access latency. We earlier saw the huge tail latency with random selection. Here, we can see

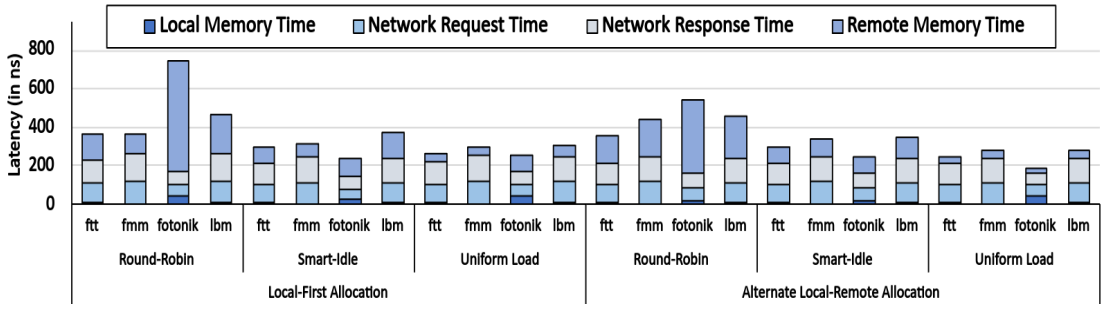


FIGURE 6.6: Local/Remote/Network Latency breakdown

that round-robin selection had significantly brought it down but could not mitigate its impact completely. Many memory requests are still completed with more than *500ns*. The reason is that although round-robin seems to be the default choice for the simple task of memory node selection, it does not distribute the memory traffic optimally across different memory nodes. It is where the uniform load partition and smart-idle selection come into the picture. Both of them combined with local-remote alternate page allocation is better than a simple round-robin and could reduce the tail latency to a reasonable extent. The graph shows that only a few percent of memory accesses take more than *500ns* to complete, keeping the average memory access time to a reasonable limit. The workloads benefit from the optimum selection with the proposed policies, whereas the alternate local-remote allocation exploits the memory access pattern in *fotonic* and *fft*. It allows these workloads to utilize local memory for a longer time, decreasing the total memory traffic on the network and contention on remote memory pools.

6.3.3 Overall Latency Breakdown

We observe the overall latency breakdown for all the memory accesses of a workload in local or remote memory during the run. Both uniform load partition and smart-idle suffer lesser network delays than round-robin as memory request packets also get distributed equally across all the links connecting to the different memory pools. With round-robin, due to some instances when a particular link sees more packets, it suffers more network congestion and higher delays relatively. However, we saw a big variation in average remote memory access time for all the benchmarks through different policies, which is also our motivation behind exploring these policies. The complete results are shown in Fig. 6.6.

6.3.4 Performance Slowdown

This section shows the performance slowdown for each workload in the above experimentation compared to a system with 100 percent local memory. We apply a straightforward

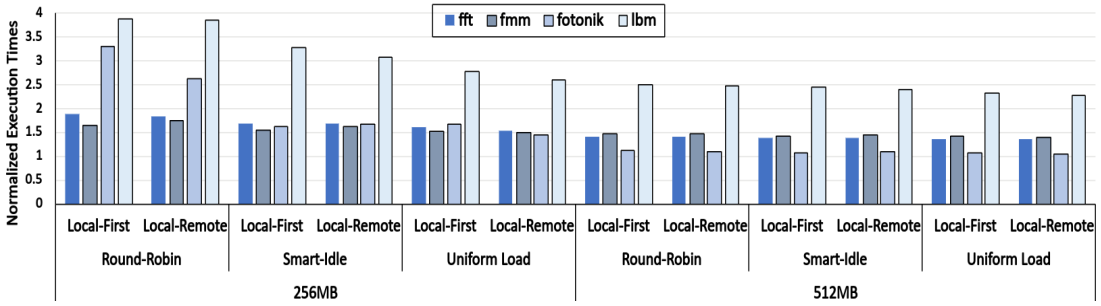


FIGURE 6.7: Execution times normalized against entirely local memory (as 1) vs. DMS with two different scenarios (256MB and 512MB of local memory at each compute node)

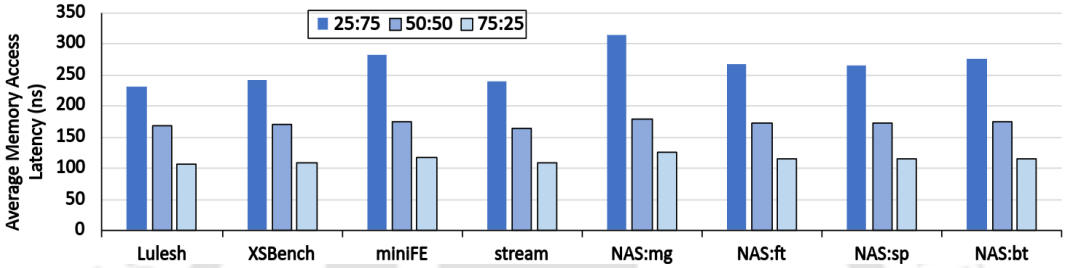


FIGURE 6.8: Average memory latency for HPC workloads and mini-apps with different proportions of local and remote memory using alternate local-remote and uniform-load partition

way to determine the performance slowdown by tracing total simulated instructions, the number of instructions of each type, assuming a fixed number of cycles for each instruction type, the number of last-level cache misses, and the average main memory latency. Eventually, the total number of main memory accesses remains the same, and the average memory latency is only the differentiating factor between an entirely local memory system and a DMS. In the case of 100 percent local memory, all the memory accesses are completed on average in *45ns*. In contrast, it faces significantly high latency in DMS, as shown in Fig. 6.7. The figure shows the performance difference with all the policies over the DMS and entirely local memory. Two disaggregated memory scenarios are taken, one with 256MB of local memory and another with 512MB of local memory. The performance is shown as normalized execution time where '1' is the execution time for 100 percent local memory.

6.3.5 Impact on HPC workloads

We further analyze with workload set WL-Mix2 to show the impact on average memory latency when the amount of local and remote memory is configured with a certain ratio. We experiment with 3 different configurations, 25:75, 50:50, and 75:25 memory for local and remote, respectively. The memory manager is configured to allocate a new page in the particular memory unit alternatively as per the given ratio. Fig. 6.8 shows the average memory access times of all the 8-workloads with the uniform load partition

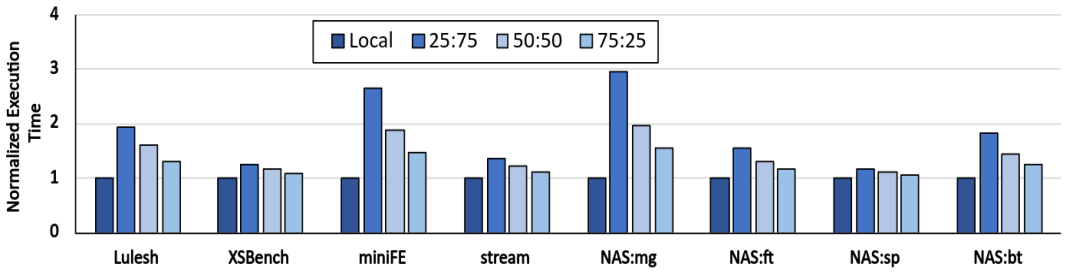


FIGURE 6.9: Normalized execution time for HPC workloads and mini-apps with different proportions of local and remote memory using alternate local-remote and uniform-load partition

policy, which eventually performed best out of the three policies with WL-mix1. Further, we show the normalized execution times for all the workloads compared to local only memory system in Fig. 6.9. The workloads with more last-level cache miss rates, such as miniFE and NAS:mg, suffer more as more memory pages are in the remote memory. At 50% local memory, the average increase in execution times across all 8 workloads is 1.83x compared to complete local memory, which drops down to 1.4x for 6 workloads, excluding miniFE and NAS:mg.

6.3.6 Complexity Analysis and Performance Impact

Our memory pool allocation policies use heuristics and try to determine a memory pool for a remote memory chunk allocation. Both smart idle and uniform load policies rely on some counter variable that gets updated during the remote memory accesses by different compute nodes. These counter variables are then used after every small interval to get an optimal memory pool that assumingly will provide the most bandwidth for future memory accesses from different compute nodes. Once the counter values are available after an interval, memory pool mapping is straightforward and does not have much complexity in its hardware implementation, as it is straight forward calculation using the counter values, which can be performed at switches.

Further, as per our experimentation, uniform load partition performs close to optimal, whereas the smart-idle does not always returns a good allocation candidate. However, uniform-load also uses more number of counters compared to smart-idle, which increases the hardware overhead by a small margin. The number of counters in smart-idle is equal to the number of memory nodes, which is equal to the number of compute nodes for uniform-load. In a real system, there will always be lesser number of memory nodes compared to compute nodes. This extra overhead also makes uniform-load more accurate by generating an optimal compute node to memory pool mapping.

6.4 Summary

This work focuses on local and remote memory allocation policies for rack-scale DMS. We show that high contention at the queues of remote memory nodes becomes a bottleneck when traditional policies (like round-robin or random) are used. We proposed two policies, uniform load partition and smart-idle selection, to evenly distribute the memory access traffic among all the memory pools and counter the high tail latency. We run our experiments on a trace-based simulator and the results show that the proposed mechanism provides improved average memory latency across local and remote memory. Compared to the conventional round-robin, the average memory latency reduced 3x with uniform load partitioning and 2x with smart idle allocation on average over multiple HPC workloads. Further, the workload execution times were reduced 33% and 20% on average with uniform-load partition and smart idle allocation, respectively, compared to the round-robin allocation.

In this work, we deal with the fairness of improving bandwidth utilization in the memory pools/nodes. However, different compute nodes run various workloads and may need to be treated with different priorities for introducing QoS. In the next chapter, we explore the role of scheduling remote memory requests for QoS in a large-scale DMS.

Chapter 7

Understanding the Performance Impact of Queue-Based Resource Allocation in Scalable Disaggregated Memory Systems

In DMS, accessing on-network memory resources incurs additional access costs and significantly impacts the performance, which is further impacted by multiple compute nodes sharing the same memory nodes. The performance can be improved if fairness is ensured for compute nodes running workloads during network and remote memory queue allocation. In this chapter, we explore the opportunities to improve fairness in the performance of all the nodes in multi-node DMS using different queue allocation methods for network and memory bandwidth partition [44]. We propose using an in-network global memory controller to control the flow of memory requests to the remote memory nodes to enforce fairness and Quality-of-Service (QoS). We utilize the memory request rate of each compute node to set their weights/priorities for different queue allocation methods. We introduce the requirements of different workloads and how they can be fulfilled in section 7.1. Section 7.2 discusses the background and motivation, and we explain our design in section 7.3. We present our experiment analysis in section 7.4 and conclude the chapter in section 7.5.

We evaluate all the scheduling policies using a trace-based disaggregated memory simulator over various benchmarks with different access patterns. Our results show that

scheduling policies significantly impact the average memory latency and the system performance in different configurations.

7.1 Introduction

In hardware memory disaggregation, the onboard local memory is replaced by remote memory nodes connected through high-speed networks to compute nodes supporting low latency and high-bandwidth coherent access. The server nodes have a small local memory, and the application's requirements are mostly fulfilled through remote memory. Therefore, these systems require optimizations to hide memory latency and improve the overall memory access cost. Further, DMS are to be run in a scalable environment, and the network will be congested due to multiple compute nodes trying to access remote memory simultaneously. As discussed previously, the memory nodes may also introduce long access delays due to contention in their memory queues. Due to such delays, the compute nodes face significant performance slowdowns. The memory request pattern at compute nodes varies as they run different workloads and require differential treatment based on the criticality of memory accesses and the intensity of the access traffic.

This work explores the problem of queue allocation for partitioning the network/memory bandwidth among the memory requests of different compute nodes. Firstly, we propose an in-network hardware mechanism to control the flow of requests to the external memory pools. Secondly, we explore the impact of commonly used queue allocation practices for maintaining the QoS. Finally, the queue allocation methods require setting weights and priorities for queues belonging to different compute nodes. So, we propose a simple mechanism that uses a memory request rate as the criteria for setting the weights and priorities. We evaluate our design on our custom-built trace-based disaggregated memory simulator with multiple compute and memory nodes. Our results show a significant performance impact of different request scheduling policies in a scalable DMS.

7.2 Background and Motivation

Fig. 7.1 shows the baseline DMS where the compute nodes forward LLC through a remote memory controller to the remote memory nodes to a central interconnect, such as a global memory controller. The global memory controller is responsible for scheduling incoming memory requests from all the compute nodes to their respective memory pools. In a DMS, groups of compute and memory nodes are deployed together in a certain ratio. However, the number of memory nodes will be lesser than the compute nodes while the compute nodes use these memory nodes simultaneously. It is expected that additional delays will be introduced due to network and contention at memory queues.

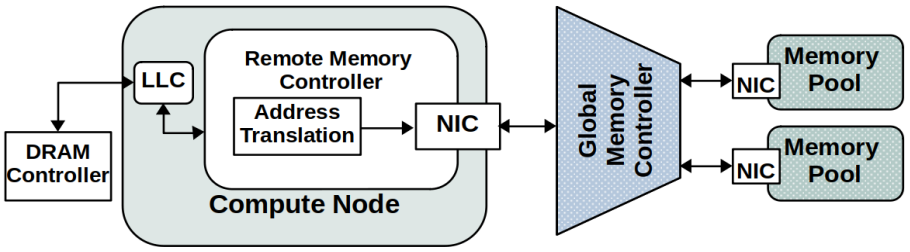


FIGURE 7.1: Overview of Global Memory Controller

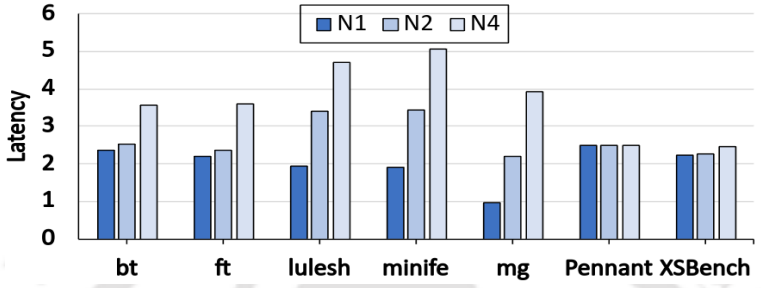


FIGURE 7.2: Increase in Memory latency on changing the nodes sharing same memory node

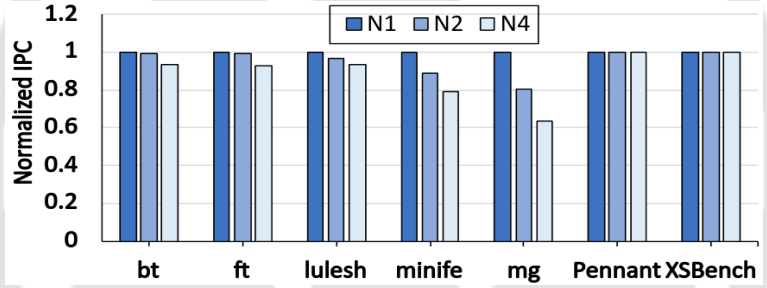


FIGURE 7.3: Drop in IPC on changing the nodes sharing the same memory node

However, the slowdowns can be eliminated by extra hardware logic at the global memory controller with multiple queues and fair allocation of network and memory resources to each compute node. The global memory controller can be implemented using an FPGA that will work at a switch to forward the memory requests.

To understand the impact of congestion and contention, we initially evaluate the memory latency and system performance using a single first-in-first-out (FIFO) queue at the global memory controller. Fig. 7.2 shows the increase in memory latency in a DMS with a local-to-remote memory ratio of 40:60 compared to an entirely local memory system. The memory latency rapidly shoots up for all the workloads when the number of nodes sharing the same memory pool increases (except for Pennant and XSBench, which have lesser memory access and are CPU-bound processes). The increase in latency is due to more network and memory traffic. Although we use the same workloads on all nodes, the scheduling of requests will play an important role when all the nodes run different workloads. The nodes with high memory request rates can unevenly grab the

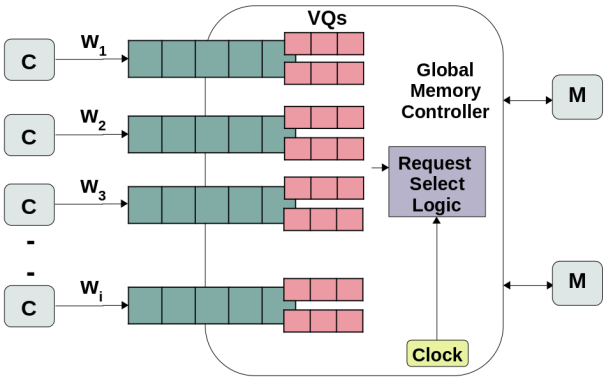


FIGURE 7.4: Multiple queues at Global Memory Controller

queues and starve the compute-bound workloads for their memory access. Fig.7.3 shows the drop in IPC due to multiple nodes sharing the same memory pool and following the same trend.

7.3 Design

In this section, we present the design of our global memory controller for implementing different queue allocation policies. The compute nodes have different memory access patterns and significant variations in their memory request rate as each node runs a different workload. We implemented multiple queuing mechanisms to enforce proper scheduling of memory requests, where each mechanism uses multiple queues at the global memory controller and has virtual queues associated with each queue. Some queue allocation mechanism requires additional information to ensure QoS and fairness among the memory requests of different nodes.

7.3.1 Weighted Round-Robin Scheduling

First, we implement a weighted round-robin scheduling, where each compute node has a separate queue for its memory requests at the global memory controller, as shown in Fig. 7.4. Each queue is assigned a weight based on the memory request rate of the node associated with that queue. As all the memory requests pass through the global memory controller, they can easily be tracked for each node, and the request rate can be calculated by dividing the total number of cycles and the number of memory requests during that period. Finally, weights are calculated by dividing the memory request rate of each node and the total request rate of all the nodes, as shown in eq. 7.1. The weights are then normalized on a scale of 100 that determines the granularity of selecting requests from the queue. Out of every 100 cycles, the queue of node 'i' will be given chances for W_i number of cycles. However, if the queue is empty, then the request from the next

queue can be taken in the same cycle. Further, we use a simple round-robin arbitrator to select between the virtual queues of the selected node's queue. The weights are reset after every epoch, and all the queues use the same weights during the initialization.

$$W_i = \frac{Req\_Rate_{node(i)}}{Total\_Req\_Rate} \tag{7.1}$$

$$W_i = ceil[100 \times W_i] \tag{7.2}$$

7.3.2 Round-Robin Scheduling

The working of round-robin scheduling is similar to the weighted round-robin, with the only difference being that the weights of all the queues are the same. Due to this, all the memory request queues get equal chances for packet selection. A two-level arbitrator picks a memory request packet from one of the queues. The first-level arbitrator selects one of the input queues, and the second-level arbitrator picks between the virtual queues of the selected input queue. If the queue has no ready packet, the next queue in the line gets the chance.

7.3.3 Priority Scheduling

Each request queue is allocated a specific priority level based on its memory request rate in this scheduling mechanism. We use four priority levels to allocate priorities to each node's queue. In each cycle, the request selector starts from the higher priority queue and moves towards the lower priority queues if any queue does not have ready packets. In a scalable DMS, there can be more compute nodes than the number of priority levels. So, we follow a first-in-first-out selection mechanism if multiple request queues have the same priority. However, correctly allocating the priority levels to each request queue is essential to enforce QoS. In a public data center, priorities can be allocated based on service-level agreements. However, it is not possible in many cases, where priorities should be dynamically assigned to request queues.

Each compute node has a different sensitivity to the memory latency as they run different workloads. The CPU-bound process often has fewer memory references and is more latency-sensitive. Whereas the memory-bound processes frequently access the memory and are lesser latency-sensitive. To ensure QoS, the memory-sensitive processes should be allocated a higher priority, and CPU-sensitive processes can be allocated a lower priority. To allocate priorities to request queues, we categorize all the nodes into four categories based on the mean of memory request rates over all the nodes. As shown in eq. 7.3, we first find the overall mean (μ_{all}) to create two subsets of the nodes, subset S_1, with each node having a request rate lower than the mean, and subset S_2 where

each node's request rate is higher than the mean. A lower mean (μ_{lower}) is calculated (in eq. 7.4) for the nodes in S_1 to further divide into two categories (say S_{1a} and S_{1b}) in the same way. A similar thing is done with the subset S_2 by calculating upper mean μ_{upper}), in eq. 7.5 and dividing into S_{2a} and S_{2b}. Finally, priorities are allocated in decreasing order to S_{1a}, S_{1b}, S_{2a} and S_{2b}. In this way, the memory request of nodes with lesser request rates is scheduled before the memory requests of nodes with higher request rates. The requests selector will only move to the lower-priority queue if the high-priority queue is empty. Finally, we adjust the priorities for changing memory request rates of the nodes by resetting it after every epoch. All the queues are kept at the same priority during the initialization process.

$$\mu_{all} = \frac{1}{n} \sum_{i=1}^{n} RR(i) \tag{7.3}$$

$$\mu_{lower} = \frac{1}{k} \sum_{i=1}^{k} RR(i), \qquad \{RR(i) > \mu_{all}\} \tag{7.4}$$

$$\mu_{upper} = \frac{1}{j} \sum_{i=1}^{j} RR(i), \qquad \{RR(i) < \mu_{all}\} \tag{7.5}$$

7.3.4 Priority-based Weighted Scheduling

Priority-based weighted scheduling (PBWS) combines weighted round-robin scheduling and the priority scheduling mechanisms [137]. As pointed out earlier, compute-intensive workloads with low memory request rates are more sensitive to delays and require faster response. These nodes should be prioritized over memory-intensive nodes with high memory request rates. However, rather than using multiple priority queues and mapping each request queue to one of them, PBWS uses only a single queue for assisting the high-priority requests. The other request queues are associated with weights based on the memory request rates of the nodes and are calculated using eq.7.1 and eq.7.2, as mentioned earlier. The requests in the highest priority queue are served first, prioritizing the compute-intensive nodes over the others. It is only when the highest priority queue is empty then the queues of the other nodes are serviced. An arbitrator-based logic similar to the weighted round-robin algorithm selects the queue to be serviced. In our experimentation, we reserve the priority queue only for one node with the lowest memory request rate.

TABLE 7.1: Simulation Parameters

CPU	3.6GHz, 8-core
L1 Cache	2-Cyc, 32KB(I/D), 8-Way
L2 Cache	20-Cyc, 256KB, 4-Way
L3 Cache	40-Cyc, 2MB per core shared, 16-Way
Cache Type	Write-Back/Write-Allocate, Round-Robin
Memory (Local/Remote)	1200x2MHz DDR4 DRAM (19.2GB/s)
Switch	400Gbps, 4MB buffer per port 5ns processing/switching delay
Network Interface (Nodes)	40/100Gbps, 1MB packet buffer 10ns (de)packetization/processing
Packet-Size	64B request, 128B response

TABLE 7.2: Benchmarks

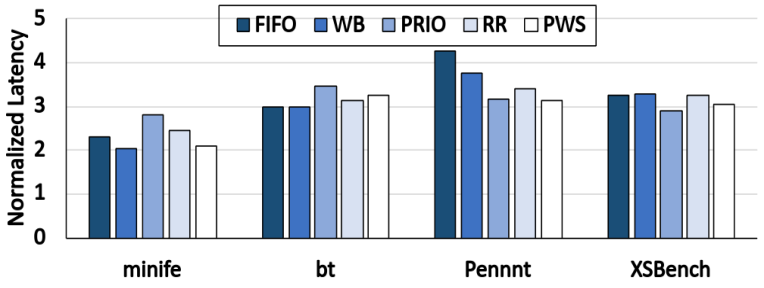
Application	**Memory Accesses** (in Millions)	**Footprint** (in MBs)
NAS:bt [114]	19.98	128
NAS:ft [114]	21.23	132
NAS:mg [114]	74.48	460
lulesh [117]	13.23	88
minife [118]	34.88	216
Pennant [119]	0.71	10
XSBench [121]	9.6	152

TABLE 7.3: Workload Mixes

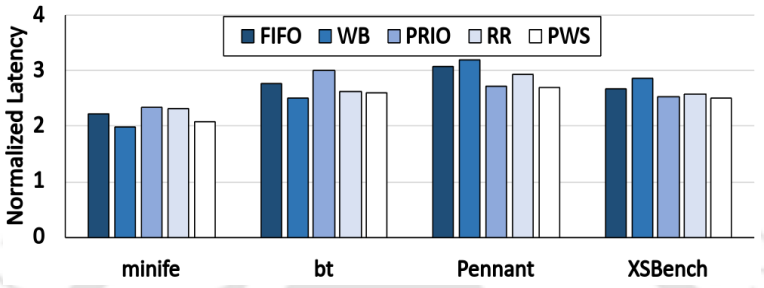
Mix-4A	minife - NAS:bt - Pennant - XSBench
Mix-4B	NAS:mg - NAS:ft - lulesh - Pennant
Mix-8	XSBench - bt - ft - lulesh - mg - minife - Pennant - XSBench

7.4 Methodology and Results

We use a trace-based disaggregated memory simulator to evaluate the proposed designs. Table 7.1 shows all the simulation parameters for the evaluation. The memory allocation to local and remote memory was performed using fixed ratios (such as 40:60) in different experiments using an alternative round-robin allocation for page allocation following the same ratio. If multiple remote memory nodes exist, we again follow the round-robin for memory node selection while allocating remote memory chunks to the compute node.



(a)

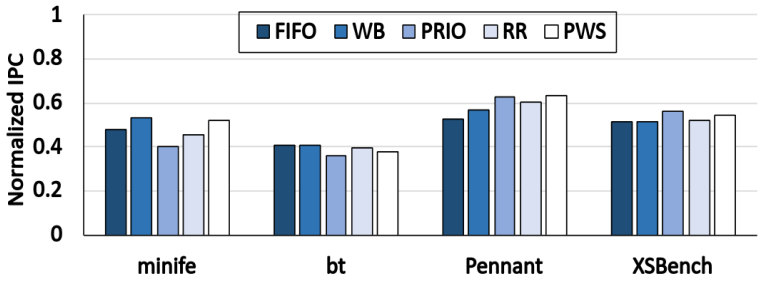


(b)

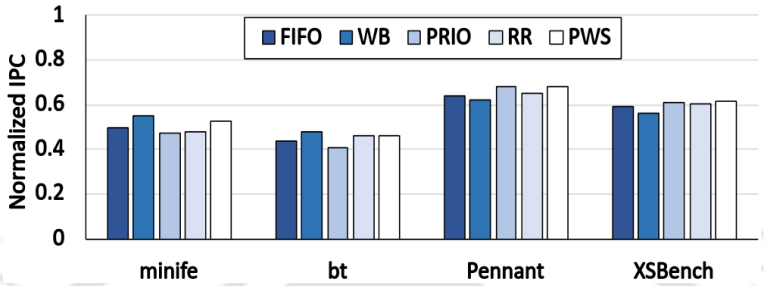
FIGURE 7.5: Normalized memory latency for Mix-4A with 60% of workloads footprint at remote to a system with entirely local memory (a) 1-shared memory node (b) 2-shared memory nodes

Further, we reset the weights or the priorities of request queues after completing an epoch of 100M cycles. The memory accesses are recorded for 1M cycles to calculate memory request rates. The packet sizes in the queue scheduling do not create any difference as the request packets are always 64 bytes (except for a few write-backs, which are limited due to write-back/write-allocate cache) and only consist of memory access addresses apart from the header information. At the same time, a response packet is always 128 bytes, as it also includes the cache line of data. Table 7.2 shows all the benchmarks with their memory access count and footprint during the simulation (100M cycles) and is a good mix of memory-bound and compute-bound workloads. We create workload mixes running on multiple nodes, each running one of a workload, shown in 7.3. We run different Node-to-Memory configurations to show the impact of request scheduling with all the policies.

We start over evaluation with Mix-4A with two nodes running compute-bound workloads (Pennant, XSBench) and the other two running medium to heavy memory-bound workloads (NAS:bt, minife). The abbreviations FIFO, WB, PRIO, RR, and PWS in graphs represent first-in-first-out, weighted-buffer round-robin, priority allocation, round-robin, and priority-based weighted scheduling, respectively. Fig. 7.5a shows the increased memory cost normalized to a local-only system. All four nodes only use a single memory node and memory allocation is performed at a 40:60 ratio, local and remote. Firstly,



(a)



(b)

FIGURE 7.6: Normalized IPC on workload Mix-4A with 60% of workloads footprint at remote to a system with entirely local memory (a) 1-shared memory node (b) 2-shared memory nodes

all the nodes face a significant increase in memory access latency as most of the workload's footprint is in remote memory. Secondly, we observe variations in memory latency on changing the scheduling latency for a good reason. Pennant and XSBench are less memory-intensive and benefit from the higher priority allocated with PRIO and PWS scheduling. On the other hand, NAS:bt and minife show improvements and perform better with weighted round-robin as due to more memory requests, larger weights are allocated to both of them and get more share of bandwidth. In fig. 7.5b, we show the memory latency of Mix-4A workloads while using two shared memory nodes. We observe a slight improvement in memory latency for all the cases as remote memory allocation is now performed across two memory nodes. The memory request traffic is distributed among them as the memory requests follow the location of their respective page, which is the reason behind the performance difference.

Fig. 7.6a and 7.6b show the IPC of all the workloads for both scenarios and are normalized to the IPC of the local-only system. With 40% of memory footprint in the local memory, all workloads perform around 60% of the baseline when 2-shared memory nodes are used. On the other hand, all the workloads face significant slowdowns with a single memory pool. Even with more increases in memory latency, Pennant faces a minor slowdown due to fewer memory requests.

In the next scenario, our workload mix is created to measure the performance when

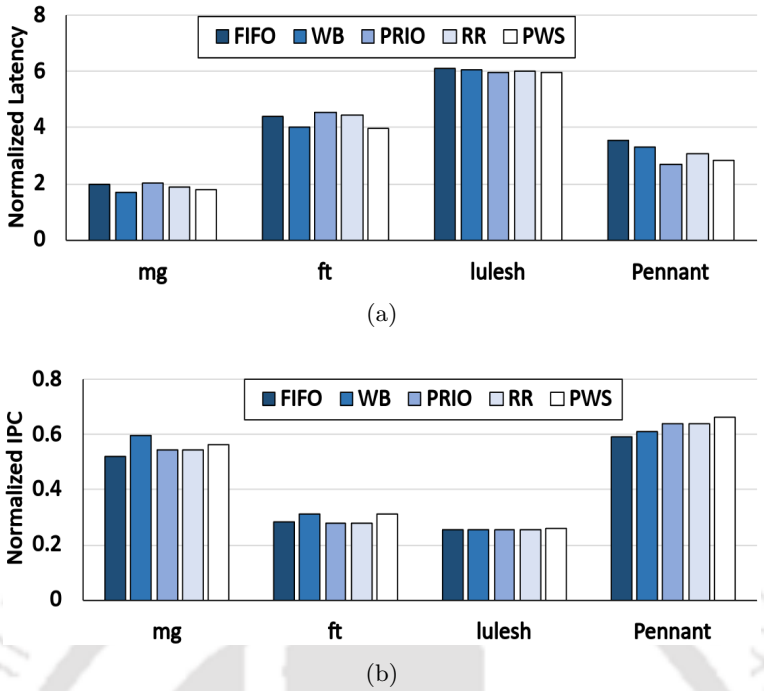
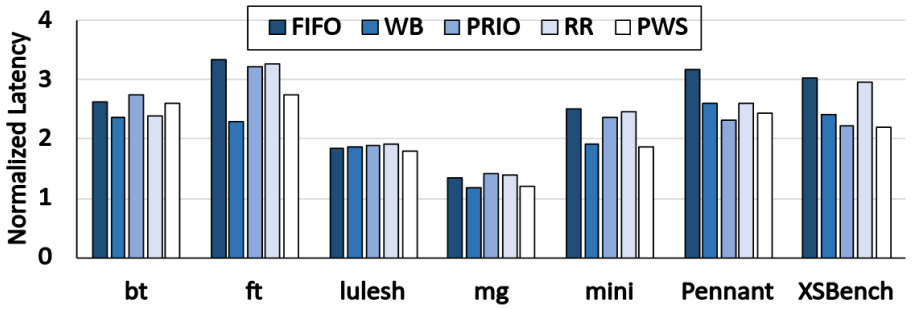


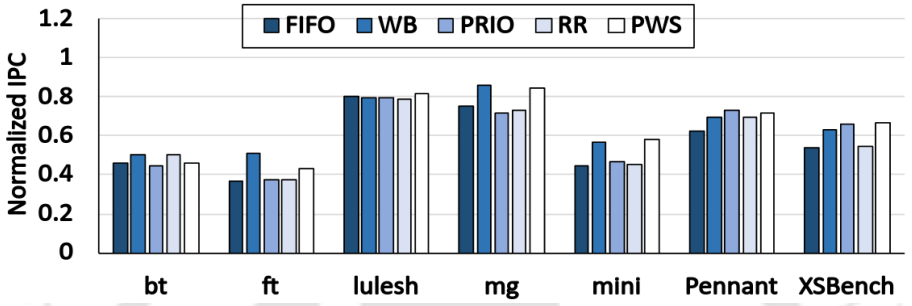
FIGURE 7.7: Workload Mix-4B using 2-shared memory node with 60% of workloads footprint at remote to a system with entirely local memory (a) Normalized memory latency (b) Normalized IPC

the system is under a huge load. We evaluate Mix-4b consisting of only one memory-intensive workload, and the other three workloads have a heavy memory-bound nature. As shown in Fig. 7.7a, all the nodes face a significant increase in memory access latency due to high memory traffic, even with 2-shared memory nodes. However, PRIO and PWS improve the memory latency of Pennant even when the system's total memory request load is very high, due to their high-priority allocation. NAS:mg, even with a small hit in memory latency, suffers a significant IPC slowdown, which is due to the memory-bound nature of the workload, shown in Fig. 7.7b. On the other hand, Lulesh faces a significant increase in its memory access latency. The reason behind this is the tail latency in memory access, as Lulesh sends memory accesses in large bursts, whereas the contention is already high. Lulesh only has a limited number of memory accesses compared to other memory-bound workloads, but during the burst time, memory access latency increases rapidly for all workloads, which is more visible in Lulesh. All the other scheduling policies fail to provide performance improvement except for WB to some extent with workloads NAS:mg and NAS:ft.

Lastly, we evaluate a bigger configuration of 8 nodes in Mix-8 with 4-shared memory pools. The workload mix has three high memory-sensitive (2-XSBench and Pennant) nodes, and the rest 5 vary in their memory sensitivity from moderate to low. The XSBench is used at two nodes in this configuration while we report the mean of their results. Fig. 7.8a shows the increase in memory latency compared to 100% local memory



(a)



(b)

FIGURE 7.8: Workload Mix-8 using 4-shared memory node with 50% of workloads footprint at remote to a system with entirely local memory (a) Normalized memory latency (b) Normalized IPC

system. As we can see memory latency of all the workloads increases between 1.25x to 3.25x. In this case, weighted round-robin performs better with multiple workloads such as bt, ft, mg, and mini. On the other hand, PWS and PRIo could also improve for Pennant and XSBench. Fig. 7.8b shows the decrease in IPC due to remote memory access with a significant improvement for minife and mg with PWS.

7.5 Summary

DMS is expected to replace traditional server systems in data centers. Memory latency remains an issue due to memory moving to the network. Further, these systems are expected to be deployed in groups of multiple compute nodes and memory nodes, the chances of increasing memory latency are even more, as multiple compute nodes share the same memory nodes. This work explored the impact of memory request scheduling policies to reduce memory latency and enforce QOS. In certain cases, our scheduling algorithms were able to reduce latency for specific workloads having high priority or larger weights. Priority-based queue allocation takes care of compute-intensive workloads, whereas weighted round-robin was able to decrease the latency of memory-bound workloads. Due to enormous memory requests, the improvement is not visible often, as it could make small improvements in IPC.

Chapter 8
Conclusion and Future Work

In this thesis, we explore large-scale DMS for memory scalability and under-utilization in server systems by working on various aspects like design, modeling, and optimization. Presently, DMSs are under huge focus as an alternative to traditional server systems with a target of reducing TCO and improving memory capacity/bandwidth and utilization. Presently, a full-fledged simulator does not exist that can be used to evaluate and model new features for optimization in a large-scale DMS. Thus, we built a simulator from scratch that is easy to use and modify. Existing techniques like hot page migration for reducing memory latency in similar systems (having multi-tiered memory with flat organization of fast and slow memory) do not work for large-scale DMS, which have more constraints due to limited network and memory bandwidth. Hence, we propose multiple optimizations for implementing page migration is DMS. The next section mentions the summary of the thesis. Section 8.1 provides insight into the possible future work in this DMS.

8.1 Summary

Figure 8.1 summarizes the contributions made toward the thesis. We covered multiple aspects, such as design, modeling, and optimizations, that can be used for building a fully operational large-scale DMS. We present the baseline DMS design in Chapter 3 that defines the memory organization, setting up memory mappings and the role of a global memory manager. We also built a simulator to evaluate the performance impact of disaggregation on different single or multi-threaded workloads. Our simulator

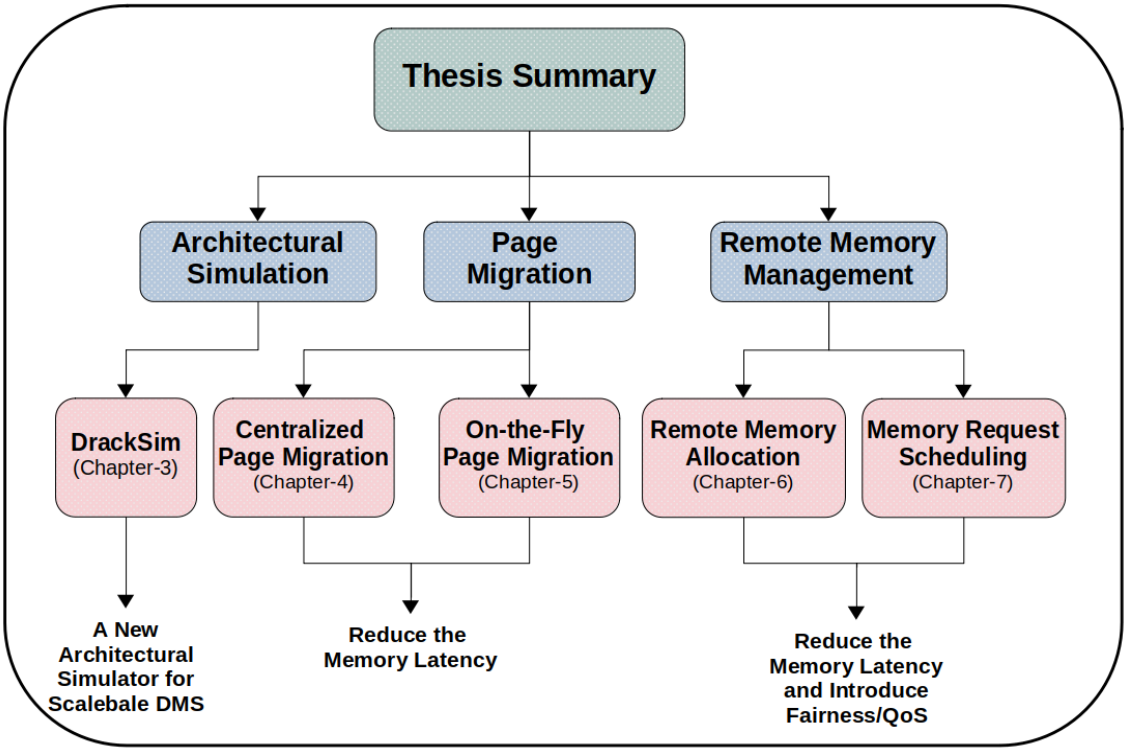


FIGURE 8.1: Thesis Summary

can be easily used and modified to implement different optimizations and allow fast simulation of a large-scale DMS. In the subsequent chapters, we proposed page migration systems tuned for a scalable DMS, considering the limited memory bandwidth when multiple compute nodes access the remote memory nodes simultaneously. The proposed design performed 10% to 100% better than the traditional RDMA-based DMS and 5% to 35% better than the baseline DMS. In Chapter 4, we proposed a centralized mechanism that identifies the hot pages for each compute node and accesses the remote page at cache block granularity to eliminate delays on critical memory accesses. An intermediate caching mechanism holds the hot pages until they are migrated to the compute nodes and reduces the data path for memory access in those pages. In Chapter 5, we proposed *CosMoS*, an on-the-fly page migration mechanism working at individual nodes. We perform workload characterization and further improve the remote page access by ordering the cache block access within a page. We also implement a multi-queue structure for scheduling hot pages based on their memory access frequency, which can be significantly different based on the memory access patterns of the workload. *CosMoS* performed 20% better than the state-of-the-art page migration design and 86% better than the baseline DMS. In Chapter 6, we propose remote memory allocation schemes that improve fairness in memory allocation by load-balancing and maximize the utilization of available bandwidth across remote memory nodes. Our proposed policies reduced the memory latency 3x to 4x on different combinations of workloads compared to the conventional policies. Finally, in Chapter 7, we evaluate the impact of memory

request scheduling when multiple compute nodes run different workloads in a large-scale DMS. We implement the commonly used queue-based scheduling mechanisms and allocate priority/weights to certain nodes based on their memory request rate. Our study reveals that suitable priority/weight allocation to different nodes can improve QoS and fairness.

8.2 Future Work

The contributions of the thesis can be extended in many ways. Some possible works that can be explored as future directions are summarized as follows.

- In Chapter 3, the proposed simulator DRackSim can run large-scale simulations. However, the queue-based interconnect modeling can become slow as the number of nodes grows. In future work, the queue-based model can be replaced with event modeling, increasing the simulation speed from 2x to 3x.
- Presently, DRackSim only models DDR4-based DRAM memory. In the future, DDR5 and HBM can also be considered for remote memory nodes that support better bandwidth scalability. Therefore, another memory simulator, such as DRAMSim3 [138] or Ramulator [139], can be integrated to support more memory technologies.
- Currently, there is no explicit management of cache coherence between last-level caches and the remote memory. Coherency is assumed by using write-back caches, and no other node accesses the data from the main memory while one node is modifying it. It will be interesting to explore the performance impact of extra latency that coherence will introduce into the system.
- In Chapters 4 and 5, we implement page migration to reduce the average memory access latency. A cache block prefetching can be implemented to hide the remote memory access latency, which eventually reduces the memory latency, like in [24]. The system predicts the address of future remote memory requests, prefetches well before the CPU requests it, and generates an LLC miss.
- To improve available memory bandwidth, an HBM can be considered an in-network cache for storing the hot pages from all the memory nodes (DDR4 with limited bandwidth). This will reduce the data path for remote memory access and eliminate many delays due to insufficient memory and network bandwidth.
- In chapter 7, we implement different queue allocation policies for scheduling remote memory requests and improving the QoS and fairness. However, the delays may still be bigger if the buffer space is not properly shared between input ports. A

dynamic buffer space allocation algorithm can be implemented to efficiently share the buffer between the input ports to minimize queueing delays.

Bibliography

[1] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, pp. 1–5, 2024.

[2] A. Bruyns, "Memory holds the keys to ai adoption | by antoine bruyns | medium," https://medium.com/@abruyns/memory-holds-the-keys-to-ai-adoption-5acd5e06508b, sep 2019, (Accessed on 03/31/2024).

[3] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *SIGPLAN Not.*, vol. 48, no. 4, p. 77–88, mar 2013.

[4] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: Association for Computing Machinery, 2012.

[5] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[6] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *2005 IEEE International Conference on Cluster Computing*, 2005, pp. 1–10.

[7] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun, "Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options," in *2009 17th IEEE Symposium on High Performance Interconnects*, 2009, pp. 123–130.

[8] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 267–278, jun 2009.

[9] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[10] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: a disseminated, distributed os for hardware resource disaggregation," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 69–87.

[11] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 868–880.

[12] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: a hardware-software co-designed disaggregated memory system," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 417–433.

[13] Y. Chang, K. Zhang, S. A. McKee, L. Zhang, M. Chen, L. Ren, and Z. Xu, "Extending on-chip interconnects for rack-level remote resource access," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 56–63.

[14] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out numa," *SIGPLAN Not.*, vol. 49, no. 4, p. 3–18, feb 2014.

[15] B. Zhao, R. Hou, J. Dong, M. Huang, S. A. Mckee, Q. Zhang, Y. Liu, Y. Li, L. Zhang, and D. Meng, "Venice: An effective resource sharing architecture for data center servers," *ACM Trans. Comput. Syst.*, vol. 36, no. 1, mar 2019.

[16] D. D. Sharma, "Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, 2023.

[17] CXL, "Cxl® specification - compute express link," https://computeexpresslink.org/cxl-specification/, nov 2020, (Accessed on 04/02/2024).

[18] "Cxl memory expansion: A closer look on actual platform," https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf, (Accessed on 04/04/2024).

[19] "Samsung unveils cxl memory module box: Up to 16 tb at 60 gb/s," https://www.anandtech.com/show/21333/samsung-unveils-cxl-memory-module-box-up-to-16-tb-at-60-gbs, April 2024, (Accessed on 07/05/2024).

[20] "Does the 4th gen intel® xeon® scalable processors support..." https://www.intel.com/content/www/us/en/support/articles/000094021/processors.html, (Accessed on 07/05/2024).

[21] "Amd epyc™ processors: Introducing the next generation of server processors," https://www.amd.com/en/partner/articles/4th-generation-amd-epyc.html, (Accessed on 07/05/2024).

[22] M. Islam, S. Adavally, M. Scrbak, and K. Kavi, "On-the-fly page migration and address reconciliation for heterogeneous memory systems," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 1, jan 2020.

[23] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang, "Supporting superpages and lightweight page migration in hybrid memory systems," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, apr 2019.

[24] V. R. Kommareddy, J. Kotra, C. Hughes, S. D. Hammond, and A. Awad, "Prefam: Understanding the impact of prefetching in fabric-attached memory architectures," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 323–334.

[25] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'20. USA: USENIX Association, 2020.

[26] P. Levis, K. Lin, and A. Tai, "A case against cxl memory pooling," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, ser. HotNets '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 18–24.

[27] E. Amaro, S. Wang, A. Panda, and M. K. Aguilera, "Logical memory pools: Flexible and local disaggregated memory," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, ser. HotNets '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 25–32.

[28] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 488–504.

[29] Intel, "4th Gen Xeon Scalable Processors — intel.com," https://www.intel.com/content/www/us/en/products/docs/processors/xeon-accelerated/4th-gen-xeon-scalable-processors.html, 2023, [Accessed 12-10-2023].

[30] AMD, "4th gen amd epyc processor architecture," https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf, 9 2023, (Accessed on 10/12/2023).

[31] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587.

[32] J. V. Quiroga, M. Torrents, N. Sonmez, D. Theodoropoulos, F. Zyulkyarov, and M. Nemirovsky, "Evaluation of a rack-scale disaggregated memory prototype for cloud data centers," in *Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP'19)*, ser. RSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–21.

[33] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755.

[34] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "Rack-scale disaggregated cloud data centers: The dredbox project vision," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 690–695.

[35] M. Bielski, I. Syrigos, K. Katrinis, D. Syrivelis, A. Reale, D. Theodoropoulos, N. Alachiotis, D. Pnevmatikatos, E. Pap, G. Zervas, V. Mishra, A. Saljoghei, A. Rigo, J. F. Zazo, S. Lopez-Buedo, M. Torrents, F. Zyulkyarov, M. Enrico, and O. G. de Dios, "dredbox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1093–1098.

[36] Q. Yang, R. Jin, B. Davis, D. Inupakutika, and M. Zhao, "Performance evaluation on cxl-enabled hybrid memory pool," in *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2022, pp. 1–5.

[37] C. Giannoula, K. Huang, J. Tang, N. Koziris, G. Goumas, Z. Chishti, and N. Vijaykumar, "Daemon: Architectural support for efficient data movement in fully disaggregated systems," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 1, mar 2023.

[38] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 417–427.

[39] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, p. 16–19, jan 2011.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[41] A. Puri, K. Bellamkonda, K. Narreddy, J. Jose, V. Tamarapalli, and V. Narayanan, "Dracksim: Simulating cxl-enabled large-scale disaggregated memory systems," in *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 3–14. [Online]. Available: https://doi.org/10.1145/3615979.3656059

[42] A. Puri, K. Bellamkonda, K. Narreddy, J. Jose, and T. Venkatesh, "A practical approach for workload-aware data movement in disaggregated memory systems," in *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2023, pp. 78–88.

[43] A. Puri, J. Jose, and T. Venkatesh, "Design and evaluation of a rack-scale disaggregated memory architecture for data centers," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 2022, pp. 212–217.

[44] A. Puri, A. Banerjee, J. Jose, and T. Venkatesh, "Understanding the performance impact of queue-based resource allocation in scalable disaggregated memory systems," in *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2023, pp. 317–324.

[45] H. Aghaei Khouzani, F. S. Hosseini, and C. Yang, "Segment and conflict aware page allocation and migration in dram-pcm hybrid main memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1458–1470, 2017.

[46] R. Pandey and A. Sahu, "Access-aware self-adaptive data mapping onto 3d-stacked hybrid dram-pcm based chip-multiprocessor," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 389–396.

[47] J.-T. Yun, S.-K. Yoon, J.-G. Kim, B. Burgstaller, and S.-D. Kim, "Regression prefetcher with preprocessing for dram-pcm hybrid main memory," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 163–166, 2018.

[48] H. G. Lee, S. Baek, C. Nicopoulos, and J. Kim, "An energy- and performance-aware dram cache architecture for hybrid dram/pcm main memory systems," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, 2011, pp. 381–387.

[49] Y. Fu, Y. Lu, Z. Chen, Y. Wu, and N. Xiao, "Design and simulation of content-aware hybrid dram-pcm memory system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1666–1677, 2022.

[50] H. Aghaei Khouzani, F. S. Hosseini, and C. Yang, "Segment and conflict aware page allocation and migration in dram-pcm hybrid main memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1458–1470, 2017.

[51] X. Cai, L. Ju, M. Zhao, Z. Sun, and Z. Jia, "A novel page caching policy for pcm and dram of hybrid memory architecture," in *2016 13th International Conference on Embedded Software and Systems (ICESS)*, 2016, pp. 67–73.

[52] O. Patil, F. Mueller, L. Ionkov, J. Lee, and M. Lang, "Symbiotic hw cache and sw dtlb prefetching for dram/nvm hybrid memory," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[53] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Energy-efficient hybrid dram/nvm main memory," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 492–493.

[54] R. Salkhordeh, O. Mutlu, and H. Asadi, "An analytical model for performance and lifetime estimation of hybrid dram-nvm main memories," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1114–1130, 2019.

[55] R. Salkhordeh and H. Asadi, "An operating system level data migration scheme in hybrid dram-nvm memory architecture," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 936–941.

[56] X. Chen, E. H.-M. Sha, W. Jiang, Q. Zhuge, J. Chen, J. Qin, and Y. Zeng, "The design of an efficient swap mechanism for hybrid dram-nvm systems," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10.

[57] Z. Peng, D. Feng, J. Chen, J. Hu, and C. Huang, "Rhpm: Using relative hotness to guide page migration for hybrid memory systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 8, pp. 2514–2526, 2023.

[58] M. Han, J. Hyun, S. Park, and W. Baek, "Hotness- and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 377–391, 2020.

[59] J. Hu, H. Liu, H. Jin, and X. Liao, "Design and simulation of multi-tiered heterogeneous memory architecture," in *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2022, pp. 113–120.

[60] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir, "Chameleon: A dynamically reconfigurable heterogeneous memory system," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 533–545.

[61] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *2005 IEEE International Conference on Cluster Computing*, 2005, pp. 1–10.

[62] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A Memory-Disaggregated managed runtime," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 261–280.

[63] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 649–667.

[64] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 249–264.

[65] M. Chapman and G. Heiser, "vNUMA: A virtual Shared-Memory multiprocessor," in *2009 USENIX Annual Technical Conference (USENIX ATC 09)*. San Diego, CA: USENIX Association, Jun. 2009.

[66] Z. Ma, Z. Sheng, and L. Gu, "Dvm: A big virtual machine for cloud computing," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2245–2258, 2014.

[67] H. Montaner, F. Silla, and J. Duato, "A practical way to extend shared memory support beyond a motherboard at low cost," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 155–166.

[68] H. Montaner, F. Silla, H. Fröning, and J. Duato, "Getting rid of coherency overhead for memory-hungry applications," in *2010 IEEE International Conference on Cluster Computing*, 2010, pp. 48–57.

[69] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 148–159.

[70] J. Laudon and D. Lenoski, "The sgi origin: a ccnuma highly scalable server," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, p. 241–251, may 1997.

[71] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, p. 302–313, apr 1994.

[72] K. Alnaes, E. Kristiansen, D. Gustavson, and D. James, "Scalable coherent interface," in *COMPEURO'90: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering - Systems Engineering Aspects of Complex Computerized Systems*, 1990, pp. 446–453.

[73] R. Fatoohi, "Performance evaluation of the dual-core based sgi altix 4700," in *19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07)*, 2007, pp. 97–104.

[74] "Konecny_paper.pdf," https://cug.org/5-publications/proceedings_attendee_lists/2007CD/S07_Proceedings/pages/Authors/Konecny/Konecny_paper.pdf, May 2007, (Accessed on 07/05/2024).

[75] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu, and S. Zhang, "Cost effective data center servers," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 179–187.

[76] C.-C. Tu, C.-t. Lee, and T.-c. Chiueh, "Marlin: a memory-based rack area network," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 125–136.

[77] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414.

[78] 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.

[79] M. Bichan, C. Ting, B. Zand, J. Wang, R. Shulyzki, J. Guthrie, K. Tyshchenko, J. Zhao, A. Parsafar, E. Liu, A. Vatankhahghadim, S. Sharifian, A. Tyshchenko, M. De Vita, S. Rubab, S. Iyer, F. Spagna, and N. Dolev, "A 32gb/s nrz 37db serdes in 10nm cmos to support pci express gen 5 protocol," in *2020 IEEE Custom Integrated Circuits Conference (CICC)*, 2020, pp. 1–4.

[80] J. Dong, R. Hou, M. Huang, T. Jiang, B. Zhao, S. A. McKee, H. Wang, X. Cui, and L. Zhang, "Venice: Exploring server architectures for effective resource sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 507–518.

[81] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 844–855, 2020.

[82] H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C.-z. Xu, "Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1350–1363, 2015.

[83] N. Alachiotis, A. Andronikakis, O. Papadakis, D. Theodoropoulos, D. Pnevmatikatos, D. Syrivelis, A. Reale, K. Katrinis, G. Zervas, V. Mishra, H. Yuan, I. Syrigos, I. Igoumenos, T. Korakis, M. Torrents, and F. Zyulkyarov, *dReDBox: A Disaggregated Architectural Perspective for Data Centers*. Cham: Springer International Publishing, 2019, pp. 35–56.

[84] D. Syrivelis, A. Reale, K. Katrinis, I. Syrigos, M. Bielski, D. Theodoropoulos, D. N. Pnevmatikatos, and G. Zervas, "A software-defined architecture and prototype for disaggregated memory rack scale systems," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2017, pp. 300–307.

[85] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 287–294.

[86] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 73–82.

[87] X. Ji, C. Wang, N. El-Sayed, X. Ma, Y. Kim, S. S. Vazhkudai, W. Xue, and D. Sanchez, "Understanding object-level memory access patterns across the spectrum," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017.

[88] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive page migration policy with huge pages in tiered memory systems," *IEEE Transactions on Computers*, vol. 71, no. 1, pp. 53–68, 2022.

[89] N. Niu, F. Fu, B. Yang, Q. Wang, X. Li, F. Lai, and J. Wang, "Pfha: A novel page migration algorithm for hybrid memory embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 10, pp. 1685–1692, 2021.

[90] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for Multi-Tiered memory systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 715–728.

[91] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 727–741.

[92] S. Wu, B. Wang, C. Yang, Q. He, and J. Chen, "A hot-page aware hybrid-copy migration method," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 220–221.

[93] 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, p. 15–22, sep 2022.

[94] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 313–326.

[95] D. Mosberger and S. Eranian, *IA-64 Linux Kernel: Design and Implementation*. USA: Prentice Hall PTR, 2001.

[96] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 273–287.

[97] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 340–349.

[98] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk's a hit: Making page walks single-access cache hits," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 128–141.

[99] T. Marinakis and I. Anagnostopoulos, "Performance and fairness improvement on cmps considering bandwidth and cache utilization," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 1–4, 2019.

[100] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200.

[101] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, ser. WCAE '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 22–es.

[102] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, jun 2005.

[103] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USA: USENIX Association, 2014, p. 401–414.

[104] Intel, "Intel® Rack Scale Design (Intel® RSD) Architecture White Paper — intel.in," https://www.intel.in/content/www/in/en/architecture-and-technology/rack-scale-design/rack-scale-design-architecture-white-paper.html, 2017, [Accessed 02-Jul-2023].

[105] J. Taylor, "Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond," in *Optical Fiber Communication Conference*. Optica Publishing Group, 2015, p. W1D.5.

[106] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1050–1055.

[107] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011.

[108] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2013, pp. 74–85.

[109] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 53–64.

[110] S. Hong, W.-O. Kwon, and M.-H. Oh, "Hardware implementation and analysis of gen-z protocol for memory-centric architecture," *IEEE Access*, vol. 8, pp. 127 244–127 253, 2020.

[111] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[112] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[113] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[114] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, and L. G. Fernandes, "Efficient nas benchmark kernels with c++ parallel programming," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 733–740.

[115] HPCG, "GitHub - hpcg-benchmark/hpcg: Official HPCG benchmark source code — github.com," https://github.com/hpcg-benchmark/hpcg/, 2019, [Accessed 18-Jul-2023].

[116] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–146.

[117] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," 7 2013. [Online]. Available: https://www.osti.gov/biblio/1090032/

[118] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, "Improving performance via mini-applications." 9 2009. [Online]. Available: https://www.osti.gov/biblio/993908/

[119] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[120] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He, "SimpleMOC – a performance abstraction for 3D MOC," in *ANS & M&C 2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, 2015.

[121] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014.

[122] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "Drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–14.

[123] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "Dc.p4: Programming the forwarding plane of a data-center switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: Association for Computing Machinery, 2015.

[124] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014.

[125] [Online]. Available: https://www.juniper.net/us/en/products/switches/ex-series/ex9200-programmable-network-switch.html

[126] [Online]. Available: https://www.intel.com/content/www/us/en/products-/network-io/programmable-ethernet-switch.html

[127] P. Messina, "The exascale computing project," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 63–67, 2017.

[128] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 293–304.

[129] Y. Tan, B. Wang, Z. Yan, Q. Deng, X. Chen, and D. Liu, "Uimigrate: Adaptive data migration for hybrid non-volatile memory systems," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 860–865.

[130] Z. Peng, D. Feng, J. Chen, J. Hu, and C. Huang, "Rhpm: Using relative hotness to guide page migration for hybrid memory systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 8, pp. 2514–2526, 2023.

[131] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/.

[132] H. Yuanfu and W. Xunsen, "The methods of improving the compression ratio of lz77 family data compression algorithms," in *Proceedings of Third International Conference on Signal Processing (ICSP'96)*, vol. 1, 1996, pp. 698–701 vol.1.

[133] R. LaRowe, C. Ellis, and M. Holliday, "Evaluation of numa memory management through modeling and measurements," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 686–701, 1992.

[134] L. Bhuyan, R. Iyer, H.-J. Wang, and A. Kumar, "Impact of cc-numa memory management policies on the application performance of multistage switching networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 3, pp. 230–246, 2000.

[135] M. Diener, E. H. Cruz, and P. O. Navaux, "Locality vs. balance: Exploring data mapping policies on numa systems," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 9–16.

[136] SPEC, "Standard performance evaluation corporation," june 2017.

[137] L. Yang, C. Pan, E. Zhang, and H. Liu, "A new class of priority-based weighted fair scheduling algorithm," *Physics Procedia*, vol. 33, pp. 942–948, 2012, 2012 International Conference on Medical Physics and Biomedical Engineering (ICMPBE2012).

[138] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[139] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

List of Publications

International Conferences

- Amit Puri, John Jose, Tamarapalli Venkatesh, "**Design and Evaluation of a Rack-Scale Disaggregated Memory Architecture For Data Centers**," 24th IEEE International Conference on High Performance Computing and Communications [HPCC], pp. 212-217, December 2022. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00060
- Amit Puri, Kartheek Bellamkonda, Kailash Narreddy, John Jose, Venkatesh Tamarapalli, "**A Practical Approach For Workload-Aware Data Movement in Disaggregated Memory Systems**," 35th IEEE International Symposium on Computer Architecture and High Performance Computing [SBAC-PAD], pp. 78-88, October 2023. DOI: 10.1109/SBAC-PAD59825.2023.00017
- Amit Puri, John Jose, Venkatesh Tamarapalli, "**Understanding the Performance Impact of Queue-Based Resource Allocation in Scalable Disaggregated Memory Systems**,"16th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip [MCSOC], pp. 317-324, December 2023. DOI: 10.1109/MCSoC60832.2023.00054
- Amit Puri, Kartheek Bellamkonda, Kailash Narreddy, John Jose, Tamarapalli Venkatesh, Vijaykrishnan Narayanan, "**DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems**," 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation [PADS], 2024 DOI: 10.1145/3615979.3656059

Journals

- Amit Puri, John Jose, Tamarapalli Venkatesh, Vijaykrishnan Narayanan, "**CosMoS: Architectural Support for Cost-Effective Data Movement in a Scalable Disaggregated Memory Systems**," (Under Submission).

Posters

- Amit Puri, John Jose, Tamarapalli Venkatesh, "**Optimizing Memory Latency in Hardware Disaggregated Memory Systems**," In PhD Forum, Embedded Systems Week, [ESWEEK], 2023