Optimizing Memory Performance in Multi-Core Systems through Data Placement

An M. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Akshay Bhosale
(234101006)

under the guidance of

Dr. Aryabartta Sahu

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Optimizing Memory Performance in Multi-Core Systems through Data Placement**" is a bonafide work of **Akshay Bhosale (Roll No. 234101006)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. Aryabartta Sahu**

Associate Professor,

Department of Computer Science & Engineering,

Indian Institute of Technology Guwahati.

May, 2025

Guwahati, Assam.

Acknowledgements

I sincerely thank my project supervisor, **Dr. Aryabartta Sahu**, for his constant guidance and support throughout this work. I also thank the **Department of Computer Science and Engineering, IIT Guwahati** for providing the necessary environment. I'm grateful to my labmates, PhD scholars, and friends for their helpful discussions.

Abstract

With increasing computational demands, optimizing memory performance has become essential in modern multi-core systems. Dynamic Random Access Memory (DRAM), the primary working memory, often struggles to meet these demands efficiently—especially in Non-Uniform Memory Access (NUMA) architectures, where memory access latency varies based on the proximity of the accessing core to the memory channel.

This thesis presents a translation-layer technique called Dynamic Migration, designed to improve DRAM performance by intelligently placing data across memory channels. The method periodically monitors access patterns to identify frequently accessed (hot) pages and migrates them to memory channels that are closer to the cores that access them most. By aligning data placement with usage patterns, this technique reduces access delays and improves system efficiency.

The approach is implemented in Ramulator2 and evaluated using memory access traces generated from SniperSim. Experimental results show that Dynamic Migration achieves a measurable reduction in average memory access latency ranging from 27% to over 50% across diverse workloads, offering a lightweight, modular solution for improving memory performance in NUMA-like systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In multi-core computing systems, memory performance is one of the most important factors affecting overall system efficiency. As the number of cores in processors continues to grow, ensuring fast and reliable memory access has become increasingly critical. DRAM (Dynamic Random-Access Memory), which serves as the primary storage for active data, is a key component in this process. However, due to its architecture, DRAM often struggles to deliver consistent performance, especially in systems handling heavy and complex workloads.

DRAM is divided into multiple channels, each serving as an independent access path. While this design enables parallelism, it also introduces challenges when multiple cores contend for memory resources. Each core in a multi-core system accesses memory channels differently due to factors like physical proximity and shared contention. This results in varying memory access latencies, particularly in Non-Uniform Memory Access (NUMA) architectures. A memory access is considered "local" when it targets a nearby memory channel and "remote" when the accessed channel is farther away. Such variations can degrade performance, especially when frequently accessed data is placed in suboptimal memory locations.

To address these challenges, efficient memory management is essential. By understanding latency patterns, access behaviors, and DRAM structure, system performance can be significantly improved. This thesis focuses on optimizing the Virtual Page Number (VPN) to Physical Page Number (PPN) translation process, which determines where a memory page physically resides. We propose techniques that align page placement with actual access patterns, minimizing delays caused by remote accesses.

Two methods are proposed in this work:

- **Local to Requester Mapping:** Each virtual page is assigned to the memory channel that is closest to the first core that accesses it.
- **Dynamic Migration:** Pages are periodically evaluated for access frequency and are migrated to a better memory channel if the access pattern suggests it will reduce latency.

Both methods are implemented in Ramulator2 [1], a cycle-level DRAM simulator, using traces generated from SniperSim [2].

Chapter 2

Background

2.1 DRAM Organization

The DRAM architecture organizes its components in a structured hierarchy to store and retrieve data efficiently. Channels, ranks, banks, rows, and columns work together to handle memory operations effectively. Fig. 2.1 illustrates the high-level logical diagram of DRAM architecture.

Channels are like pathways connecting the processor to the memory. Simultaneous data transfers are enabled by multiple channels, which increase memory bandwidth. Channels can process different memory requests concurrently, particularly beneficial in multi-core systems where multiple processors access memory at the same time.

Within each channel, ranks are formed by groups of memory chips working together to expand capacity. While multiple ranks exist within a channel, only one rank is active at a time. Further divided into banks, each rank allows independent storage operations. Banks process separate memory requests simultaneously. Data within a bank is arranged in a grid of rows and columns. A row, made up of memory cells storing individual bits, is loaded into a temporary row buffer when accessed. The data's exact location is identified by its column, which specifies its position within the row. After identification, the processor retrieves or writes the required data.

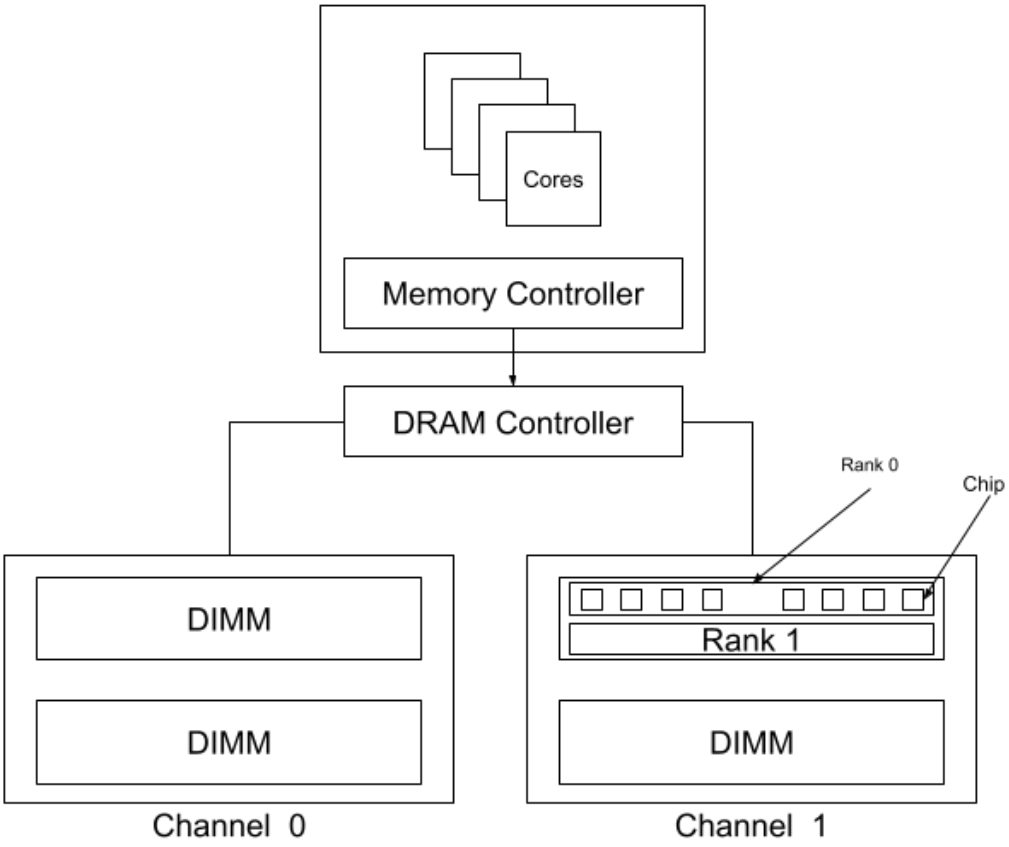

Fig. 2.1 High-level architecture of DRAM

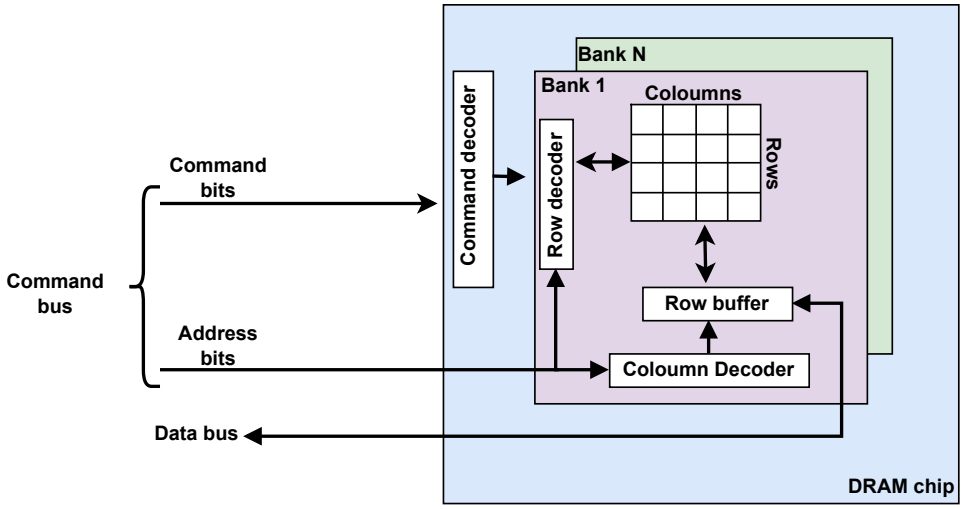

Fig. 2.2 DRAM Chip architecture

2.2 Address Mapping in DRAM

Address mapping tells us how memory addresses from the processor are translated into one address type to another address type.

2.2.1 Virtual to Physical Address Mapping

When a program requests data, it uses a virtual address. This virtual address is then translated to a physical address that tells us location in the physical memory (DRAM).

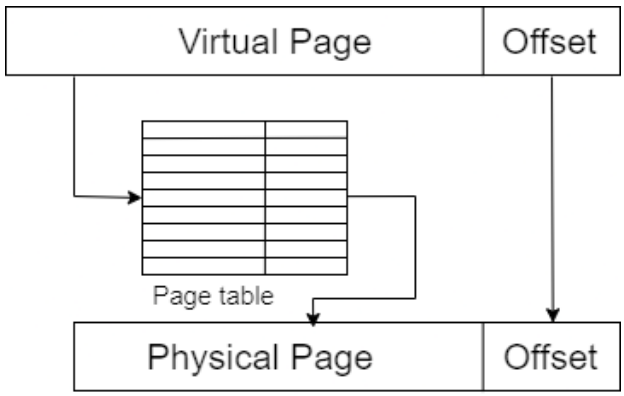

Fig. 2.3 Virtual Address to Physical Address translation

This translation uses page tables to map virtual addresses to physical addresses.

2.2.2 Physical to Hardware Mapping

After the virtual address is translated into a physical address, the memory controller maps this physical address to specific DRAM hardware components. For example, the physical address is divided into components that specify the channel, rank, bank, row, and column. Each component represents a portion of the DRAM hierarchy: The channel determines which memory channel will handle the request, the rank identifies the rank within the channel, the bank specifies the bank within the rank, the row refers to the specific row to be accessed in the selected bank, and the column identifies the exact column within the row.

Fig. 2.4 Physical Address breakdown

2.3 Memory Access Flow

When a memory request is issued, the memory controller decodes the physical address into channel, rank, bank, row, and column fields. If the requested row is not already active in the target bank, the controller issues a precharge command to close any open row, followed by an activate command to open the desired row, and then a read or write command. This sequence introduces latency. If the same row is accessed again (a row hit), the latency is significantly lower. However, a row conflict or row miss introduces additional delay. Efficient memory placement can reduce conflicts and improve access time.

2.4 NUMA Architecture

In Uniform Memory Access (UMA) systems, all cores have equal access latency to all memory. However, this model becomes inefficient in large multi-core processors. To address scalability and performance issues, systems adopt Non-Uniform Memory Access (NUMA) architectures. In NUMA, memory is divided into regions, each physically closer to a subset of cores. Accessing memory within the same region (local access) is faster than accessing memory in a different region (remote access).

NUMA introduces challenges in memory management because suboptimal placement decisions can lead to remote accesses and increased latency. Understanding which core accesses which memory page most frequently is crucial for improving placement and overall performance.

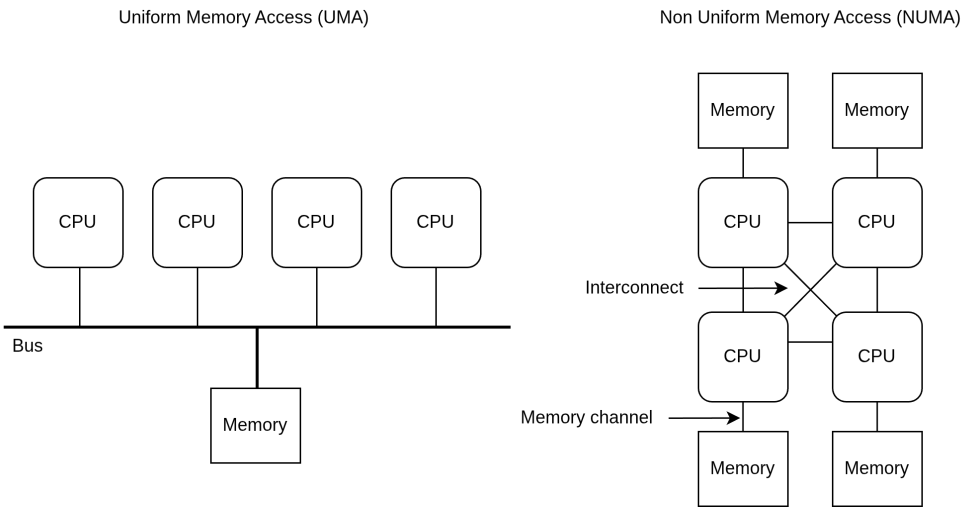

Fig. 2.5 Differences between a NUMA and UMA architecture.

2.5 Interconnect and Remote Access

In systems with multiple memory channels or NUMA nodes, an interconnect fabric connects cores to remote memory regions. When a memory request crosses this interconnect to reach a remote channel, it incurs additional latency. While all cores can access all memory, the latency depends on distance and contention across the interconnect.

Reducing remote memory accesses, and thereby reducing interconnect usage, is a practical way to lower memory latency. This is why many memory optimization strategies, including the one in this thesis, aim to keep data closer to the accessing core.

Chapter 3

Related Work

Improving memory performance in NUMA (Non-Uniform Memory Access) systems has been a common goal in both research and industry. In this chapter, we discuss some of the key ideas that have been proposed in the past.

Some researchers have worked on placing threads and memory smartly across NUMA nodes to reduce slow memory accesses. For example, Giacobazzi et al. [3] suggested using classifiers to choose between different placement strategies like "Compact" or "Scatter" for threads and "First-touch" or "Interleave" for memory. They trained these classifiers using past data from hardware counters. However, this method depends heavily on prior profiling and may not work well when program behavior changes during execution. COMPLACE [4] is another technique that places threads based on how often they communicate with each other. While this helps reduce communication delay, it also adds overhead and needs information that is not always available.

Another area of work deals with how virtual memory is managed. Panwar's PhD thesis [5] talks about several techniques to reduce the delay caused by virtual-to-physical address translation. He proposed vMitosis, which moves or copies page tables closer to the cores that use them. These approaches aim to reduce TLB misses and improve translation performance. While useful, they mainly optimize metadata like page tables.

Another well-known approach is Carrefour [6], a runtime system that dynamically switches between memory interleaving, collocation, and replication based on observed access patterns and system congestion. Carrefour is adaptive and improves memory locality, but it depends on kernel modifications and hardware performance counters, which makes it hard to apply or evaluate in simulator-based research like ours. Our focus is different—we look at where the actual data is placed in DRAM and try to keep it close to where it is most used.

Another relevant work is the paper *Run-time Adaptive Data Page Mapping* [7], which proposed a method for mapping pages in 3D-stacked DRAM-based systems. Their technique tracks how far a memory access has to travel and tries to keep pages close to the core that accesses them. They also use a small SRAM buffer to cache frequently accessed data. This leads to good results but is built for a specific type of hardware that includes 3D-stacked memory and a mesh interconnect. In contrast, we work with standard NUMA systems and avoid hardware-specific dependencies. Also, as pointed out in [7], many of these studies focus more on reducing network travel time and ignore DRAM access latency itself. Our method directly targets this by making sure pages are placed on the DRAM channels that offer the fastest access.

In summary, past research shows that smart placement of threads and memory, better use of hybrid memory systems, and faster address translation can all help improve performance. But many of these methods rely on prior training, hardware changes, or detailed profiling. Our method avoids all of these. It simply observes which core is using which data and moves the data to a nearby DRAM channel. This approach is light, runs at runtime, needs no changes to the software or hardware, and works well in typical NUMA systems. It provides a practical way to reduce memory delays by making smarter choices about where data lives.

Chapter 4

Methodology

Our objective is to minimize memory access latency in NUMA-like DRAM systems by placing frequently accessed data closer to the cores that use them most. The placement strategies are implemented by controlling the translation from virtual page numbers (VPNs) to physical page numbers (PPNs), which effectively decides the memory channel where the page will reside.

4.1 Overview of the Approach

We implemented two memory translation strategies:

Local to Requester Mapping: In this method, the first core that accesses a page determines the memory channel assignment. The virtual page is mapped to the physical address space in the channel topologically closest to that core. Once assigned, the page does not migrate.

Algorithm 1 Local to Requester Mapping

Input: Virtual Page Number `VPN`, Accessing Core ID `core_id`
if `VPN` already mapped **then**
    Return existing physical address mapping
else
    Determine the best memory channel
    if Free page exists in chosen channel **then**
        Randomly select a free physical page from the channel
    else
        Evict an existing page in that channel
        Allocate the freed page
    Map `VPN` to selected physical page
Construct final physical address by combining `PPN` and offset bits
Return translated physical address

Dynamic Migration: This method periodically monitors page access patterns. Pages that are accessed frequently (hot pages) are migrated to the channel closer to the core that accesses them the most. Migration is triggered periodically, and each candidate page is evaluated to check if relocating it would provide a latency benefit. Only if this condition is met is the page migrated to a new channel closer to its most frequent accessing core.

Algorithm 2 Dynamic Page Migration

Input: Virtual Page Number `VPN`, Accessing Core ID `core_id`
Update per-core access count for `VPN`
if Access count update triggers a migration window **then**
    for all hot pages **do**
        Determine dominant core for the page
        Identify access distribution of the page across all cores
        if a single core dominates accesses **then**
            Select the channel with the lowest latency to that core
        else
            Select the channel that minimizes total latency across major accessing cores
        if Already in best channel or under cooldown **then**
            continue
        Estimate latency gain if migrated
        if Gain > Cost **then**
            Migrate page to new channel
            Update page table and cooldown timestamp
if `VPN` already mapped **then**
    Return existing mapping
else
    Allocate free page in best channel for `core_id` (or evict)
    Update mapping tables
    Return translated physical address

4.2 Migration Policy

The dynamic migration strategy works in periodic windows. In each window, the system monitors page access frequencies and identifies pages that cross a defined hotness threshold. For each hot page, the access distribution across all cores is examined. If a single core dominates, the page is considered for migration to the channel nearest to that core. In cases where multiple cores access the page significantly, a weighted latency cost is calculated for all channels, and the channel with the lowest total latency is selected.

To ensure meaningful and stable decisions, pages are only migrated if the estimated gain exceeds a threshold and the page is not currently in a cooldown phase. Additionally, pages with very low access counts are ignored to avoid reacting to noise.

Our algorithm makes decisions based on recent access history, assuming short-term stability in memory behavior—a common trait in real-world programs that exhibit phase-based execution. Expected latency gain is calculated using actual access counts and core-to-channel distances, weighted to reflect realistic cost. Migration is only allowed if this estimated gain exceeds a threshold and access distribution has remained consistent.

Chapter 5

Experimental Setup

This chapter describes the simulation environment, configuration parameters, benchmarks, and evaluation metrics used to assess the performance of the proposed memory placement strategies.

5.1 System Configuration

We assume a system with eight cores and eight DRAM channels. Each core is topologically mapped to a DRAM channel to reflect NUMA-like memory access latency differences. Ramulator2 simulates DRAM access patterns using pre-generated traces from SniperSim, allowing us to evaluate different translation and page placement strategies.

A diagram of the architecture showing the core-channel layout is provided in Fig. 5.1.

Table 5.1 System Configuration Parameters

Component	**Configuration Details**
Cores	8
Frontend	SimpleO3
Instruction Count	1 Billion
Memory System	GenericDRAM
DRAM Type	DDR4
Preset	DDR4_8Gb_x8
Channels	8
Ranks per Channel	2
Timing Preset	DDR4_2400R
Scheduler	FRFCFS
Refresh Policy	AllBank
Row Policy	OpenRowPolicy (cap = 4)
Address Mapping	ChRaBaRoCo

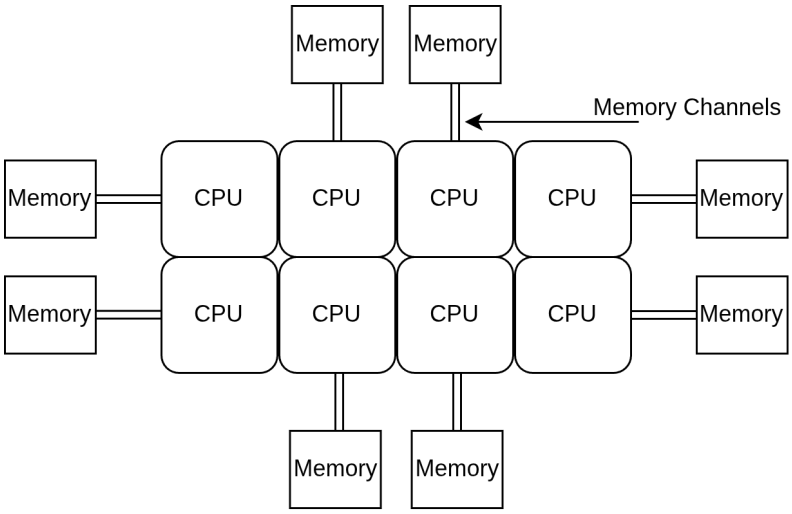

Fig. 5.1 Eight-core, eight-channel system used for simulation

This setup models a simplified NUMA environment by treating each DRAM channel as a logically distinct memory region with different latencies based on core-to-channel proximity. The core-to-channel proximity setup effectively captures NUMA-like memory access differences

5.2 Simulation Framework

Our simulation pipeline combines SniperSim and Ramulator2. SniperSim runs full-system benchmarks and produces memory access traces, including details of accessed addresses and issuing core IDs. These traces are processed by Ramulator2, which simulates DRAM behavior using a custom translation layer to implement Local to Requester and dynamic migration policies.

5.3 Benchmarks

We evaluated our translation strategies using OpenMP-based benchmarks, including workloads from the SPEC OMP2001 suite as well as other commonly used kernels. *Art*, *equake*, and *fft* are from SPEC OMP2001. *Art* performs thermal image recognition using an ART2 neural network; *equake* simulates seismic wave propagation with highly irregular memory access; and *fft* executes the Fast Fourier Transform with strided access patterns. The other benchmarks—*matmul*, *gauss*, and *PCNN*—are also OpenMP-based. *Matmul* performs dense matrix multiplication with regular access, *gauss* involves moderately structured memory behavior, and *PCNN* represents a convolutional neural network layer with bursty, semi-structured access. This mix enables a well-rounded evaluation of the proposed approach.

5.4 Translation Policies Compared

We compare the following memory page placement techniques Random Mapping (Baseline translation using random assignment of VPN to PPN), Local to Requester, Dynamic Migration

5.5 Configuration Variants of Dynamic Migration

To evaluate the flexibility and responsiveness of our dynamic migration technique, we simulated nine different configurations by varying three key parameters: window size, hot page threshold, and cooldown interval. Each configuration represents a unique balance between aggressiveness and stability in migration behavior.

Table 5.2 Dynamic Migration Configuration Variants

Label	Window Size	Threshold	Cooldown	Motivation
C1	25,000	1,000	20	Highly aggressive, migrates quickly but limits thrashing via long cooldown
C2	25,000	2,000	20	Slightly conservative vs. C1 to avoid reacting to minor access spikes
C3	50,000	5,000	10	Balanced setting for stable detection of meaningful hot pages
C4	50,000	10,000	10	Tests if stricter migration improves selectivity and reduces overhead
C5a	100,000	10,000	1	Aggressive long-window migration with minimal remigration control
C5b	100,000	10,000	5	Adds moderate cooldown to C5 to limit unnecessary remigrations
C5c	100,000	10,000	10	Similar to CD1 but with stronger remigration control
C5d	100,000	10,000	15	Most conservative cooldown in the C5-CD3 group
C6	200,000	30,000	5	Very conservative; only moves extremely hot pages after long observation

This range of configurations allows us to study the sensitivity of the migration strategy and identify which parameter combinations yield optimal latency improvements under varied memory access patterns.

5.6 Evaluation Metrics

We measure the effectiveness of each policy using two key metrics: average memory latency and percentage improvement. Average memory latency serves as the primary performance metric, reflecting the delay experienced during DRAM access. Percentage improvement quantifies the latency reduction achieved by a given policy compared to the baseline random strategy.

Chapter 6

Results and Analysis

This chapter presents the performance evaluation of three memory page placement strategies: Random mapping, Local to Requester mapping, and the proposed Dynamic Migration approach. Simulations were conducted on six diverse benchmarks under a NUMA-like core-to-channel latency model, with each benchmark executing for one billion instructions.

6.1 Overall Latency Comparison

Fig. 6.1 compares the average memory latency achieved by Random mapping, Local to Requester mapping, and the best-performing Dynamic Migration configuration for each benchmark.

The results show that Dynamic Migration consistently outperforms both Random and Local to Requester approaches across all workloads. Latency reductions range from 27% to over 50% for most benchmarks. Notably, *Equake* and *Gauss* see improvements of over 50% and 40%, respectively, compared to Random. Even in more regular access patterns like *Matmul*, the dynamic method achieves a significant 34.5% improvement.

These results confirm the advantage of adaptive page placement strategies, especially in systems with non-uniform access costs.

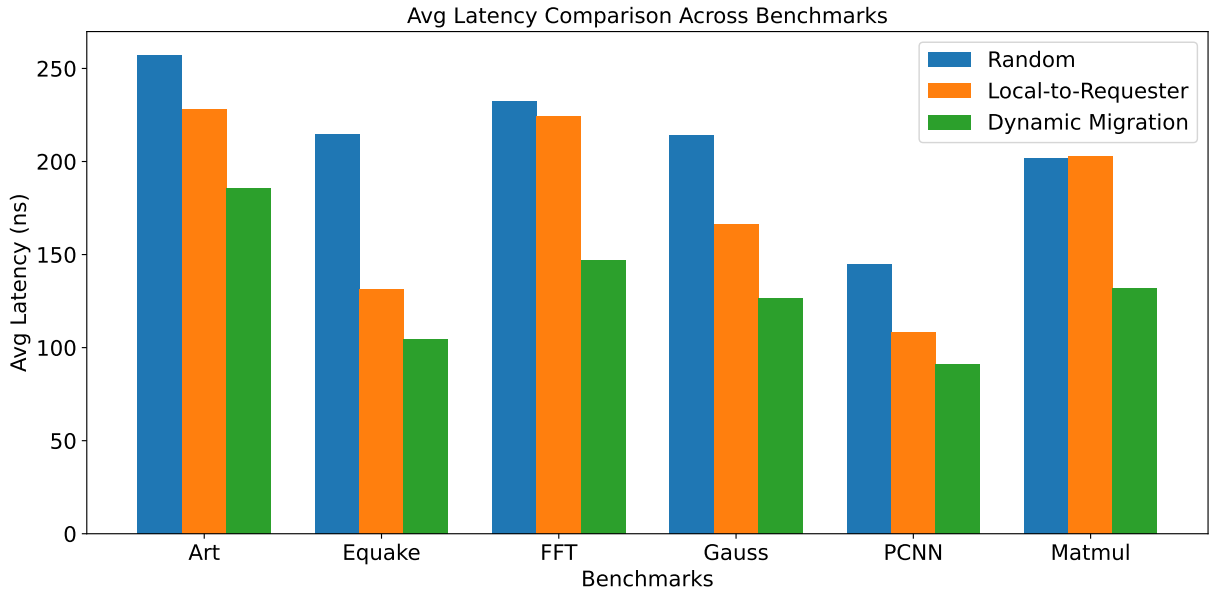

Fig. 6.1 Avg memory latency across six benchmarks using Random, Local to Requester, and Dynamic Migration mapping

6.2 Configuration Comparison

Fig. 6.2 through Fig. 6.7 present the latency achieved by all nine Dynamic Migration configurations (C1–C6), alongside Random and Local to Requester mappings for each benchmark.

Despite variations in parameter values, all dynamic configurations offer consistent improvements over the baseline mappings.

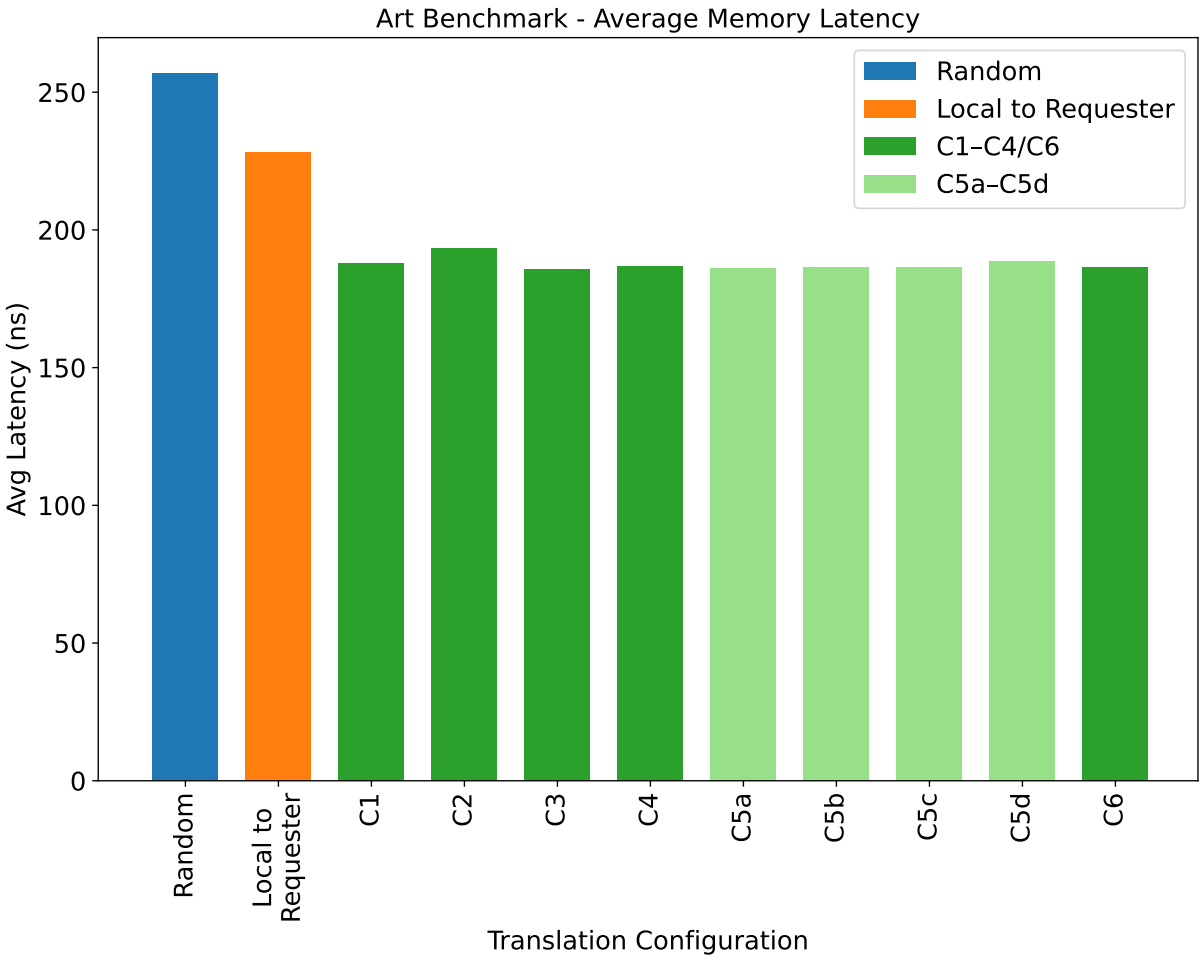

Fig. 6.2 Avg latency across all configurations for the *Art* benchmark

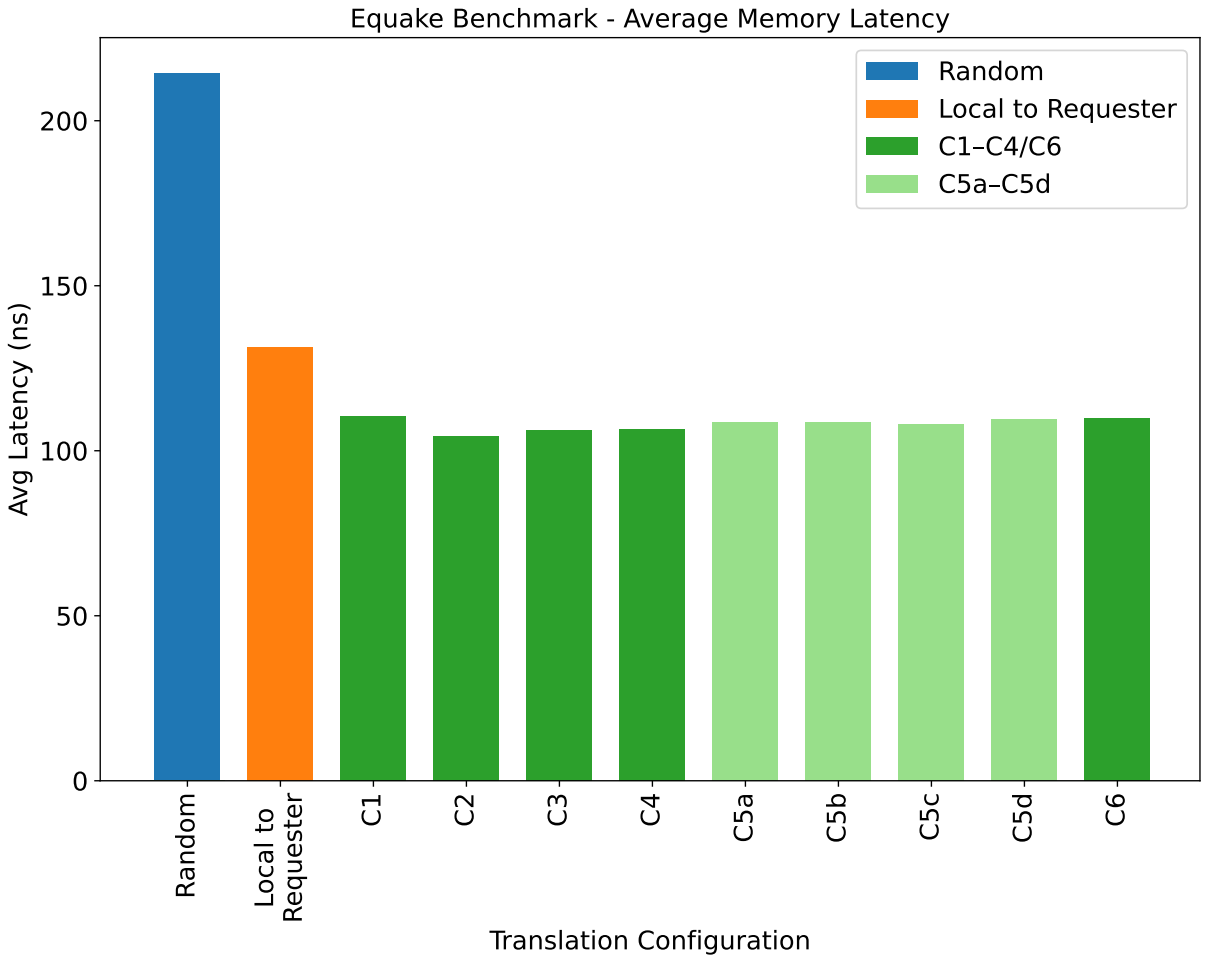

Fig. 6.3 Avg latency across all configurations for the *Equake* benchmark

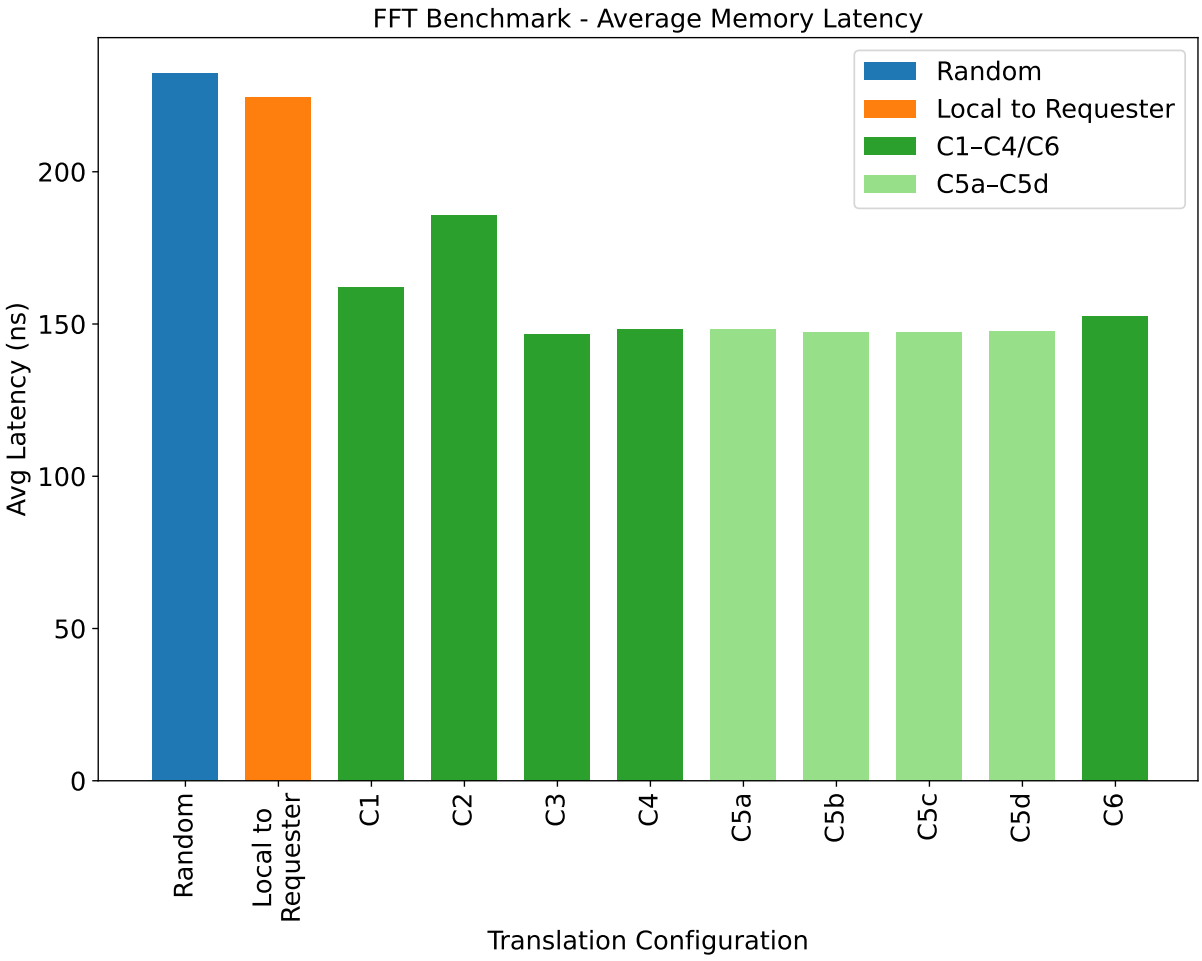

Fig. 6.4 Avg latency across all configurations for the *FFT* benchmark

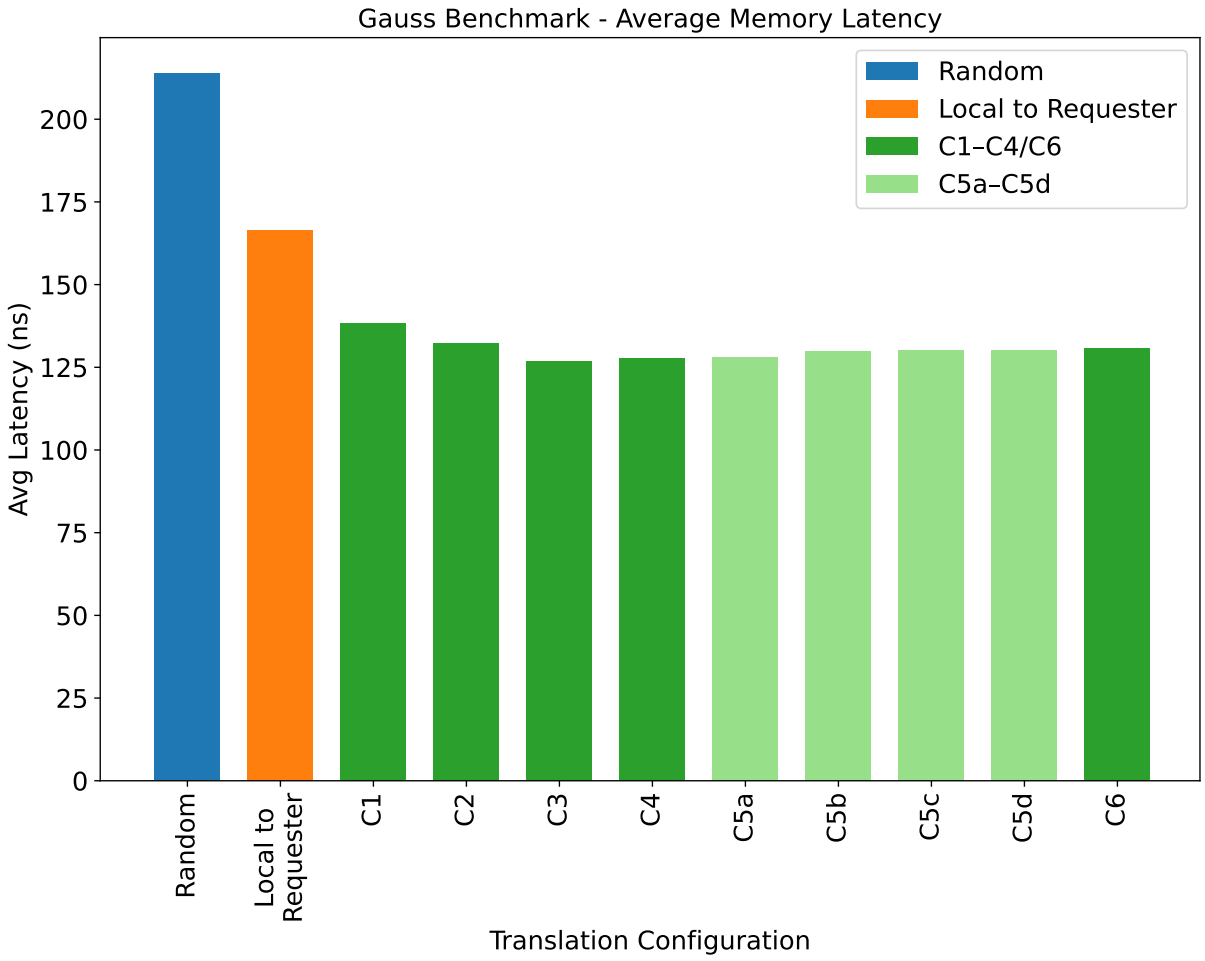

Fig. 6.5 Avg latency across all configurations for the *Gauss* benchmark

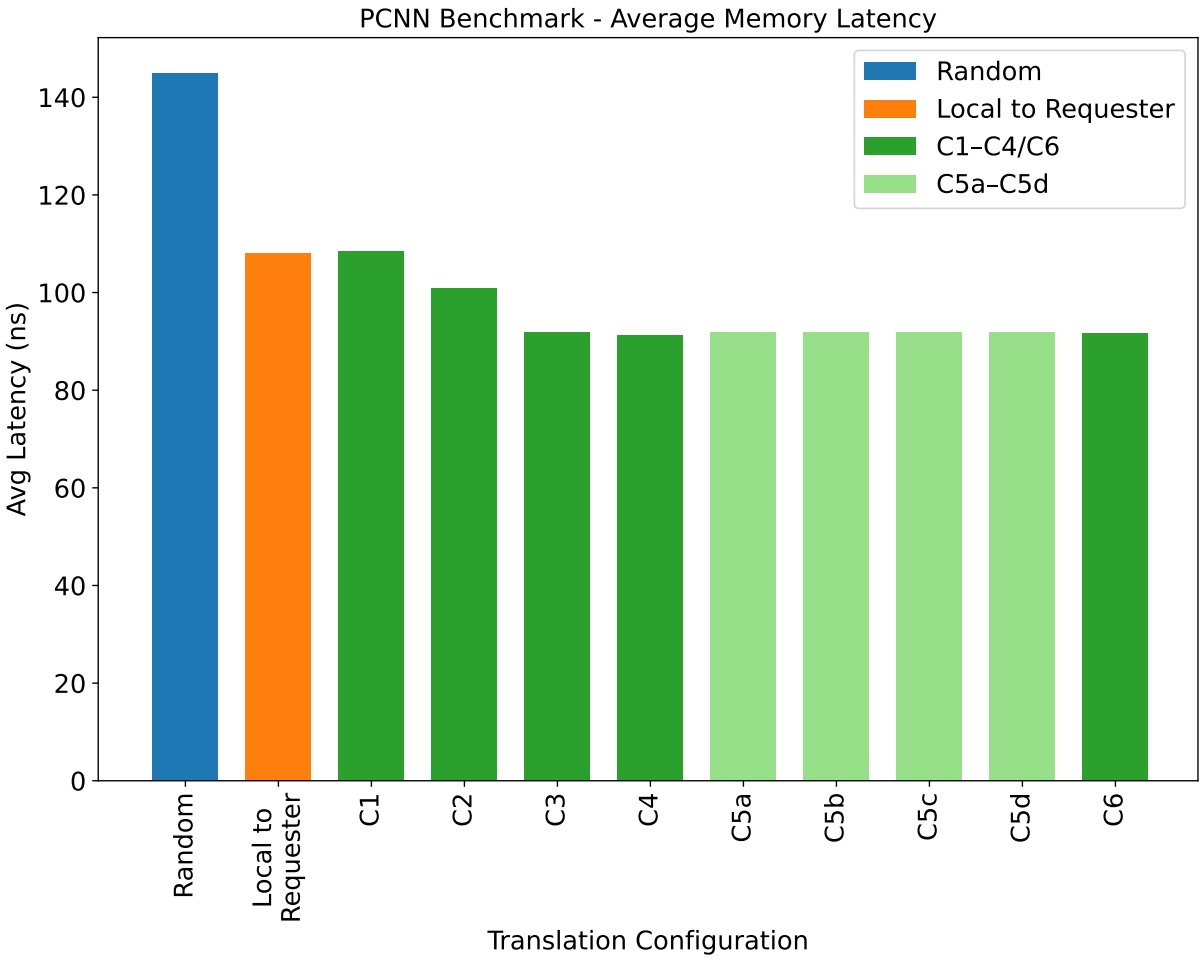

Fig. 6.6 Avg latency across all configurations for the *PCNN* benchmark

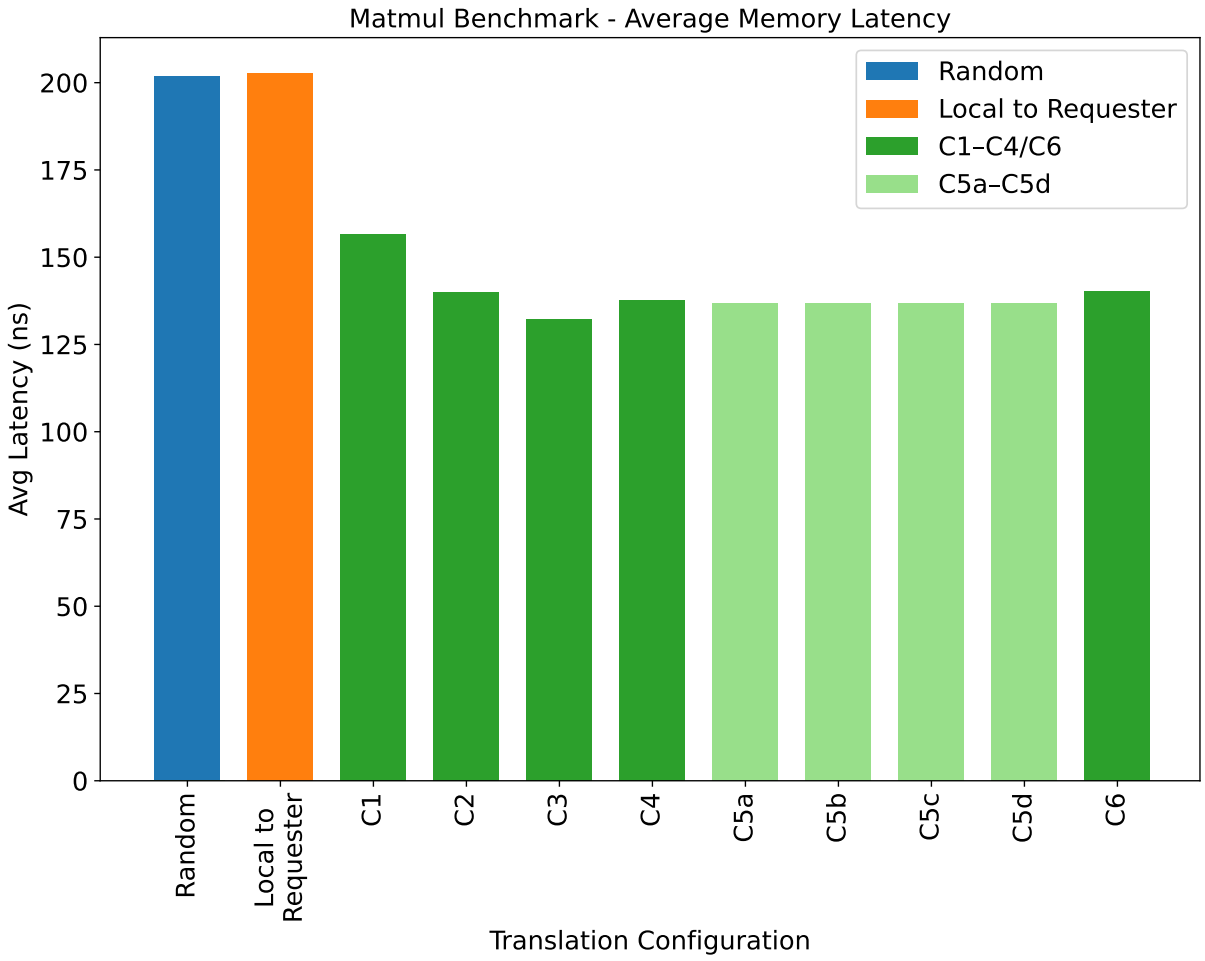

Fig. 6.7 Avg latency across all configurations for the *Matmul* benchmark

6.3 Parameter Sensitivity Analysis

This section examines how each parameter of Dynamic Migration influences memory latency. The key parameters analyzed are window size, hot page threshold, and cooldown window. Despite variations, the results demonstrate that the system remains robust across a wide configuration space.

Smaller windows (e.g. in C1–C2) respond more rapidly to shifting access patterns, enabling early identification of hot pages. However, they occasionally trigger migration

for pages that appear hot only for short time, potentially causing unnecessary remapping. Larger windows (e.g. in C6) reduce noise and improve stability, but they may miss short-lived hot pages entirely.

Lower thresholds allow more pages to be considered hot. In configurations like C1, this results in more migration opportunities. However, not all hot pages identified this way provide a migration benefit—some are short bursts with insufficient overall contribution to latency. Higher thresholds (e.g. in C6) help filter out noise but can exclude genuinely useful candidates when paired with long windows.

Introducing cooldown windows prevents rapid remigration of the same page. This makes the system more stable, especially under noisy or bursty workloads, ensuring that the system does not spend excessive time on repetitive migrations. The difference in latency between configurations like C5a, C5b, C5c, and C5d is minimal across all benchmarks, suggesting that beyond a certain point, increasing cooldown adds little value.

Dynamic Migration is robust: Even with wide parameter variation, all configurations outperform Random and Local to Requester. This confirms that the migration logic, particularly the latency gain check, ensures meaningful decisions regardless of parameter tuning.

No single configuration dominates: Moderate settings (e.g. C5a–C5d) strike a balance between responsiveness and control.

Minimal gain from fine-tuning: The latency difference among dynamic configurations is often within 2–3%, indicating the system's inherent stability and resistance to overfitting.

Chapter 7

Conclusion and Future Work

Conclusion

This thesis introduced a lightweight, translation-layer-driven memory page placement strategy named Dynamic Migration. The technique was implemented in the Ramulator2 simulator and evaluated using memory access traces generated from SniperSim. By modifying the virtual-to-physical page translation process. Results showed consistent latency reduction compared to baseline methods, with improvements ranging from 27% to over 50% in evaluated benchmarks. Which support the viability of using translation-layer mechanisms for effective memory management in NUMA environments.

Future Work

- **Adaptive Configuration**: Parameters such as window size or hot page threshold could be tuned dynamically based on bandwidth of memory of system.
- **Shared Page Replication**: Pages accessed frequently by multiple cores could be replicated across channels to reduce latency. However, this would require a page-level coherence mechanism to ensure consistency across replicas.

References

[1] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, , and O. Mutlu, "Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator," 2023.

[2] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, Aug. 2014.

[3] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in numa architectures: A statistical learning approach," pp. 1–10, 08 2019.

[4] R. Kirkpatrick, C. Brown, and V. Janjic, "Comprof and complace: Shared-memory communication profiling and automated thread placement via dynamic binary instrumentation," 11 2022.

[5] A. Panwar, *Operating System Support for Efficient Virtual Memory*. PhD thesis, Indian Institute of Science, Bangalore, 2022.

[6] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," vol. 48, pp. 381–394, 04 2013.

[7] R. Pandey and A. Sahu, "Run-time adaptive data page mapping: A comparison with 3d-stacked dram cache," *Journal of Systems Architecture*, vol. 110, p. 101798, 2020.