# Novel Affinity-Based Data Placement Mechanism for Processing-in-Memory Architectures

by

**Parker Hao Tian**

B.Sc., Simon Fraser University, 2021

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

in the
School of Computing Science
Faculty of Applied Sciences

**© Parker Hao Tian 2023**
**SIMON FRASER UNIVERSITY**
**Summer 2023**

Copyright in this work is held by the author. Please ensure that any reproduction
or re-use is done in accordance with the relevant national copyright legislation.

# Declaration of Committee

| | |
|---|---|
| **Name:** | **Parker Hao Tian** |
| **Degree:** | **Master of Science** |
| **Thesis title:** | **Novel Affinity-Based Data Placement Mechanism for Processing-in-Memory Architectures** |
| **Committee:** | **Chair:** Yuepeng Wang<br>Assistant Professor, Computing Science |
| | **Alaa Alameldeen**<br>Supervisor<br>Associate Professor, Computing Science |
| | **Arrvindh Shriraman**<br>Committee Member<br>Associate Professor, Computing Science |
| | **Zhenman Fang**<br>Examiner<br>Assistant Professor, Engineering Science |

# Abstract

Processing-in-Memory (PIM) architectures have been extensively considered to reduce costly data transfers between processors and memory. Prior PIM work proposed using processing units in the logic layer of stacked memories, e.g., Hybrid Memory Cubes (HMC), to exploit the high bandwidth between memory and the logic layer. While such approaches can improve energy efficiency and performance, they incur significant overheads due to the need to transfer data from remote memory locations to processing units where computations are performed.

In this thesis, we demonstrate that a large fraction of PIM's latency per memory request is attributed to data transfers and queuing delays from remote memory accesses. To improve PIM's data locality, we propose DL-PIM, a novel architecture that dynamically detects the overhead of data movement, and proactively moves data to a reserved area in the local memory of the requesting processing unit. DL-PIM uses a distributed address-indirection hardware lookup table to redirect traffic to the current data location. While some workloads benefit from this architecture, others are negatively impacted by the extra latency due to indirection accesses. DL-PIM uses an adaptive mechanism that considers the cost and benefit of indirection, and dynamically enables/disables it to avoid degrading workloads hurt by indirection. Overall, DL-PIM reduces average memory latency per request by 54% and improves performance by 15% for workloads that have non-trivial data reuse (6% speedup for all representative workloads).

**Keywords:** Processing-in-Memory; Computer Architecture; Computer Systems, Memory Systems; Data Locality

# Acknowledgements

To begin, I would like to first acknowledge the unceded and traditional territories of Musqueam, Squamish, Tsleil-Waututh, Katzie, Kwikwetlem, Qayqayt, Kwantlen, Semiahmoo and Tsawwassen indigenous nations and peoples where Simon Fraser University's three campuses are located, and where I have had the privilege of living, working and studying on for the past years.

I would also like to dedicate my thesis to all the people whose have lost their lives, directly or indirectly the horrific COVID-19 pandemic, as well as the healthcare workers, community workers and scientists who dedicated themselves during the pandemic.

I would like to take this opportunity to appreciate my supervisor, Professor Alaa R. Alameldeen. Your patience and generosity has guided me through the entire journey and made me a better researcher.

I would also like to thank the fellow colleagues of my lab, Mahmoud Abumandour, Marzieh Barkhordar, Brian Fu, Abdelrahman Hussein, Yonas Kelemework and Fateme Shokouhinia.

I would like to thank my friend, Mogami Nakayama (Pen name: Miao Liu) for supporting me throughout the entire journey.

I would like to thank all those who have trusted in me years ago, and made it possible for me to stay in Canada on a Humanitarian and Compassionate status. Namely, LSLAP student representatives: Alexei Parish, Issac Won, Benjemin Israel, Kaikai Zhuang, Chris Wong and Astitwa Thapa; Psychologist Dr. Helen Ferrett; Threapist Nadina Dodd, Joshua Ruberg; and finally, Senior Officer S. Wilkinson of Immigration, Refugees and Citizenship Canada, who had faith in me when I had none, and gave me a new life in here in Canada.

# Table of Contents

**Declaration of Committee** ii

**Abstract** iii

**Acknowledgements** iv

**Table of Contents** v

**List of Tables** vii

**List of Figures** viii

**1 Introduction** 1

**2 Background** 6
- 2.1 PIM and the Memory Wall . . . . . . . . . . . . . . . . . . . . . . . . 6
- 2.2 Hybrid Memory Cube (HMC) . . . . . . . . . . . . . . . . . . . . . . . 7

**3 DL-PIM architecture** 9
- 3.1 Hardware structures . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
- 3.2 Subscription protocol . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  - 3.2.1 Subscription flow . . . . . . . . . . . . . . . . . . . . . . . . 12
  - 3.2.2 Resubscription flow . . . . . . . . . . . . . . . . . . . . . . . 12
  - 3.2.3 Negative acknowledgement of subscription . . . . . . . . . . . . 12
  - 3.2.4 Unsubscription flow . . . . . . . . . . . . . . . . . . . . . . . 12
  - 3.2.5 Special cases for unsubscription . . . . . . . . . . . . . . . . 13
  - 3.2.6 Dirty bit . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
- 3.3 Memory requests . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
- 3.4 Adaptive subscription architecture . . . . . . . . . . . . . . . . . . . 14
  - 3.4.1 Hops-based adaptive policy . . . . . . . . . . . . . . . . . . . 15
  - 3.4.2 Latency-based adaptive policy . . . . . . . . . . . . . . . . . . 15
  - 3.4.3 The "Subscription Away" problem and solutions . . . . . . . . . . 16
  - 3.4.4 The "Always Unsubscription" problem and solutions . . . . . . . . 17

3.4.5 Fixing the "always unsubscription" problem with periodic subscription 17

**4 Evaluation** **20**

**5 Results and Discussion** **24**

5.1 The *Always-Subscribe* policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

5.2 Comparing *Always-Subscribe* and *Adaptive* policies . . . . . . . . . . . . . . . 26

5.3 Sensitivity to different per-hop latency . . . . . . . . . . . . . . . . . . . . . . . . 30

5.4 Sensitivity to core count . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

5.5 Sensitivity to different adaptive policies . . . . . . . . . . . . . . . . . . . . . . . 32

5.6 Sensitivity to different adaptive thresholds . . . . . . . . . . . . . . . . . . . . . 33

5.7 Sensitivity to subscription table sizes . . . . . . . . . . . . . . . . . . . . . . . . . 33

**6 Related Work** **39**

**7 Conclusions** **41**

**Bibliography** **43**

# List of Tables

Table 4.1 Workloads used in our simulation . . . . . . . . . . . . . . . . . . . . . 20
Table 4.2 Simulated Baseline System configuration . . . . . . . . . . . . . . . . . 23

Table 5.1 Baseline configuration execution cycles . . . . . . . . . . . . . . . . . . 24
Table 5.2 The top 10 access memory locations of CHABsBez and the number of accesses to those locations . . . . . . . . . . . . . . . . . . . . . . . . . 28
Table 5.3 The top 10 access memory locations of SPLRad and the number of accesses to those locations . . . . . . . . . . . . . . . . . . . . . . . . . 29

# List of Figures

Figure 1.1 Average latency per memory request across workloads from DAMOV [81] 2
Figure 1.2 Breakdown of memory latency into data transfer latency, queuing delay and array access latency . . . . . . . . . . . . . . . . . . . . . . 3
Figure 1.3 Average hops travelled per memory request. Each read request is split into a request packet, a data header packet and 4 data packets. Each write request is split into a data header packet and 4 data packets. Each packet is counted as 1 hop in this figure . . . . . . . 4
Figure 1.4 Coefficient of variation (CoV) for memory request distribution across workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Figure 2.1 Representation of an HMC system with 16 vaults from Micron's technical specification document [79] . . . . . . . . . . . . . . . . . . 8

Figure 3.1 DL-PIM hardware structures added to the logical base of each HMC vault . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
Figure 3.2 Flowchart detailing the subscription process . . . . . . . . . . . . . 19

Figure 4.1 Representation of 32 vaults in a 6 × 6 HMC network . . . . . . . . 22

Figure 5.1 Baseline MPKI without warmup . . . . . . . . . . . . . . . . . . . . . 26
Figure 5.2 Performance gain for the *always-subscribe* policy, measured by the execution cycles of the baseline divided by that of the *always-subscribe* policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
Figure 5.3 Average number of local and non-local accesses per subscription of *always-subscribe*. The blue portion indicates the average number of local accesses from the subscribed vault to a subscribed block after it was successfully subscribed. The orange portion indicates the average number of remote accesses to a subscribed block. . . . . . . . . . . 28
Figure 5.4 Performance gain of *always-subscribe* and *adaptive* normalized to the baseline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
Figure 5.5 Average latency per memory request for *always-subscribe* and *adaptive* policies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

Figure 5.6 Average total hops per memory access for *always-subscribe* and *adaptive* policies . . . 32
Figure 5.7 Breakdown of transfer, queuing and access latency in total memory latency for *always-subscribe* and *adaptive* policies . . . 33
Figure 5.8 Coefficient of variation (CoV) of the access distribution per vault for *always-subscribe* and *adaptive* policies. A high CoV implies uneven distribution where some vaults have much higher demand than others 34
Figure 5.9 Breakdown of GETS and GETX requests for CHABsBez and SPLRad 34
Figure 5.10 Average network traffic (in bytes per cycle) for *always-subscribe* and *adaptive* policies . . . 35
Figure 5.11 *Adaptive* speedup with different latency per hop . . . 35
Figure 5.12 Breakdown of memory latency into data transfer latency, queuing delay and array access latency for different latency per hop . . . 36
Figure 5.13 *Adaptive* speedup with different core number configurations . . . 36
Figure 5.14 Breakdown of memory latency into data transfer latency, queuing delay and array access latency for different core numbers . . . 37
Figure 5.15 Performance speedup of selected benchmark with different adaptive policies . . . 37
Figure 5.16 Performance speedup of selected benchmarks' adaptive policy with different adaptive thresholds . . . 38
Figure 5.17 *Adaptive* speedup with different subscription table sizes . . . 38

# Chapter 1

# Introduction

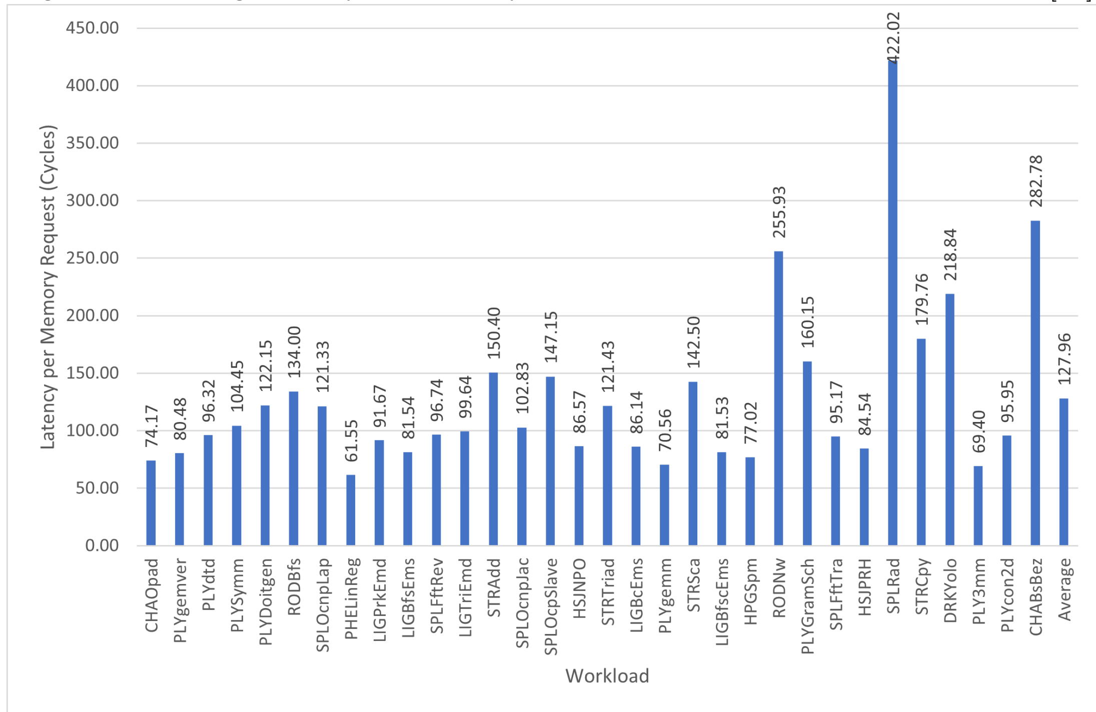
Many sectors in today's global economy are highly dependent on applications such as machine learning, data analytics, graph analytics, transaction processing, and scientific high-performance computing. Such Big Data applications have exponentially growing data needs, leading to an increasing memory footprint and higher dependence on memory speed and bandwidth [23]. In compute-centric systems, moving data between memory and processors consumes a significant amount of energy and causes performance loss, especially for workloads that have poor temporal and spacial locality. To reduce the data movement overhead, processing-in-memory (PIM) architectures have been proposed.

PIM systems utilize the high bandwidth available within memory to increase computational throughput and improve performance. By reducing the data movement overhead between processors and memory, PIM systems improve both performance and energy efficiency. For example, using 64-bit fully functional ARM-like PIM core saves 50.9% of energy consumption and improves the performance by up to 57.2% in the TensorFlow Lite workload [38].

While PIM has shown to be promising for many applications, it is limited by how the memory architecture is implemented [81]. In modern memory systems, memory is divided into different modules, ranks and banks. In Micron's Hybrid Memory Cube (HMC), a stacked memory architecture that has been extensively considered for PIM, the memory system is divided into different vaults [79]. When a PIM processing unit needs to access data from another vault, data has to be transferred between the home and requesting vaults with considerable overhead. In Figure 1.1, we evaluated many common workloads [81] that will be further discussed in Chapter 4. Using the DAMOV simulation framework [81], we show that each memory request has nearly 128 cycles of overall latency on average across all workloads. This is in contrast to the average memory array access latency of 60 cycles. Figure 1.2 shows that 53% of latency per memory request is caused by data transfers and queuing delays from remote memory accesses.

To explain data transfer latency, we show the average number of network hops traveled for each memory request across all workloads in Figure 1.3. On average, each memory

Figure 1.1: Average latency per memory request across workloads from DAMOV [81]

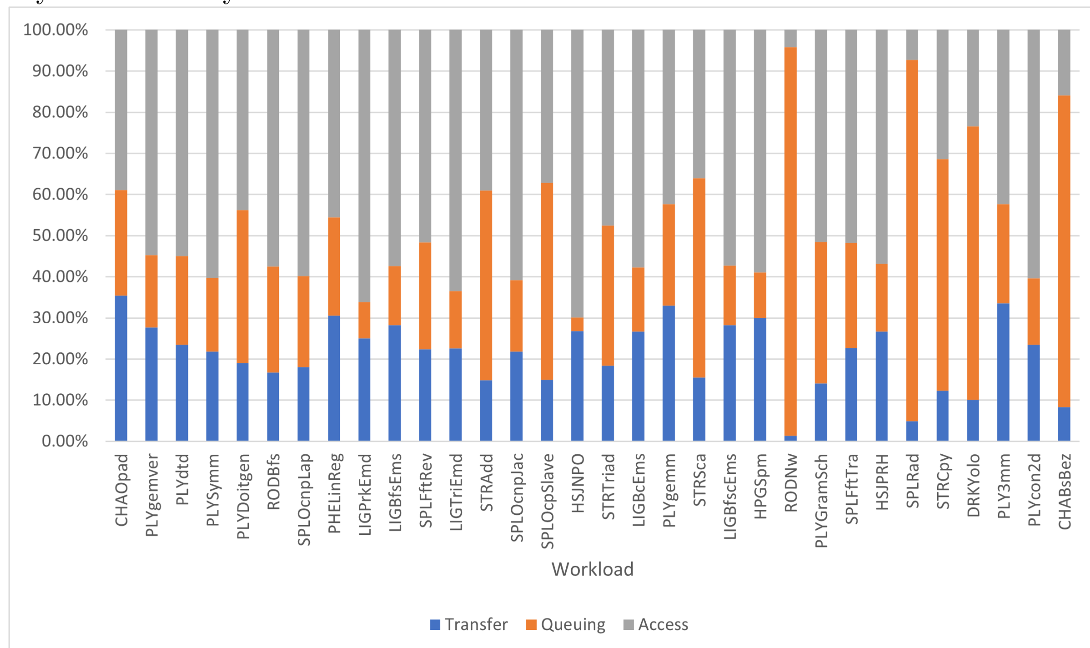



request in a $6 \times 6$ HMC memory network needs to travel 21 hops (accounting for one cycle per hop for each packet of a request).

Another major contributor to overall latency per request is queuing delay. Accesses to different vaults could be imbalanced. Memory locations in a vault can be accessed by multiple vaults at the same time. As each vault can only serve one location per cycle, the memory requests will have to be queued, which causes significant delays. Further increasing such delay is the uneven distribution of memory requests across different vaults. Consider the distribution of requests going to different vaults in a $6 \times 6$ HMC system, we measure the coefficient of variation of this distribution (standard deviation divided by mean). A high coefficient of variation (CoV) implies that a few vaults have much higher demand than the rest. Figure 1.4 illustrates that while most workloads (low coefficient of variation) have balanced access distribution across vaults, some workloads have highly imbalanced distributions indicated by a high coefficient of variation. Uneven request distributions cause significant additional queuing delays at high-demand vaults. This is illustrated in Figure 1.2 where workloads with high coefficients of variation can have 70-80% of its memory latency caused by queuing.

To reduce data transfer and queuing delays, an efficient PIM architecture should place data at the vault where it is mostly accessed. However, optimal placement is impractical since it requires a prior knowledge of which processing elements need to access different program data elements. In this thesis, we attempt to reduce data transfer and queuing delays

Figure 1.2: Breakdown of memory latency into data transfer latency, queuing delay and array access latency

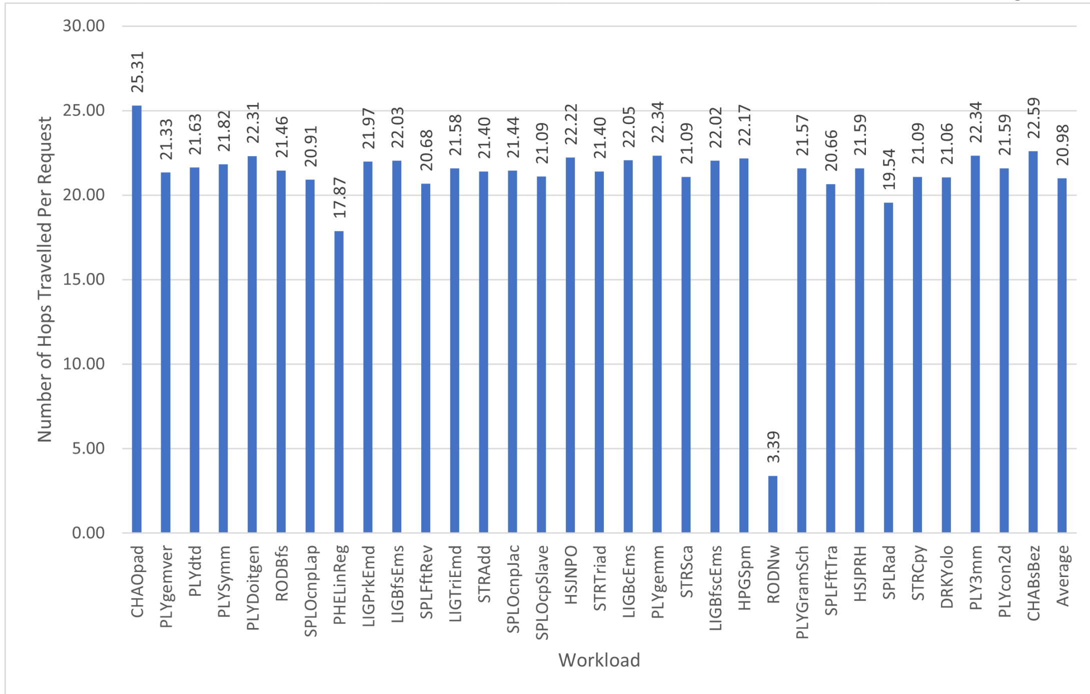



across memory modules ("vaults" for HMC) in a PIM system using a hardware indirection-based mechanism. Our architecture, Data Locality-based PIM (DL-PIM) reserves a small memory area at each vault, and dynamically moves a memory block into that reserved space if the benefit of movement outweighs the cost. We maintain a distributed hardware *subscription table* where each vault tracks local blocks that moved to remote vaults, and remote blocks that moved to the current vault. By examining runtime information across memory requests, DL-PIM can detect when subscriptions help or hurt performance of various workloads, and therefore dynamically turn subscriptions on or off.

In this thesis, we make the following main contributions:

- We demonstrate the high overhead caused by data movement *within a PIM system*, unlike other works focusing on processor-memory data movement.
- We propose an *always-subscribe* architecture that dynamically moves data to requesting memory module on first access, which helps some workloads while negatively impacting others.
- We propose an *adaptive* mechanism that turns on subscription only when it benefits a workload's performance.
- We show that DL-PIM improves average performance by 6% for all workloads, and by 15% (and up to more than 2X) for workloads that have non-negligible data reuse.

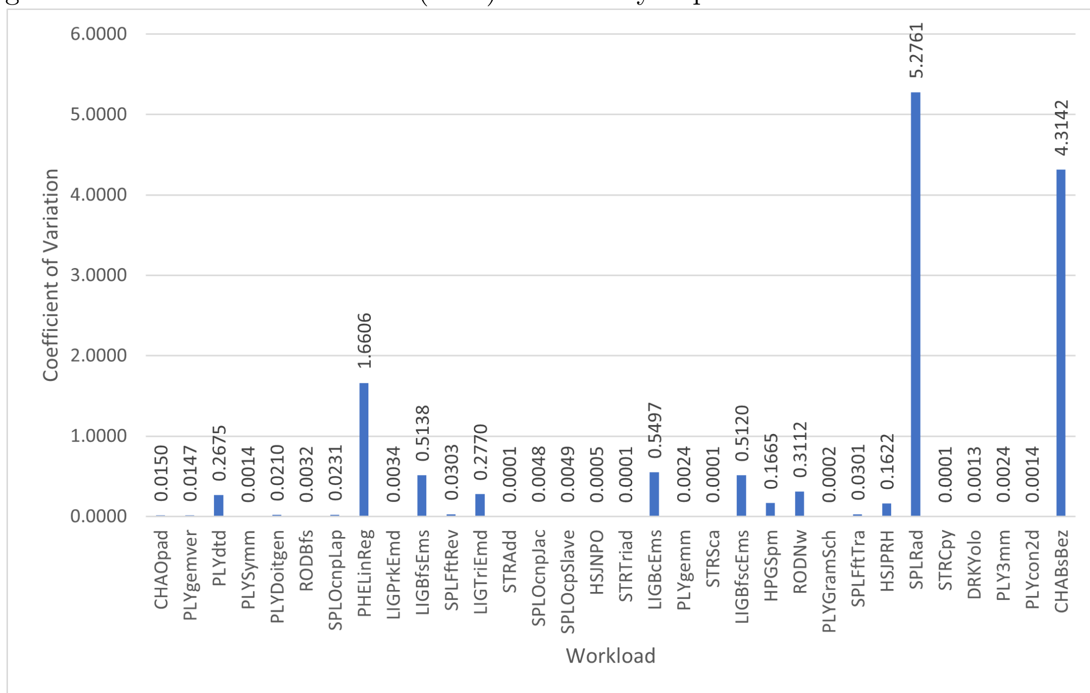
Figure 1.3: Average hops travelled per memory request. Each read request is split into a request packet, a data header packet and 4 data packets. Each write request is split into a data header packet and 4 data packets. Each packet is counted as 1 hop in this figure




This is caused by a 54% reduction in average memory latency per request for these workloads.

In Chapter 2, we will discuss the background surrounding PIM and HMC. Then, we will discuss the implementation of our architecture, including the hardware overheads and implementation flows in Chapter 3. We will provide the details about how we evaluate and set up the simulation infrastructure for our architecture in Chapter 4. We will analyze the results in Chapter 5 and provide insights behind them. Finally, we will discuss related works in Chapter 6 and conclude in Chapter 7.

Figure 1.4: Coefficient of variation (CoV) for memory request distribution across workloads



# Chapter 2

# Background

## 2.1 PIM and the Memory Wall

With the exponential improvement in CPU performance over the past few decades, memory has become the primary performance bottleneck for many important applications. Modern processors use a multi-level cache hierarchy that exploits spatial and temporal locality to reduce average memory access time. When data is reused in the cache hierarchy, the cost of data transfer to/from memory is amortized across many cache hits. As main memory is much slower than caches, cache misses incur significant performance overhead. Unfortunately, many workloads exhibit poor locality where cache data is reused infrequently, which requires high time and energy overheads to continuously transfer data from memory (where it resides) to processors (where computations are performed). This *memory wall* [78] affects processor-centric systems and degrades the performance of data-intensive workloads, e.g., database systems, machine learning and graph processing.

To reduce the performance and energy overheads of continuously moving data between processors and memory, memory-centric systems have been extensively investigated for the past few decades, gaining traction in academic research and industry. Processing-in-memory (PIM) architectures integrate computations into the memory system [89]. PIM can exploit the high internal bandwidth in memory and avoids transferring data to/from the CPU, saving time and energy.

PIM hardware implementations can be classified into two categories: processing near memory and processing using memory [38]. Both categories address the data movement bottleneck in different ways. Processing near memory architectures place processing units near memory, e.g., in the logic layer of a stacked memory technology, to provide high bandwidth and low latency communication between the processing unit and the memory. Processing using memory architectures use the architecture and properties of memory cells to allow for data operations without the involvement of a CPU or accelerator [38]. Although there are many proposals for processing using memory architectures [1, 28, 32, 93, 92, 95, 91, 47, 17, 83, 57, 19, 90, 72, 94, 5, 73, 6, 7, 8, 71, 62, 96, 63, 64, 33, 12, 49, 106, 48, 110],

they present significant complexity due to the need to modify standard memory interfaces and internal hardware. On the other hand, with the development of 3D-stacked memories like Micron's Hybrid Memory Cube (HMC) [79], processing near memory architectures [99, 3, 53, 26, 56, 58, 68, 65, 14, 15, 16, 101, 9, 10, 21, 30, 34, 35, 40, 42, 52, 51, 59, 67, 4, 76] have been gaining traction recently.

## 2.2 Hybrid Memory Cube (HMC)

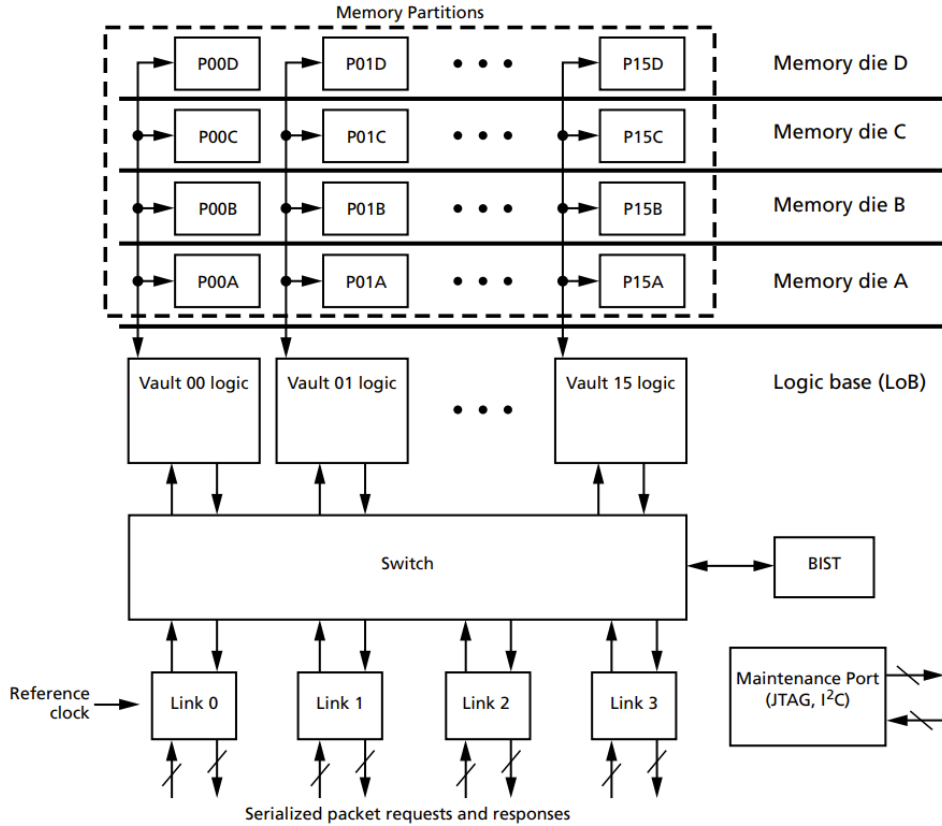
HMC utilizes the 3D-stacked through-silicon-via technology that stacks dies together. HMC divides the memory system into different *vaults*, each including one logical die and multiple memory dies. The logical die acts as a memory controller (called "vault controller"). A processing near memory architecture that places the processing units in the logical die benefits from high bandwidth and low latency accesses to memory. When reading from or writing to HMC from another vault or from another component, a different protocol is used. Instead of the traditional channel-based communication protocol, HMC utilizes a packet-based communication protocol. Each packet, i.e., *FLIT*, is 128 bits (16B) in size. HMC supports 16B, 32B, 64B or 128B memory blocks. Given that HMC would also require one FLIT to store the operation information in a header and a tail, each data access may require between 2 and 9 FLITs.

When communicating with external components, e.g., processors, HMC requires communication *links* with specifications like output buffers and input buffers, However, HMC does not define the required communication technology for internal communications and only uses crossbar switches as an example in the specification document [79]. In this thesis, we will assume that the vaults are connected in a crossbar switch network, with each vault only having input buffers of size 16 entries.

Although HMC can be used to implement a processing near memory architecture, it still faces challenging design choices. Due to the area limitations with the memory dies, each HMC vault can have only limited capacity. As a result, an HMC system would require multiple vaults interconnected with each other (usually via a crossbar switch network) to provide larger capacity. A 16-vault HMC system is illustrated in Figure 2.1. In such system, each vault acts like a router that is interconnected with the neighbouring vaults, and forwards non-local requests to the vault that is closest to its destination. Although with more vaults there are more logical dies that we could increase computational parallelism, they introduce complexities due to requiring remote accesses to data from other vaults. While each logic die has high bandwidth and low latency to the memory dies within its own vault, accesses to other vaults incur high data transfer and queuing delays, degrading performance and energy efficiency.

**Queuing Delay**. As HMC is a packet-based memory system, each vault can serve one memory request at a time. When multiple vaults need to access memory locations from a

Figure 2.1: Representation of an HMC system with 16 vaults from Micron's technical specification document [79]




vault, these requests need to be queued. HMC has I/O buffers that handle this scenario, and packets arrived have to be queued in these buffers until they reach the head of the queue.

**Non-Local Access Overhead**. We use the DAMOV [81] framework's default $6 \times 6$ network to demonstrate the network data transfer and queuing overhead across workloads. Figure 1.2 shows that data transfer and queuing latency account for 53% of memory access latency on average. To address the overhead caused by non-local memory accesses, we propose a mechanism that "attracts" data to the vault where it is likely to get most accesses. Our proposal (described in the next chapter) attempts to reduce communication (data transfer) overhead and queuing delay.

# Chapter 3

# DL-PIM architecture

Our proposal seeks to address the issues discussed in the previous chapter by implementing an architecture that not only dynamically "attracts" data to the vault where it is likely to get more accesses, but also balances the load between vaults such that the queuing delay would decrease.

## 3.1 Hardware structures

In our implementation, we added the following components (Figure 3.1) to each HMC vault's logic base to provide data transfer and reduces overhead. In the remainder of the thesis, we call such transfer a *subscription*.

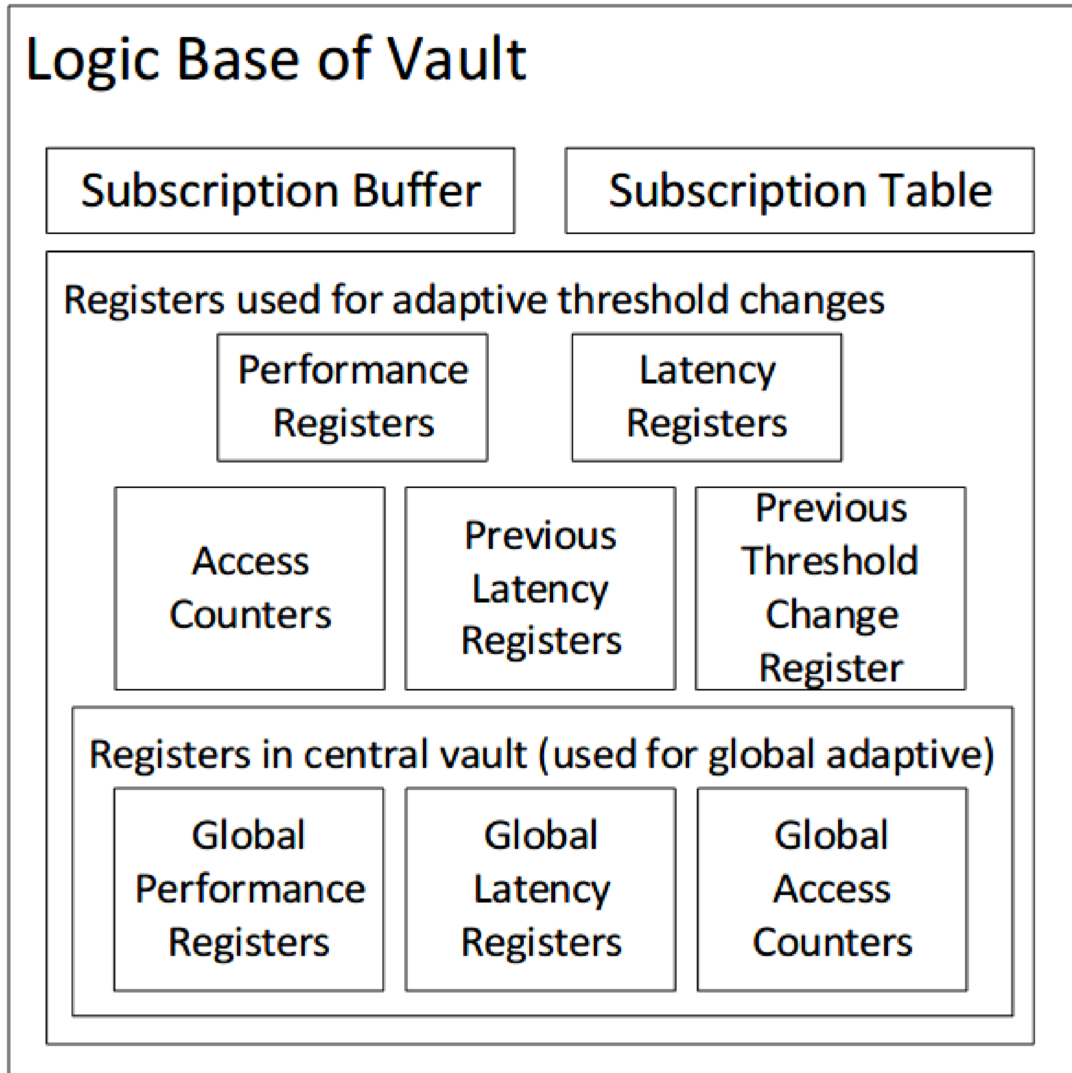
- **Subscription Table**: A cache-based hardware lookup table.
- **Subscription Buffer**: A fully-associative cache.
- Registers to accumulate the performance for each PIM core to dynamically turn subscriptions on/off.
- Reserved space in memory to hold subscribed data.

For the *central vault* in our *global adaptive* policy (details will be discussed later), we also added registers to record the performance for the entire system.

We modify the HMC packet transfer protocol, and add a new field to allow for the transfer of subscription request both alongside and separate from regular data access requests.

At early stages of our implementation, we have considered using a subscription threshold-based policy in which we have a count table to record the number of accesses to each address. The count table is a direct-mapped cache with 8192 entries and 32-bits per entry, split between an 8-bit counter and a 24-bit tag. In each data access, we calculate a tag using the address of the block, and compare it with the tag of the corresponding entry in the count table. If the tag does not match, we reset the counter to 0 and update the tag

Figure 3.1: DL-PIM hardware structures added to the logical base of each HMC vault




with the current tag, by which we evict current entry and replace it with the incoming entry. If the number of the count is greater than the threshold and the access is not local, the architecture will trigger the subscription routine. However, during our experiments, we discovered that almost all subscription-friendly workloads benefit from a 0-count threshold subscription policy in which we subscribe to an address on first access. Therefore, the count table is unnecessary and is not included in DL-PIM.

**Subscription Table** We use a cache-based lookup table called *Subscription Table* which maps the original address of a data block to the current address with each subscription. The subscription table is a 4-way set-associative cache with 2048 sets per table per vault. Therefore, each vault has 8192 entries. Each entry has two addresses: the original address and one subscribed address. Additionally, each entry also has three state bits that indicate any of the following states:

- Invalid (or "Unsubscribed")
- Pending Subscription
- Subscribed
- Pending Resubscription
- Pending Unsubscription

We discuss the use and meaning of each of these states in the next few subsections.

**Subscription Buffer.** As our subscription table has limited size, we only allow for a subscription if space in the subscription table is available. When the subscription table is full, we utilize least-frequently used (LFU) algorithm to locate the least frequently or least recently used entry as our victim for *unsubscription*. However, the unsubscription does not take effect instantly, and would require communication over the network. As such, the buffer is used to temporarily store our request pending the completion of unsubscription.

The subscription buffer is a 32-entry fully-associative cache with an entry for subscription request, with from vault, to vault, subscription address, and other information. The exact format of a subscription request will be provided in the following section. Each subscription buffer entry has a valid bit which will be set when the corresponding subscription table set has available space. In each cycle, we attempt to process a valid subscription request, if any, based on valid bits.

## 3.2 Subscription protocol

We implement a packet-based subscription protocol. Each subscription packet includes the following information:

- From vault: The vault that originated the request
- To vault: The destination vault for the request
- Address: The memory address of the requested block
- Request type: Subscription Request, Subscription Request Negative Acknowledgement, Subscription Data Transfer, Subscription Transfer Acknowledgement, Unsubscription Request, Unsubscription Transfer Acknowledgement, Turn On Subscription, Turn Off Subscription
- Dirty bit: Whether the data being unsubscribed or resubscribed has been modified since subscription

### 3.2.1 Subscription flow

When an address is accessed, the vault that the *subscription block* will be subscribed to (called "requester vault") will first check if it has the space to subscribe such block, and push the subscription request into the subscription buffer if it requires unsubscription to free up space. Then, it will push a packet with the address and send it to the home vault which originally held such block (called "original vault"), and update the subscription table of the corresponding address with the *Pending Subscription* state. Upon receiving such request, the original vault will perform the same check, ensure the address is not subscribed or pending subscription/unsubscription to/from another vault, and update the state to *Pending Subscription*. Then, it will start the transfer of the subscribed data. The requester vault will then acknowledge such transfer. When both sides acknowledge, the subscription table entry is marked as *Subscribed*.

### 3.2.2 Resubscription flow

Our protocol also supports a special case of subscription called *Resubscription* which occurs when a vault requests subscription for a data block that is already subscribed to another location. In that case, we still process the subscription, but the original vault will redirect this request to the vault currently holding the memory block (called "subscribed vault"). The subscribed vault will then change the state of its subscription table entry to *Pending Resubscription* and start sending the data to the requester vault. Upon receiving data, the requester vault will send two acknowledgements, one to the original vault (so it modifies its subscription table and change the entry to the new subscribed address), and one to the subscribed vault (so it can evict such entry from its table).

### 3.2.3 Negative acknowledgement of subscription

In some cases, a subscription request cannot be completed successfully. The subscription table has a limited size, and we would utilize the subscription buffer to temporarily hold the address while we make space. However, the subscription buffer is also limited in size, and in the case that the subscription buffer is also full, we cannot complete the subscription at all. In that case, the original vault would send a subscription negative acknowledgement to the requester vault, and the requester vault will then just rollback the subscription by removing the subscription entry. A similar situation would happen when the original vault (or subscribed vault, in the case of resubscription) is currently in the process of subscribing the address block to another vault, or unsubscribing the address block from another vault.

### 3.2.4 Unsubscription flow

Unsubscription is triggered when the subscription table of a given vault is full and requires some free space. However, there is a special case when the requester vault is the original

vault, so we cannot redirect the subscription request back to the original vault, and as such we change the subscription request to unsubscription.

When unsubscription happens, it can be initiated by either the original vault (when it wants the data back) or the subscribed vault (when it wants to push the data back to the original vault). When the original vault is initiating, it will send an unsubscription request to the subscribed vault. When the subscribed vault initiates such process, it can skip this step. Then, the subscribed vault will mark the subscription table's corresponding entry as *Pending Unsubscription*, and start transferring the data back to the original vault. Upon receiving such data, the original vault will mark it as *Unsubscribed*, and acknowledges it to the subscribed vault, which will also mark the entry as *Unsubscribed*.

In case an unsubcription request is sent for an address in the process of subscription, we wait until the subscription flow finishes before we proceed with unsubscription. If an unsubscription request is sent for an address in the process of resubscription, we wait for the resubscription to complete and forward the unsubscription request to the requester vault to ensure data consistency.

### 3.2.5 Special cases for unsubscription

As with subscription, there are special cases of unsubscription. When an address is in the process of being subscribed, we wait until the subscription finishes before proceed with the unsubscription. As for the resubscription, when it happens, we not only wait for the resubscription to happen, but also forward the unsubscription request to the requester vault in this case, to ensure data consistency. The wait should be no longer than the time it takes for the resubscription to finish.

### 3.2.6 Dirty bit

To reduce the data transfer required for unsubscription, we add a dirty bit to our subscription reserved space, which is set when the block is written. When processing unsubscription, we check if the dirty bit is currently set. If the dirty bit is reset, we only transfer an acknowledgement packet instead of the full data, as it would already exist in the original vault. This bit would also be forwarded in the case of resubscription using the dirty bit in the request packet.

## 3.3 Memory requests

Unlike the baseline HMC memory, our architecture uses the subscription table to provide dynamic address translation to any subscribed address. However, as our subscription table is distributed across all vaults, any memory request requires the vault requesting memory access ("requester vault") to access both the home vault that originally held the block ("original vault") and the vault that currently holds the most recent copy ("subscribed

vault"). We here will discuss the process of serving both the read and write requests in the baseline HMC implementation and our architecture. We denote the Manhattan distance between the original vault and requester vault as $h_{ro}$, between the requester vault and the subscribed vault as $h_{rs}$, and between the subscribed vault and the original vault as $h_{so}$.We denote the size of a data block as $k-1$ flits, which makes a data transfer packet $k$ flits (since one flit is used as a header for routing and other information). Our memory access protocol is inspired by the SGI Origin system [66] based on the DASH protocol [70]. Our protocol applies to the reserved area in each vault that contain subscribed blocks, as opposed to caches in Origin.

**Read Requests.** When serving a read request in the baseline HMC implementation, the requester vault would send a request to the original vault. When the original vault receives such request, it would transfer the data back to the requester vault. Such access requires $(k+1)h_{ro}$ cycles of network overhead assuming a single cycle latency per hop.

In our architecture, we first check if the requested address is currently in the local reserved subscription space. If so, our access is a local access and has no network overhead. If not, we send the read request to the original vault since the requester vault does not have an up-to-date subscription information for the requested block. The original vault would check its subscription table to identify the subscribed vault (if any). If the block is not subscribed to another vault, the original vault sends the data to the requester. If the block is subscribed, the original vault forwards the request to the subscribed vault, which transfers the data back to the requester vault. In total, a read access needs $h_{ro} + h_{so} + kh_{rs}$ cycles of network overhead.

**Write Requests.** In the basline HMC, the requester vault writes directly to the original vault, and such operation would have $kh_{ro}$ cycles of network overhead. In DL-PIM, we still write the data to the original vault in case of remote access, and let the original vault forward the written data to the subscribed vault. In the case of local access, the network overhead is 0.

## 3.4 Adaptive subscription architecture

Our implementation supports the binary *always-subscribe* and *never-subscribe* configurations. While many workloads benefit from proactively subscribing and moving data to the requester vault, others are hurt by the extra latency and traffic caused by indirection requests described in the previous section. Therefore, we also implemented and evaluated a few adaptive strategies to dynamically turn subscriptions on and off. Our adaptive architectures focused on these two properties: number of hops travelled and access latency of each request. We aggregate these statistics in hardware registers as shown in Figure 3.1. Initially, we considered a policy where we continuously kept track of the cost and benefit of subscription in a *feedback register* (discussed below), and broadcast the policy change

to every vault once the feedback register changes from positive to negative or vice-versa. However, this has proven to be inefficient, as our subscription policy would change rapidly, and broadcasting the new subscription policy to all vaults incurs high overhead. Therefore, we opted to divide the execution into "epochs", and make decisions using the aforementioned information at the end of each epoch, which is every $10^6$ (one million) cycles in our implementation. At the end of each epoch, a decision about the subscription policy for the next epoch is broadcast, and the information stored in aggregate registers is then cleared so each epoch's decision is only based on the previous epoch's cost/benefit behavior.

We also explored to allow for a "sliding" subscription policy in which we have a sliding number of access threshold when subscription happen. However, as discussed in Section 3.1, this count threshold implementation is proven to be inefficient, so we are not considering this adaptive mechanism.

### 3.4.1 Hops-based adaptive policy

Since our architecture attempts to reduce data movement across the HMC network, the number of hops per packet is a key metric to consider. To assess whether subscriptions are helping or hurting traffic, we use a register called *feedback register* at each vault. We measure the number of hops travelled by each packet, and using the vault's address to estimate the number of hops travelled if the requested address is not subscribed. We then compare the two hops' counts. If the estimated original hop is higher than the actual number of hops when subscribed, it means that our subscription helps reduce the distance traveled for that packet, and we therefore increment the feedback register. Otherwise, if the subscribed hops is higher, it means that our subscription is increasing the distance traveled per packet, and we therefore decrease the feedback register. As such, a positive feedback register value means that the workload is benefiting from subscription, while a negative value means that the workload is negatively impacted. In the first epoch, we turn on subscription across all vaults and collect the cost/benefit information in each vault's feedback register. At the end of each epoch, if the feedback register is negative, we turn off the subscription for the next epoch; otherwise we turn it on.

### 3.4.2 Latency-based adaptive policy

A more accurate predictor of performance is the access latency per request. This includes all latency components including transfer (communication) and queuing delays. While estimating hops without subscription is straight-forward as discussed above, the latency is dependent on factors other than hops travelled so we can't easily estimate the latency for a memory request without subscription. To address this problem, we used a different strategy to make decisions about subscriptions. At the completion of each request, we record its latency by adding it to a *latency register* (Figure 3.1). We also record the number of requests served in a given epoch in another register called *request register*. At the end of

each epoch, we calculate the average latency for all memory requests in this epoch. In the initial epochs we use the hops-based feedback register to decide the subscription policy, and aggregate the epoch's request latency into the latency register. For each epoch afterwards, we decide whether the average latency per memory request has increased or decreased by a certain threshold compared to the latency from the previous epoch (stored in a *previous latency register*). If average latency is lower or within the threshold, we continue with the same subscription policy in the next epoch. If the per-request average latency has increased beyond the threshold, we reverse the decision and enable or disable subscription in the next epoch. We have tried different thresholds and have found that a 2% threshold is heuristically the best option for most workloads. We therefore use a 2% latency change threshold for the latency-based adaptive policy.

### 3.4.3 The "Subscription Away" problem and solutions

One downside for both hops-based and latency-based adaptive policies is the "subscription away" problem. While the *feedback register* mechanism measures the benefit of subscription to a given vault, it does not take into account the impact on the original vault that included the data. The original vault is negatively impacted as the block has been "subscribed away" from it. This is not measured by the previous feedback register mechanism since it only updates the register of the accessing vault. To address this problem, we not only update the feedback register of the accessing vault, but also update the feedback register of the subscribed vault when the memory location is subscribed and the feedback is negative (i.e. when the subscription is causing the memory request to travel more hops). This addresses "subscription away" problem by recording the negative feedback of such subscriptions in the feedback register.

The latency-based policy suffers from the same "subscription away" problem since one vault may benefit significantly from subscription with cost incurred by other vaults. From the subscribed vault's perspective, average latency is lower since the subscribed block incurs only the local access latency; but other vaults could incur communication and queuing delays. However, this negative impact on other vaults is not measured by the feedback mechanism of the subscribed vault. With no negative feedback recorded, all vaults would opt for the greedy policy of *always-subscribe* which may cause thrashing and an overall performance and energy degradation. To address this problem, we need to make decisions at a global level for all vaults. Instead of each vault individually deciding whether to subscribe or not for the next epoch, each vault will send a special packet to the vault at the centre of the network, called "central vault" before an epoch ends, e.g., at 90% of an epoch, or $9 \times 10^5$ cycles into an epoch. Then the "central vault" will calculate the global per-request average latency using individual vaults' aggregate latency and request counts, and decide on whether subscription has been helping or not. Finally, the central vault will broadcast this decision to every vault in the network before the beginning of the next epoch. Since the

central vault's computation is complex, it needs to execute in software and would therefore incur some latency (we estimated it to be 1000 cycles) before the new global policy is in effect across all vaults after the beginning of an epoch.

### 3.4.4 The "Always Unsubscription" problem and solutions

Another problem with the initial adaptive policy is that we can only compute the number of hops for subscribed requests when subscription is turned on. When subscriptions are turned off, we cannot determine when we enter a phase where subscriptions would help performance since no subscription benefit can be recorded, which means that the adaptive policy will be stuck at the *never-subscribe* policy. This issue is slightly better for the latency-based adaptive policy, in which the latency would be higher than the previous epoch when if the policy change towards *never-subscribe* is incorrect. However, when the program enters a phase where it benefits from subscription after a phase where it doesn't benefit (or vice versa), the initial adaptive policy is hard to adapt.

To address this problem, we implemented a dynamic set sampling mechanism similar to the one proposed by Qureshi, et al. [86]. As our implementation includes a cache-like set-associative subscription table, we select two *leading sets*. In one leading set, subscriptions are always turned on; while it is always turned off in the other leading set. We record the feedback and latency for each of the leading sets separately. At the end of each epoch, we compare hops and/or average latency for the leading sets to determine whether subscription is beneficial. If one leading set has lower latency or higher feedback, we will use that leading set's policy for every other set in our subscription table in the next epoch.

While this implementation solves the "always unsubscription" problem, it has a potential drawback. With some memory locations always in the *never-subscribe* leading set, memory accesses to those locations would not benefit from subscription. This can significantly degrade performance for workloads where subscription causes demand across vaults to be evenly distributed. For example, CHABsBez and SPLRad which benefit most from DL-PIM (as discussed more in detail in Chapter 5) by smoothing the memory acccess distribution across vaults. As the placement of memory location is hard to predict at runtime, it is possible that the most accessed locations are within the *never-subscribe* set, and as such, our architecture would have no benefit.

### 3.4.5 Fixing the "always unsubscription" problem with periodic subscription

Another way to address the "always unsubscription" problem is to simply add another counter to our implementation. This counter will increase by one at the beginning of each epoch the subscription is turned off, and reset to 0 whenever we are re-enabling subscription. If the counter is greater than 10, it means that we have completed 10 epochs without

subscription, and therefore we turn the subscription back on to see if we are entering a region where subscription would be beneficial.

This implementation also suffers from performance issues as dynamic set sampling. However, different from dynamic set sampling, it is workloads which subscription harms that would have most performance degradation. As our architecture would enter "always subscription" at least once every $10^7$ (10 million) cycles, workloads that do not benefit from subscription would have some subscription during their executions, and as a result, perform worse under this policy.

As latency is easier to capture in real hardware using clock, we are using latency-based adaptive policy as our baseline adaptive policy, and we use global adaptive to avoid the "subscription away" problem. While "always unsubscription" is a big issue in real hardware, as our workloads have consistent data access pattern and both of the aforementioned solutions have performance costs, we are using the adaptive policy without set sampling or periodic subscription in our evaluation.

Figure 3.2: Flowchart detailing the subscription process

# Chapter 4

# Evaluation

We use the DAMOV simulation framework [81] to implement and evaluate our proposed architecture. DAMOV integrates the ZSim [88] CPU simulator and the Ramulator [61] memory simulator. It simulates a configurable number of traditional CPUs or Processing-in-Memory cores with HMC Memory. We simulate an inter-vault network model for a $6 \times 6$ inter-connected memory network with 32 vaults (Figure 4.1). We instrumented DAMOV to dynamically analyze inter-vault traffic and overhead in a distributed PIM system.

In our evaluation, we configure a baseline system similar to DAMOV [81] that has 2.4Ghz PIM cores with 32KB L1 cache directly connected to a 4GB HMC memory system. Since the default memory configuration for network analysis in DAMOV has 32 vaults, we opted to configure 32 cores for our default evaluation configuration which is illustrated in Figure 4.1.

The DAMOV benchmark suite [81, 46] included over 300 applications from different domains such as big data and machine learning. We use the 44 *representative applications* identified by DAMOV for our evaluation. Those representative applications are from benchmark suites Chai [45], Darknet [87], Hashjoin [11], HPCG [27], Ligra [97], PARSEC [13], phoenix [108], PolyBench [84], Rodinia [20], SPLASH2 [105] and STREAM [77]. Among these applications, seven were too short to provide useful insights and five had compatibility issues with our simulation environment. In the next chapter, we present results from 32 representative applications and focus on 13 that have non-negligible data reuse for most of our analysis. A full list of all workloads is shown in Table 4.1 [81].

Table 4.1: Workloads used in our simulation

| Suite | Benchmark | Function | Short Name |
|---|---|---|---|
| Chai | Bezier Surface | Bezier | CHABsBez |
| | Padding | Padding | CHAOpad |
| Darknet | Yolo | gemm_nn | DRKYolo |

Continued on next page

Table 4.1: Workloads used in our simulation (Continued)

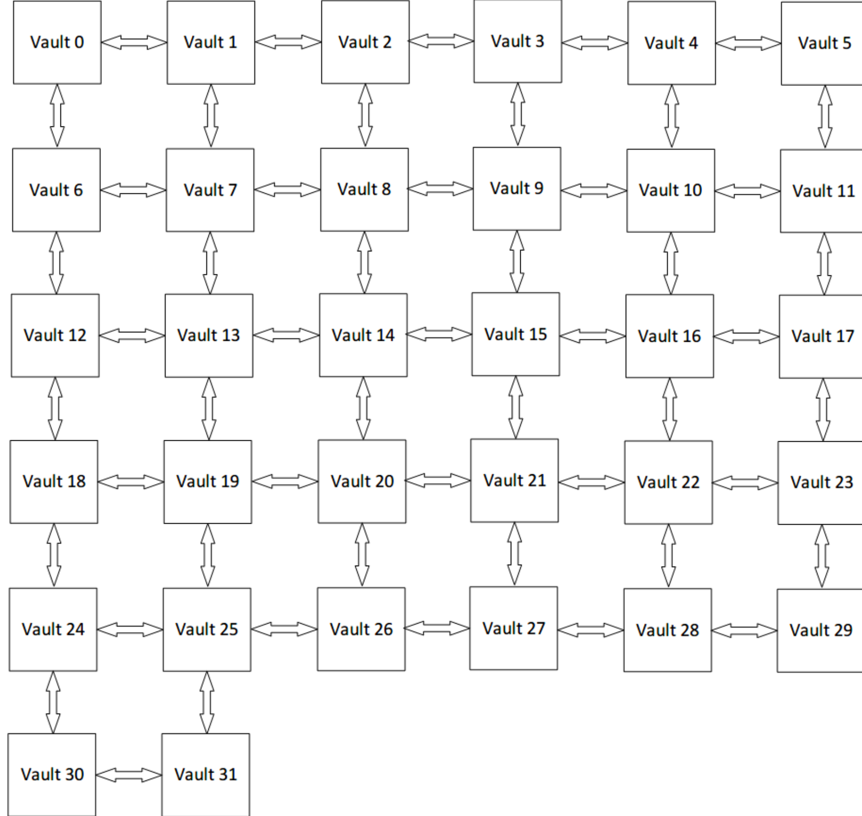
| | | | |
|---|---|---|---|
| Hashjoin | NPO | ProbeHashTable | HSJNPO |
| | PRH | HistogramJoin | HSJPRH |
| HPCG | HPCG | ComputeSPMV | HPGSpm |
| Ligra | Betweenness Centrality | EdgeMapSparse (USA) | LIGBcEms |
| | Breadth-First Search | | LIGBfsEms |
| | BFS-Connected Components | | LIGBfscEms |
| | PageRank | EdgeMapDense (USA) | LIGPrkEmd |
| | Triangle | EdgeMapDense (Rmat) | LIGTriEmd |
| Phoenix | Linear Regression | linear_regression_map | PHELinReg |
| PolyBench | Linear Algebra | 3 Matrix Multiplications | PLY3mm |
| | | Multi-resolution analysis kernel | PLYDoitgen |
| | | Matrix-multiply C=alpha.A.B+beta.C | PLYgemm |
| | | Vector Mult. and Matrix Addition | PLYgemver |
| | | Gram-Schmidt decomposition | PLYGramSch |
| | | Symmetric matrix-multiply | PLYSymm |
| | Stencil | 2D Convolution | PLYcon2d |
| | | 2-D Finite Different Time Domain | PLYdtd |
| Rodinia | BFS | BFSGraph | RODBfs |
| | Needleman-Wunsch | runTest | RODNw |
| SPLASH2 | FFT | Reverse | SPLFftRev |
| | | Transpose | SPLFftTra |
| | Oceanncp | jacobcalc | SPLOcnpJac |
| | | laplaccalc | SPLOcnpLap |
| | Oceancp | slave2 | SPLOcpSlave |

Continued on next page

Table 4.1: Workloads used in our simulation (Continued)

| | | | |
|---|---|---|---|
| | Radix | slave_sort | SPLRad |
| STREAM | Add | Add | STRAdd |
| | Copy | Copy | STRCpy |
| | Scale | Scale | STRSca |
| | Triad | Triad | STRTriad |

Figure 4.1: Representation of 32 vaults in a $6 \times 6$ HMC network




Due to DAMOV being an integrated framework with ZSim simulating the CPU and Ramulator simulating the memory, we cannot easily obtain the number of instructions executed on Ramulator, where we implemented DL-PIM. Therefore, we decided to warm up our simulation based on the number of memory requests. We executed the simulations with $10^6$ memory requests of warm up. Also, the system requirements of ZSim and Ramulator require us to run simulations on a virtual machine with the configuration detailed in Table 4.2.

Since different workloads need widely different execution cycles to complete (Table 5.1), we present most results using the performance gain metric, computed by dividing the baseline execution time (in cycles) by our architecture's execution time. A performance gain that

Table 4.2: Simulated Baseline System configuration

| | |
|---|---|
| Operating System | Ubuntu 18.04 x86-64 running on QEMU 4.2.1 |
| CPU | two Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz |
| Memory | 128GB total size; HMC v2.0; 32 vaults; 8 DRAM banks/vault; 256B row buffer; DRAM@166 MHz; 8B burst width at 2:1 core-to-bus freq. ratio; Open-page policy; HMC default interleaving [81] |

is higher than 1 indicates our architecture is improving performance, and below 1 indicates performance degradation. Actual runtimes can be computed using the baseline numbers in Table 5.1.

# Chapter 5

# Results and Discussion

We simulated the baseline configuration without any subscription and the *always-subscribe* configuration where we always subscribe on first access to a memory block. Figure 5.1 shows the average absolute misses per thousand instructions for all workloads. We show normalized performance gain for the remainder of this chapter. As shown in Figure 5.1 and Table 5.1, each workload has different memory access patterns and execution characteristics, and therefore performs differently with our architecture.

Table 5.1: Baseline configuration execution cycles

| Workload | Average Execution time (Cycles) |
|---|---|
| CHAOpad | 794,386,166.4 |
| PLYgemver | 23,700,267.8 |
| PLYdtd | 12,994,520.6 |
| PLYSymm | 101,468,991.6 |
| PLYDoitgen | 62,169,710.8 |
| RODBfs | 18,475,108.2 |
| SPLOcnpLap | 76,855,633.2 |
| PHELinReg | 36,723,166 |
| LIGPrkEmd | 58,343,807.4 |
| LIGBfsEms | 38,385,511.6 |
| SPLFftRev | 29,429,555.8 |
| LIGTriEmd | 48,599,133.2 |
| STRAdd | 59,096,723.2 |
| SPLOcnpJac | 34,626,366.6 |

Continued on next page

Table 5.1: Baseline configuration execution cycles (Continued)

| | |
|---|---|
| SPLOcpSlave | 36,668,710.2 |
| HSJNPO | 36,084,341.2 |
| STRTriad | 61,050,112.8 |
| LIGBcEms | 41,995,613.4 |
| PLYgemm | 67,804,009.4 |
| STRSca | 67,526,104.6 |
| LIGBfscEms | 38,507,868 |
| HPGSpm | 25,567,980 |
| RODNw | 227,088,329 |
| PLYGramSch | 92,543,354.2 |
| SPLFftTra | 28,601,089.2 |
| HSJPRH | 18,716,494.6 |
| SPLRad | 157,606,333.6 |
| STRCpy | 51,808,295 |
| DRKYolo | 30,656,832.4 |
| PLY3mm | 68,038,019.2 |
| PLYcon2d | 9,358,548.8 |
| CHABsBez | 30,089,343.4 |

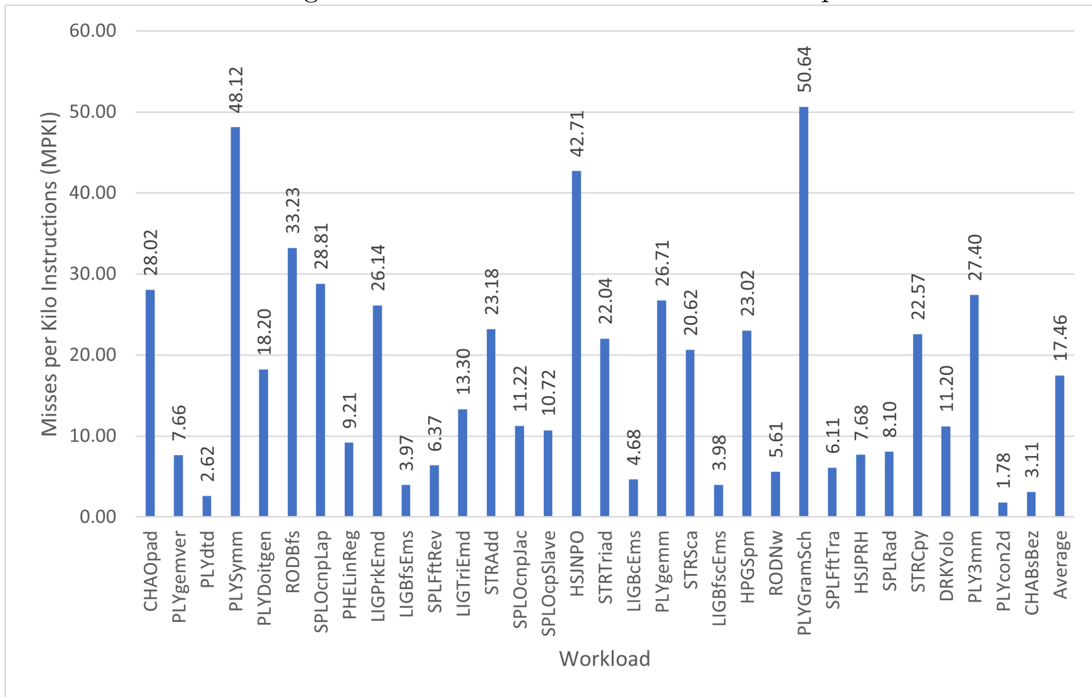
In the following section, we will only be discussing workloads that are significantly affected (positively or negatively) from our model. We will be analyzing both *always subscribe* policy and latency based global *adaptive* policy in details.

## 5.1 The *Always-Subscribe* policy

In Figure 5.2, we analyzed the performance of all 32 representative workloads with the *always-subscribe* policy, where we always perform subscription on the first access of a memory location. Some workloads show significant speedups. For example, SPLRad have up to 105% performance gain. On the other hand, workloads such as PLYgemm and PLY3mm have up to 17% performance losses. On average, all benchmarks have a geomean performance gain of nearly 6%.

Figure 5.2 also shows that many workloads do not demonstrate any performance impact (Speedup 1.00), indicating subscription has little to no effect. This is despite the fact that for

Figure 5.1: Baseline MPKI without warmup



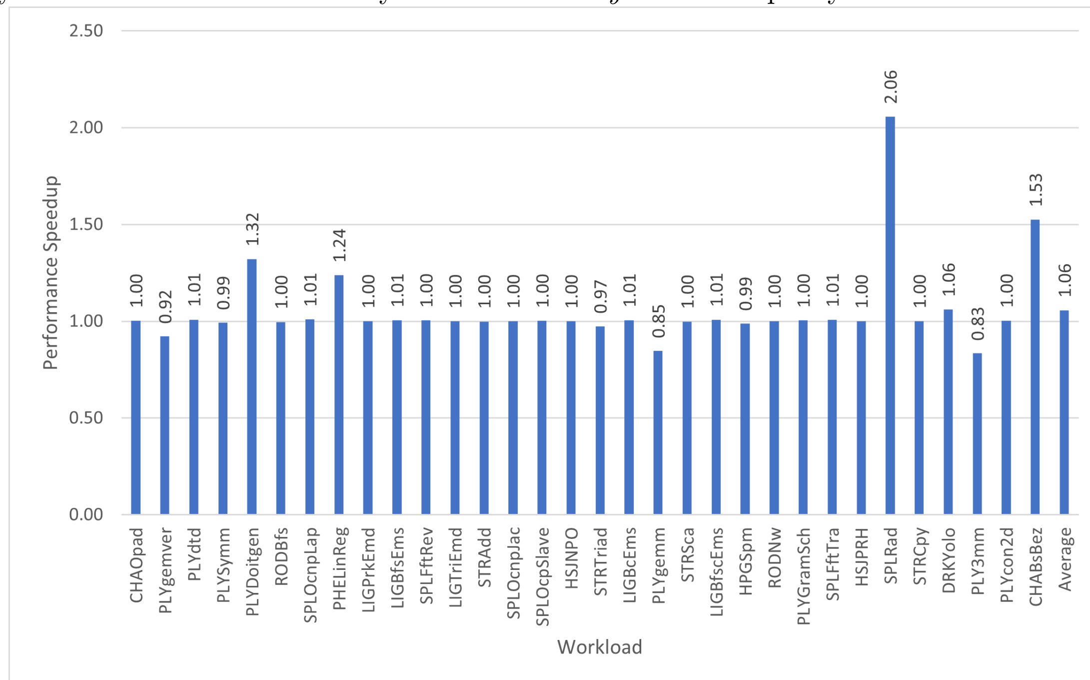

many of these benchmarks, data transfer and queuing delays still represent a considerable percentage of overall memory latency (Figure 1.2). Unfortunately, many of these workloads have very poor reuse properties as shown in Figure 5.3. The figure shows the average number of times a subscribed block has been reused either by the local subscribed vault (blue) or by a remote vault (orange). Many workloads have near zero average reuse per block, indicating that blocks are never accessed again after it has been subscribed and moved to the subscribed vault. This incurs no extra overhead for *always-subscribe* since the subscription data transfer would have been done anyway in the baseline when a processing units requests a block from a remote vault. However, negligible reuse implies that no benefit can be obtained from *always-subscribe*.

In the remainder of this chapter, we focus only on workloads that show non-negligible reuse as illustrated in Figure 5.3.

## 5.2 Comparing *Always-Subscribe* and *Adaptive* policies

Figure 5.4 shows that both *always subscribe* and *adaptive* policies affect performance significantly for many workloads. While most selected workloads benefit from *always-subscribe*, some are negatively affected. The geometric mean of speedup for selected benchmarks is nearly 14% for *always-subscribe* and 15% for *adaptive*, which successfully reduces performance degradation for workloads that are hurt by *always-subscribe*.

Figure 5.2: Performance gain for the *always-subscribe* policy, measured by the execution cycles of the baseline divided by that of the *always-subscribe* policy

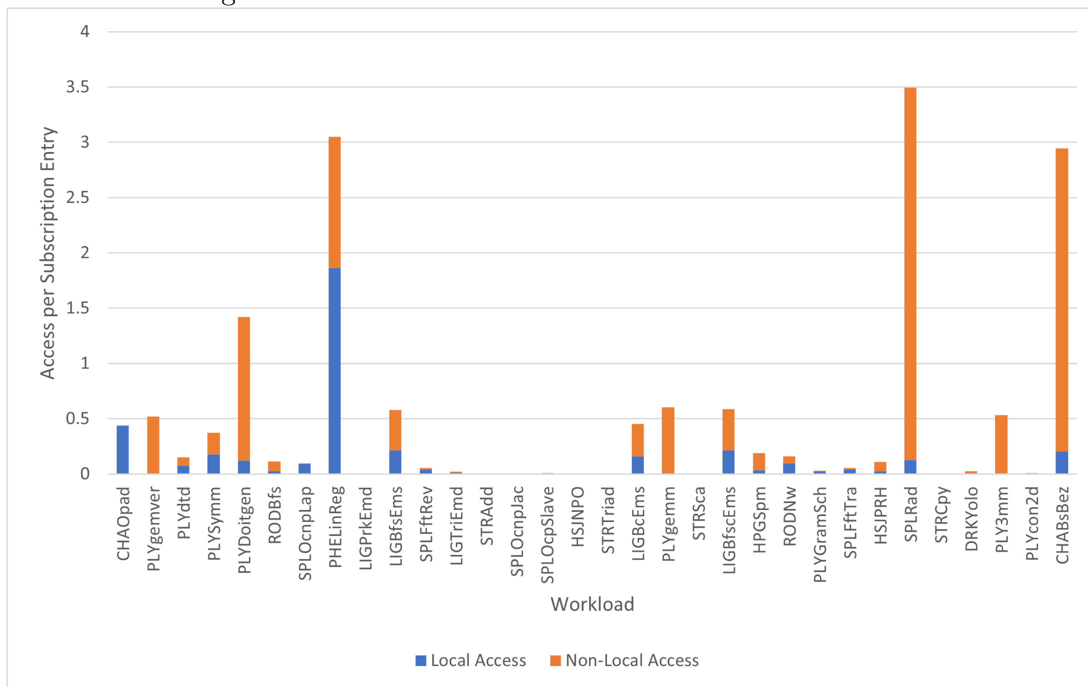



To provide insight into why some workloads benefit from *always-subscribe* while others are hurt by it, we show the average memory latency per request for the baseline, *always-subscribe* and *adaptive* in Figure 5.5. On average, *adaptive* reduces the average memory per request by nearly 54% across these selected workloads. However, different workloads have different reasons for higher/lower average per-request latency. We explain a few of these reasons next.

PHELinReg exhibits a 20% performance gain from *always-subscribe* since it significantly reduces the average number of hops per request. Figure 5.6 shows that PHELinReg has 33% lower total hops per memory request for *always-subscribe* compared to the baseline, resulting in a smaller fraction of memory access latency caused by inter-vault data transfers (Figure 5.7). Such improvement in overall memory latency is demonstrated in Figure 5.5.

CHABsBez and SPLRad do not have a high data transfer reduction with *always-subscribe* and even exhibit a higher average number of hops per request compared to the baseline (Figure 5.6). However, both have high coefficients of variation (Figure 5.8) indicating an uneven distribution where some vaults have much higher demand than the rest, therefore dominating performance. We investigated this behavior in Table 5.2 and Table 5.3 which shows the top 10 memory addresses for both workloads and their corresponding access count. The table demonstrates that a few hot locations contribute a large fraction of memory accesses which significantly increases queuing delays and degrades performance. For these two workloads, *always-subscribe* significantly reduces the coefficient of variation,

Figure 5.3: Average number of local and non-local accesses per subscription of *always-subscribe*. The blue portion indicates the average number of local accesses from the subscribed vault to a subscribed block after it was successfully subscribed. The orange portion indicates the average number of remote accesses to a subscribed block.




making the vault access distribution more uniform. This alleviates the queuing bottleneck (as shown in Figure 5.7) and significantly improves performance.

While our configuration and most PIM systems utilize cache to provide data locality, this is not proven to be useful in these two workloads. We further analyzed the cache behaviour pertaining the two most accessed locations (000001FFFF867B4D for CHABsBez and 000001FFFF88D4CD for SPLRad respectively). As it can be shown in Figure 5.9, both workloads' most accessed access has almost half of its requests being GETX, which invalidates other copies of the block. Our further analysis of the cache trace also shows that, there is usually a GETS request (requesting a read-only copy from another core) following the GETX request. Such request has to be served by the memory. These two memory locations also constitute 78.15% (for CHABsBez) and 88.9% (for SPLRad) of total memory requests. Therefore, our architecture evens out the access count to these hotly contented locations and reduces the queuing latency.

Table 5.2: The top 10 access memory locations of CHABsBez and the number of accesses to those locations

| Address (Hex) | Count |
|---|---|
| 000001FFFF867B4D | 3,745,249 |

| | |
|---|---|
| 0000000000018583 | 887,143 |
| 000000000001874B | 27,375 |
| 000000000001874C | 9,453 |
| 0000000000018752 | 3,063 |
| 0000000000018755 | 2,639 |
| 000000000001874D | 1,169 |
| 0000000000018751 | 755 |
| 00000000000185A1 | 231 |
| 000001FFFF87C2C3 | 132 |

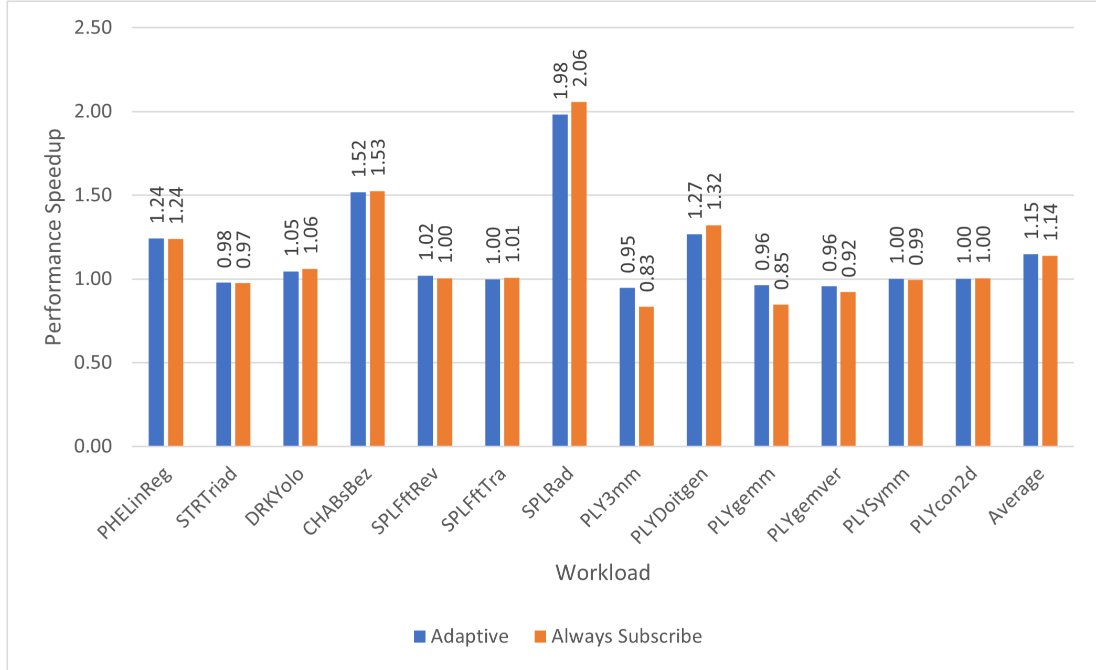
Table 5.3: The top 10 access memory locations of SPLRad and the number of accesses to those locations

| Address (Hex) | Count |
|---|---|
| 000001FFFF88D4CD | 14,925,709 |
| 00000000000282CF | 502,086 |
| 00000000000282CE | 481,196 |
| 00000000000283CD | 159,922 |
| 00000000000283D6 | 44,307 |
| 00000000000283D7 | 7,203 |
| 00000000000283D9 | 4,021 |
| 000001FFFF88D564 | 230 |
| 000001FFFF652B40 | 116 |
| 000001FFFF88D4C9 | 75 |

Workloads such as PLYgemm and PLY3mm incur 15-17% worse performance from *always-subscribe*. This is caused by *always-subscribe* increasing the coefficient of variation where it was lower with the baseline (Figure 5.8). For these workloads, *always-subscribe* makes the access distribution per vault more uneven and introduces hot vaults that dominate performance. The *adaptive* policy reduces the impact of the uneven distribution of *always-subscribe* since subscriptions are turned off after the first epoch, which limits the performance degradation to only 5%.

PLYDoitgen shows performance improvement with *always-subscribe* despite its low coefficient of variation. Figure 5.7 explains this behavior since a smaller fraction of memory latency is attributed to queuing delay. While the memory access distribution for the entire run is evenly distributed, each vault can become a hotspot for a short period of time causing high queuing delay. With *always-subscribe*, short-term hotspots are evened out therefore improving performance.

Figure 5.4: Performance gain of *always-subscribe* and *adaptive* normalized to the baseline

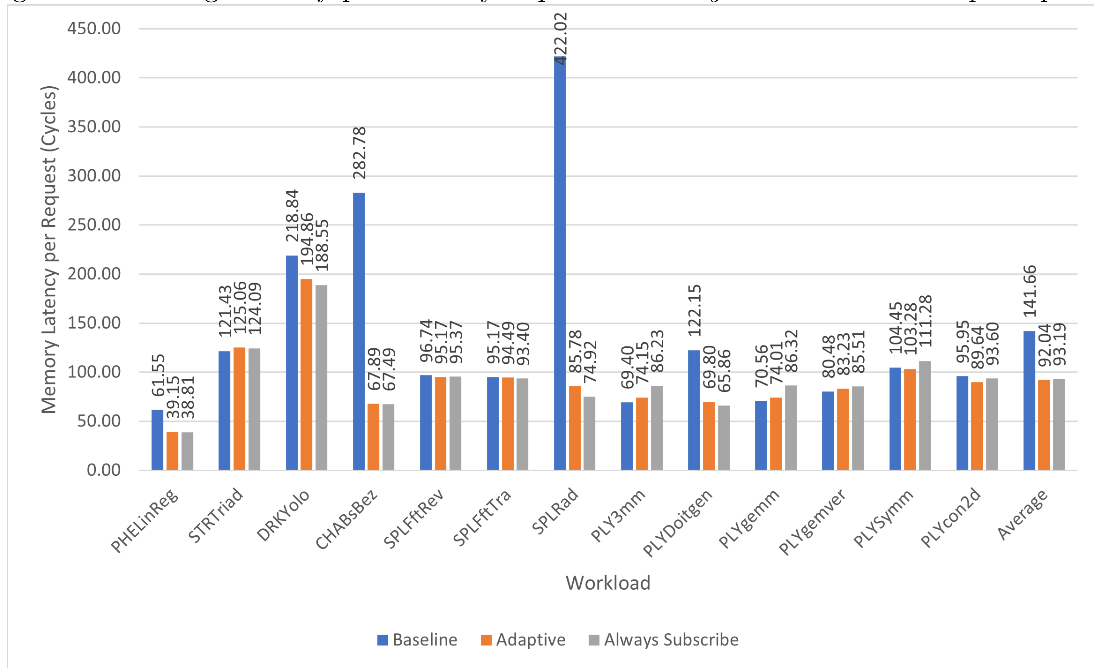



**Total Traffic.** An interesting result is shown in Figure 5.10 which summarizes the total network bandwidth demand (in bytes per cycle) for *always-subscribe* and *adaptive* compared to the baseline. This figure takes into account traffic caused by both memory accesses and subscription requests (the latter was not included in Figure 5.6). Some workloads (e.g., PHE-LinReg) have significant reductions in bandwidth demand vs. the baseline. However, most workloads have higher bandwidth demand for *always-subscribe* with an average increase of 88% vs. the baseline due to additional subscription traffic, mainly caused by the low degree of reuse for subscribed blocks (Figure 5.3). However, *adaptive* has an average increase of only 14%. While both flavors of DL-PIM increase average network bandwidth, the reduction in execution time from more local accesses would still lead to a more energy-efficient design.

## 5.3 Sensitivity to different per-hop latency

We modified the baseline configuration and run multiple different simulations. The first one we explored is the latency per packet per hop. As discussed in Chapter 2, HMC does not specify the technology used for intra-memory communications, and uses a crossbar based architecture as an example. In the original DAMOV implementation, it takes one cycle for each packet to travel one hop across the network and be forwarded to the next vault. This is dependent on different technologies used, which may have different latency and different routing algorithms. As such, we performed simulations for 1, 2, 4, 8 cycles per hop latency, as shown in Figure 5.11.

Figure 5.5: Average latency per memory request for *always-subscribe* and *adaptive* policies



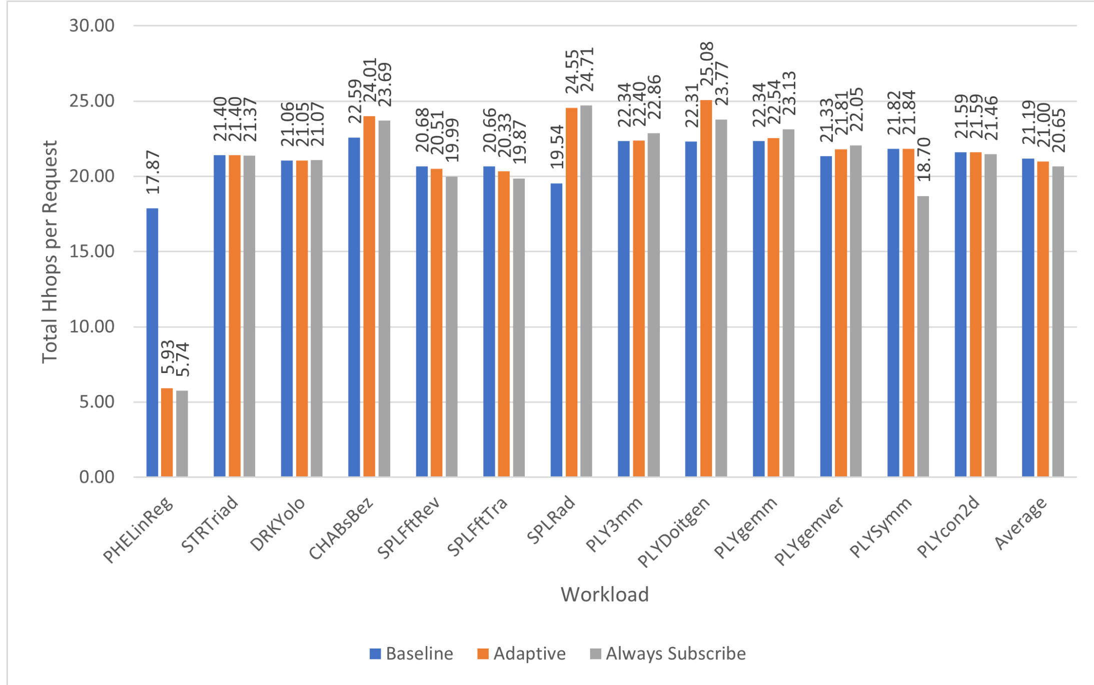

With the increase of latency per hop per packet, workloads that suffer from high data transfer overhead, like PHELinReg, benefit more from DL-PIM, while workloads that suffer from high queuing overhead, like CHABsBez, SPLRad and PLYDoitgen, tend to benefit less from subscription.

This is because, with high latency per hop per packet, the percentage of data transfer latency out of total latency is higher. As a result, there are fewer packets in queuing, and therefore, the workloads with high queuing latency do not benefit from our architecture in those configurations. This is further shown in Figure 5.12, in which all benchmarks show an increase in the percentage of transfer latency as the overhead per hop goes up.

## 5.4 Sensitivity to core count

We also run simulations with 4, 64, and 128-core configurations. While our baseline configuration uses only 32 cores, some PIM architectures have higher core counts to utilize the parallelism available. Because of the HMC architecture, by increasing the core count, we also increase the number of vaults in a given memory system. Therefore, our different core counts have different memory capacities, with 512MB for 4 core configuration, 8GB for the 64 core configuration, 16GB for the 128 core configuration, and 4GB for the baseline 32 core configuration. For each of those configurations, the memory vaults are configured to be placed in an $n \times n$ 2-D network, with $n = ceil(\sqrt{c})$ where $c$ stands for the number of vaults in our system. The result are shown in Figure 5.13.

Figure 5.6: Average total hops per memory access for *always-subscribe* and *adaptive* policies



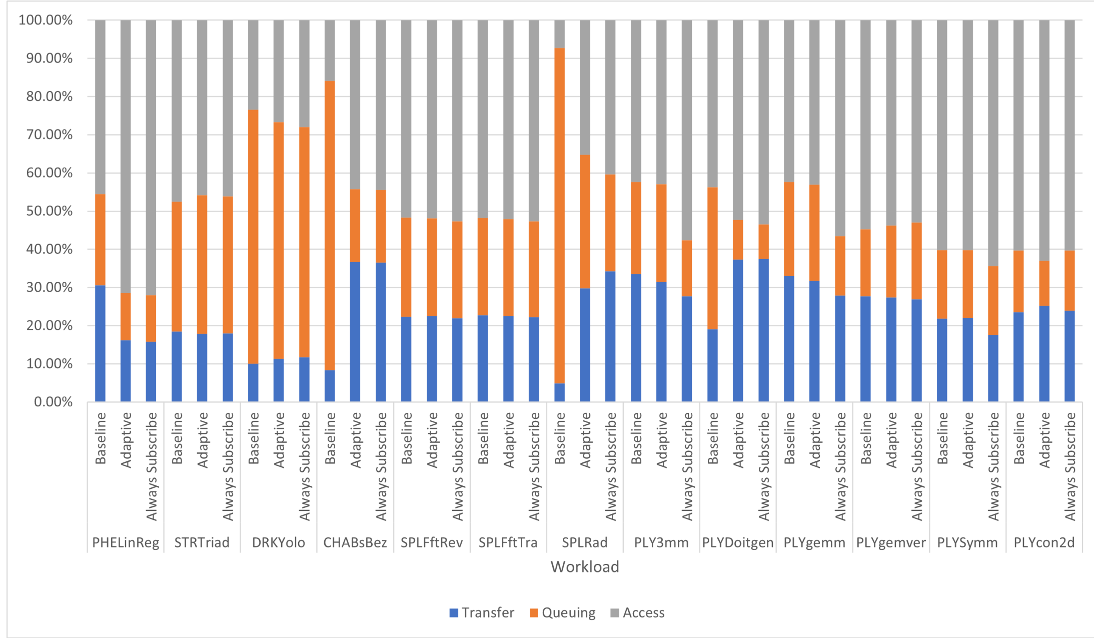

With the higher core count, the network size increases, and therefore the transfer latency as shown in Figure 5.14 increases. The increase of parallelism also means that more requests will be sent at a given time, and the queuing latency for workloads with high queuing latency also increases. Therefore, these two type of workloads will benefit more from DL-PIM.

## 5.5 Sensitivity to different adaptive policies

As discussed in section 3.4, we have developed different adaptive policies to dynamically turn subscription on and off. We have decided to use the latency-based adaptive policy with global adaptive policy and no set sampling. In this section, we will analyze the result of different adaptive policies, as shown in Figure 5.15.

Without set sampling, the "subscription away" problem is significant and is causing performance degradation in various configurations with SPLRad, PLY3mm, PLYgemm, PLYgemver and PLYSymm. Furthermore, while set sampling allows us to detect when a program enters a subscription benefiting region, its performance cost is still significant. Finally, hops-based subscription policy does not work well with SPLRad, which has a high coefficient of variation and queuing delays.

Figure 5.7: Breakdown of transfer, queuing and access latency in total memory latency for *always-subscribe* and *adaptive* policies




## 5.6 Sensitivity to different adaptive thresholds

As it is hard to measure the latency of memory requests without subscription vs. with subscription, we are using the change in latency from previous threshold as the indicator of whether we should make a policy change. We also experimented with different adaptive thresholds, and the results can be seen in Figure 5.16.

While SPLRad performs the best with 5% adaptive threshold (i.e. when the current hop's average latency is 105% or more or 95% or less of the previous hop), PLY3mm and PLYgemm are performing similarly as the "always subscribe" configuration. Because we start our simulation with the subscription turned on, with a high threshold, our result would converge to the "always subscribe" policy.

## 5.7 Sensitivity to subscription table sizes

We analyzed the impact of subscription table sizes on the performance. With larger tables, DL-PIM can hold more subscribed memory locations for each vault, but would increase hardware overhead. Figure 5.17 shows that some workloads (e.g., PLYDoitgen) saw a performance improvement with a larger subscription table, but that improvement flattened with a 8192-entry subscription table. We decided to use 8192 entries as our default configuration in this thesis, which incurs a 0.125% state overhead relative to the 4GB vault memory size.

Figure 5.8: Coefficient of variation (CoV) of the access distribution per vault for *always-subscribe* and *adaptive* policies. A high CoV implies uneven distribution where some vaults have much higher demand than others

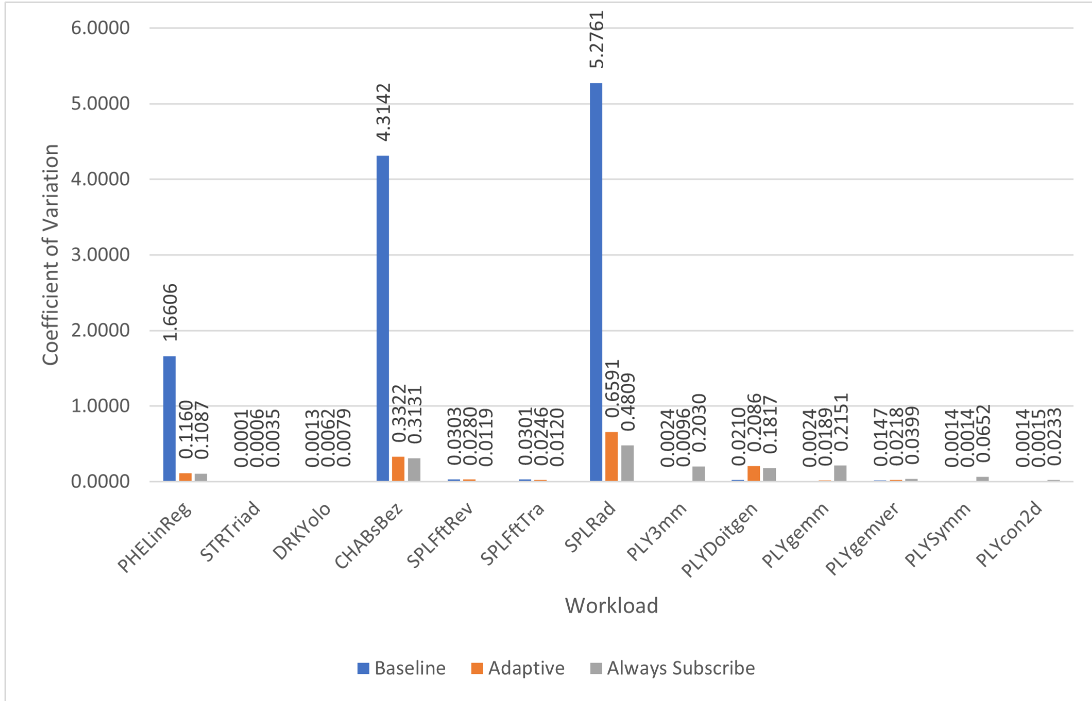



Figure 5.9: Breakdown of GETS and GETX requests for CHABsBez and SPLRad

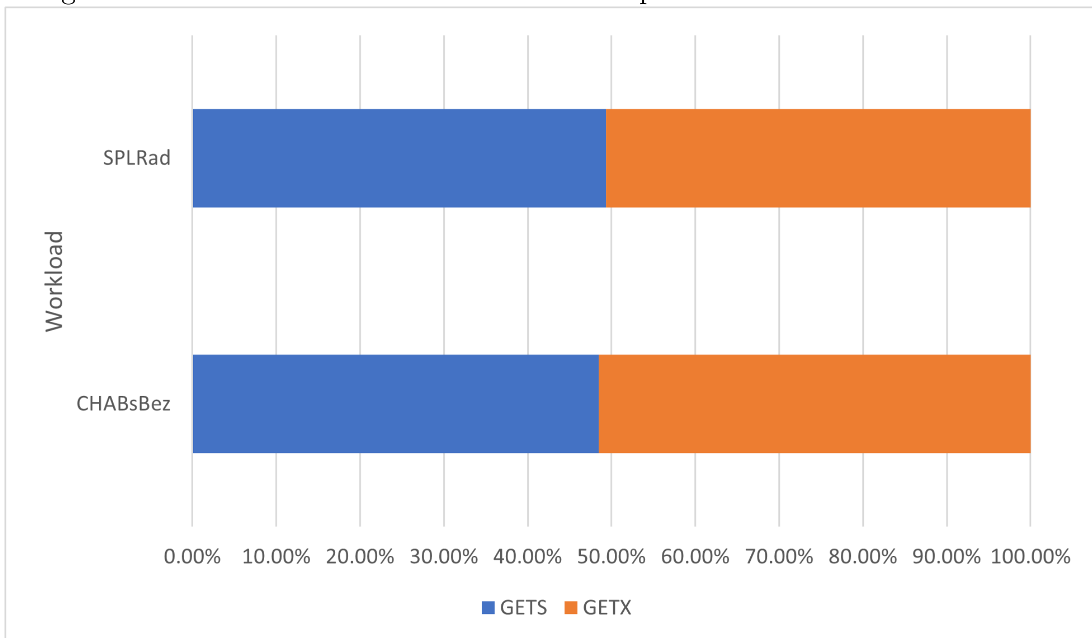


Figure 5.10: Average network traffic (in bytes per cycle) for *always-subscribe* and *adaptive* policies

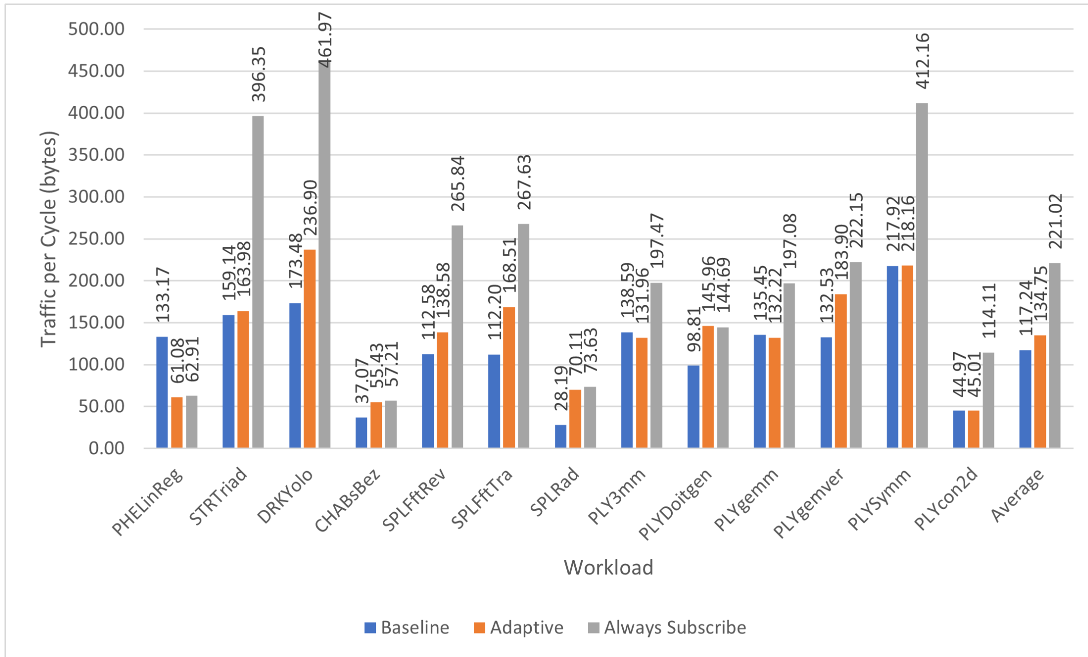



Figure 5.11: *Adaptive* speedup with different latency per hop

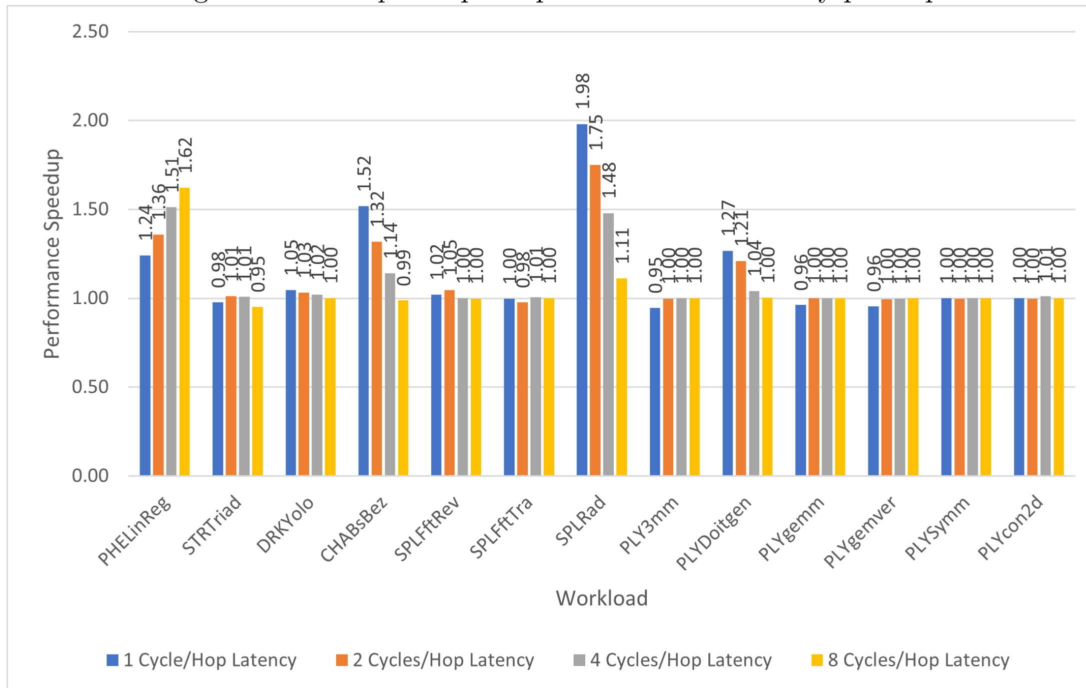


Figure 5.12: Breakdown of memory latency into data transfer latency, queuing delay and array access latency for different latency per hop

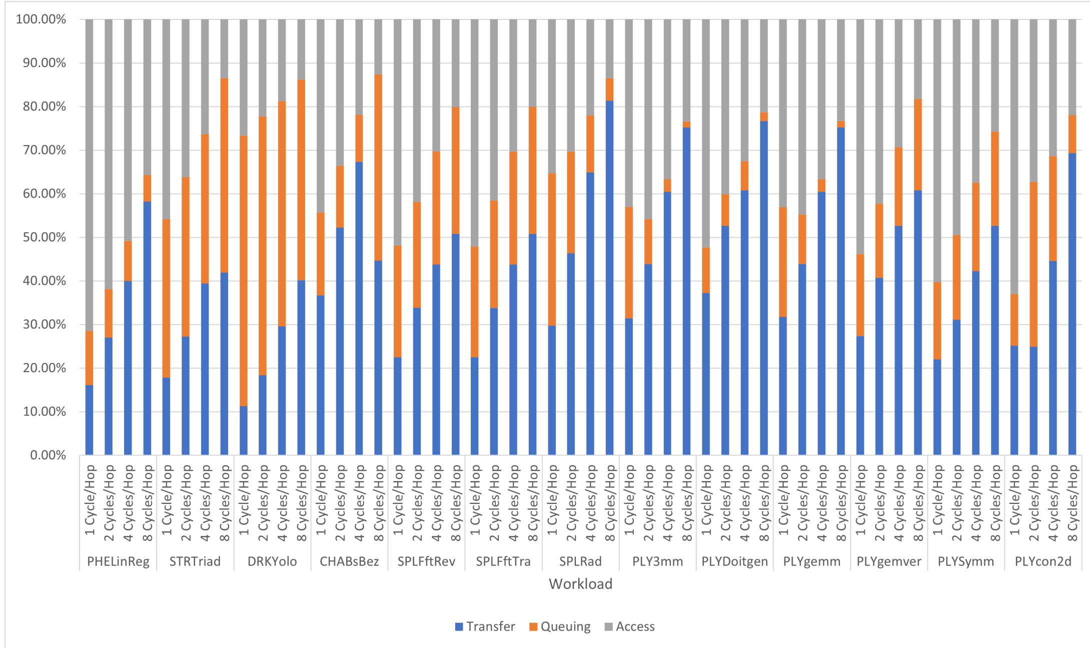



Figure 5.13: *Adaptive* speedup with different core number configurations

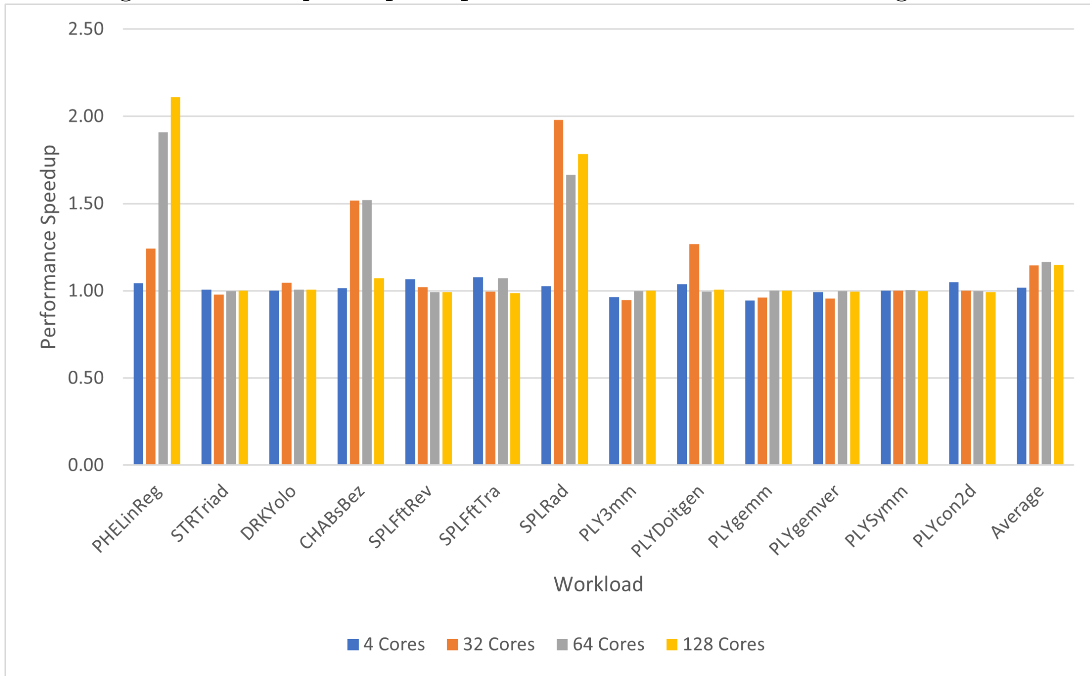


Figure 5.14: Breakdown of memory latency into data transfer latency, queuing delay and array access latency for different core numbers

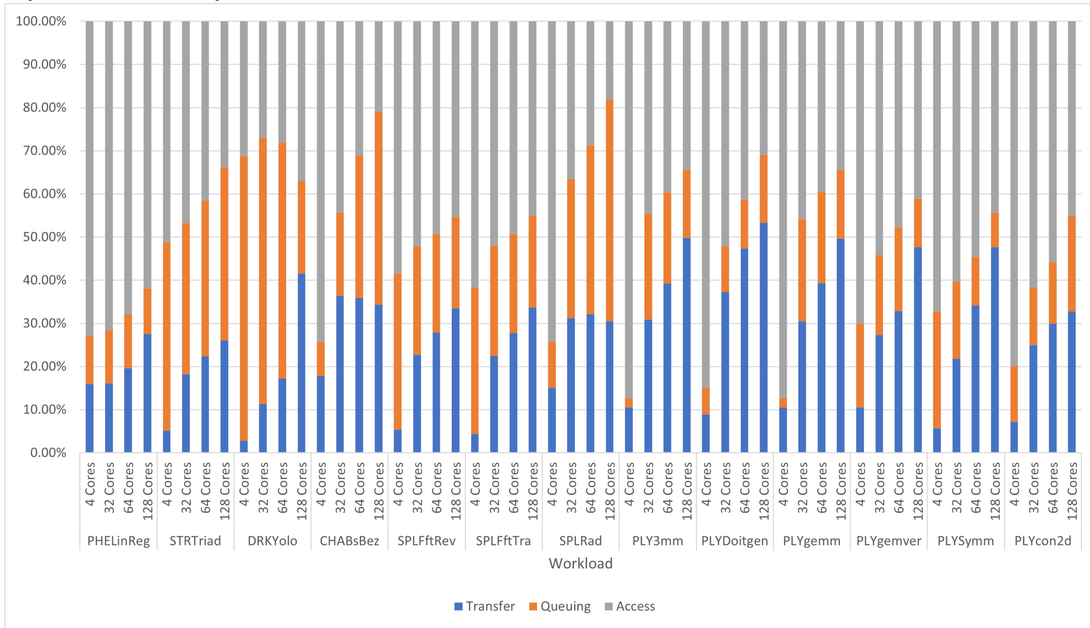



Figure 5.15: Performance speedup of selected benchmark with different adaptive policies

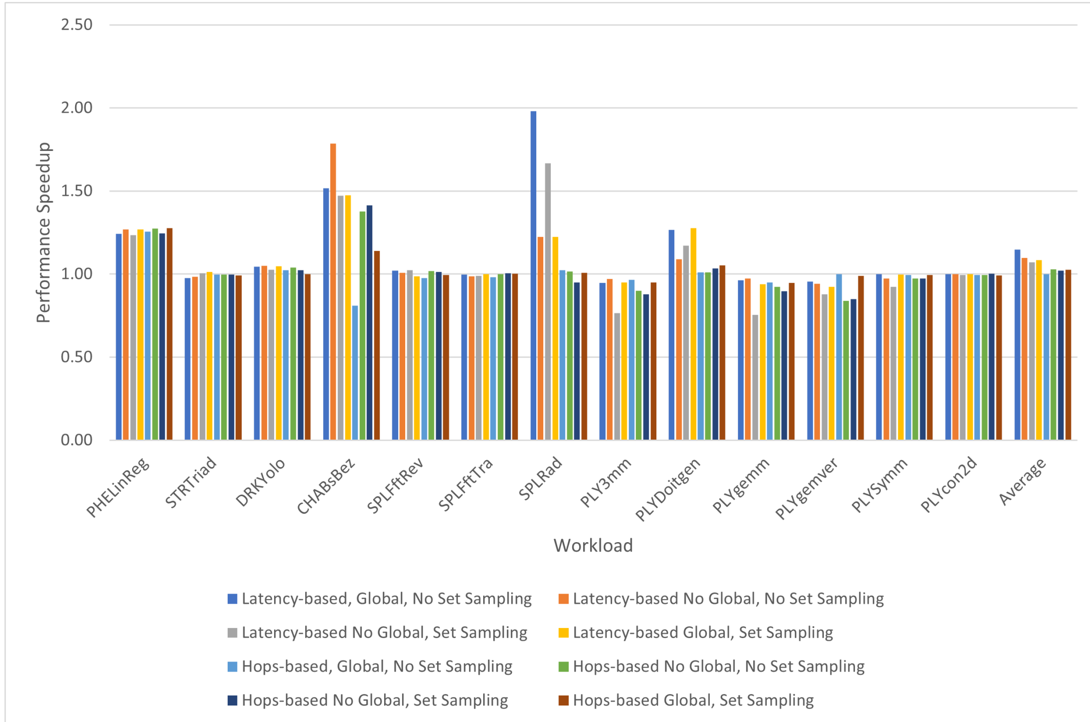


Figure 5.16: Performance speedup of selected benchmarks' adaptive policy with different adaptive thresholds

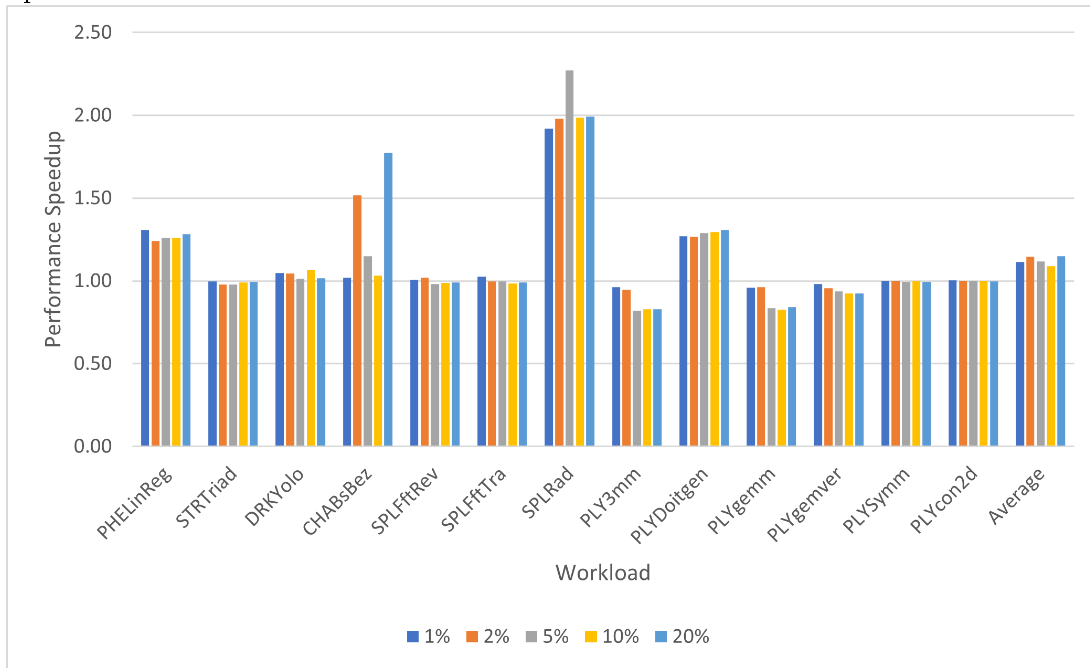
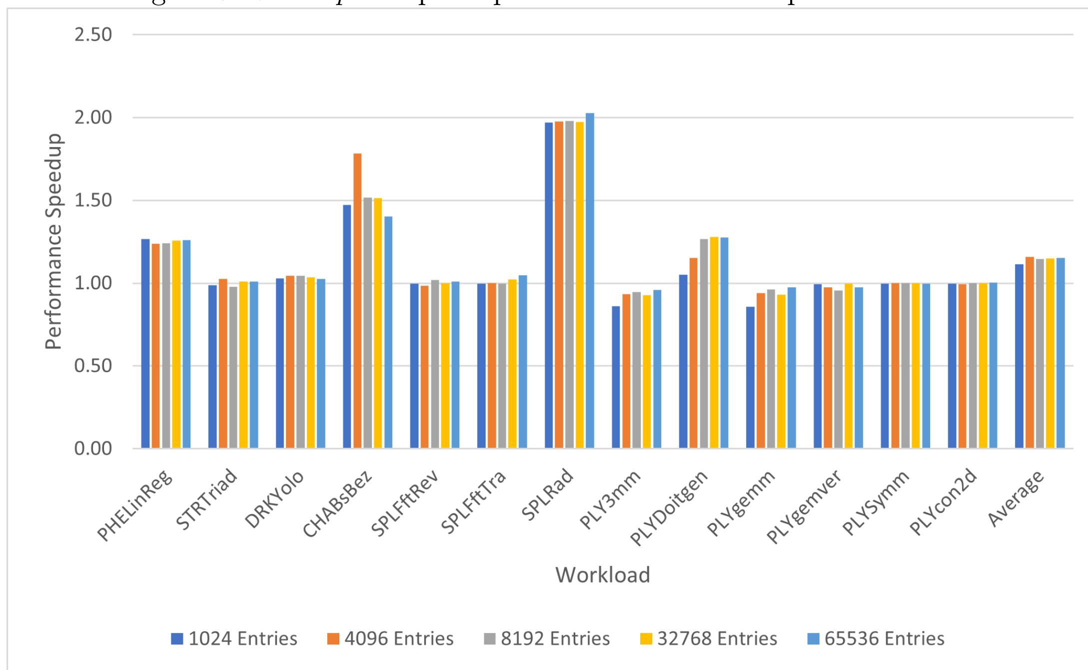



Figure 5.17: *Adaptive* speedup with different subscription table sizes



# Chapter 6

# Related Work

This work is inspired by GPS [80], which utilizes a subscription based hardware-software integrated framework to accelerate multi-GPU systems. GPS uses manual subscription management in software, allows data use by local and remote GPUs, and evaluates several automatic subscription algorithms. Conversely, DL-PIM is a hardware-only solution that does not change the existing program models or APIs.

A recent proposal by Tian et al. [104] (ABNDP) investigated reducing the network overhead in a PIM system via a hardware/software co-design that proposes a new programming model, and uses a *traveller cache* for remote data that resembles a traditional cache. One traveller cache is used by a group of 4 or more vaults. ABNDP divides the program into smaller tasks using its programming model, and schedules tasks dynamically based on data location. Our DL-PIM proposal is a hardware-only solution that uses a 3D-stacked memory system, and uses a reserved area in memory (not a separate cache) to subscribe remote addresses and reduce data movement per memory access. ALP [37] also seeks to alleviate the data movement overhead. However, ALP focuses mostly on the data movement overhead between the host and the PIM accelerators, instead of the data movement overhead between different PIM accelerators. Furthermore, ALP uses dynamic run-time information to decide whether and when to offload a "tightly connected segment" onto PIM accelerators, and our architecture instead subscribes on first access.

Prior work [69, 98] propose using stacked DRAM as a memory-side cache, automatically copying/moving data upon request. Most DRAM cache works seek to increase the capacity or reduce the latency of DRAM caches [85, 75, 109, 74, 54, 41, 111], but all focus on processor-centric systems. Other works target reducing the communication latency between different distributed DRAM caches in a multi-node system [22]. Our proposal uses stacked DRAM technology as a PIM system to reduce process-memory data movement.

Some prior work [112] proposed replacing the crossbar network in HMC with a mesh network and a reduction/dispersion tree architecture where memory controllers in the same row are connected together with one set of reduction/dispersion trees. While this proposal reduces communication overhead between memory controllers and the outside processors, it

does not work well with PIM architectures as it disallows communications between memory controllers without passing through the processor, so it does not reduce processor-memory data movement.

Our system at a high-level has some similarities to the Cache-only Memory Access (COMA) architecture [24, 103, 29]. COMA dynamically migrates data requested by a local node to a local *attraction memory* from the remote node upon access. However, COMA can replicate multiple copies of the same data block across different attraction memories for shared blocks. Our proposal invalidates the original copy on subscription. Although this may cause performance degradation, it is easier to implement and manage. The Reactive Non-uniform Cache Access (R-NUCA) machine [50] also seeks to alleviate data movement overhead by combining different caches to form *clusters*, and dynamically adjusts the cluster size based on software-defined parameters. R-NUCA attempts to evenly distribute the data across different caches within a cluster, to ensure even access latency.

PIM architectures have been extensively used to accelerate many important applications like graph processing [25, 2, 114, 113], neural networks [36, 82, 68, 7, 96], sparse matrix-vector multiplication [107, 39], weather prediction [100, 102], regression [44], time series analysis [31] and bioinformatics [60, 18, 43, 55, 8, 76]. Our proposal accelerated a fraction of the DAMOV workloads but did not focus on a specific application domain. We attempted to design a general PIM architecture that could benefit a variety of workloads.

# Chapter 7

# Conclusions

While PIM systems target reducing data movements between processors and memory, they incur significant data movement due to processing data in remote memory modules. We showed that most of the latency per memory request is caused by data transfer and queuing delays between different memory modules (vaults). We proposed DL-PIM, a subscription-based architecture for PIM systems that seeks to reduce the average latency per memory request by increasing the fraction of local accesses for PIM processing units. We proposed an *always-subscribe* policy that moves data from remote memory modules to a reserved area in the local module on first access. We also proposed an *adaptive* policy that alleviates performance degradation for some workloads caused by the extra traffic of *always-subscribe*. Our adaptive policy shows an average performance improvement of 6% across DAMOV representative workloads, and 15% for workloads that have non-trivial data reuse caused by a 54% reduction in average memory latency per request. However, DL-PIM had no impact on many workloads with poor data reuse. It also increased bandwidth demand by 14% due to the extra traffic caused by subscriptions.

Although our architecture is achieving significant performance improvement for many commonly used workloads, there are some limitations to it.

First of all, our architecture is focused on HMC memory architecture only. While HMC used to be a promising architecture for PIM systems, the cessation of development of it means that we should explore the potential to migrate this work to more recent architectures like High Bandwidth Memory (HBM).

Furthermore, our implementation allocates an area within the existing memory to provide local data access. While this is simple to implement, it produces additional hardware overheads as we need additional memory hardware to store subscribed data. This limits the size of our subscription table and the potential benefits of our architecture. In the future, we can explore a "swap" mechanism in which we do not add additional memory space but instead "swap" the subscribed data with a non-frequently accessed local memory location.

Another issue with our current subscription mechanism is the "subscription away" issue that we've discussed previously. To permanently solve this issue, we can explore a "copy"

mechanism in which the data is kept in its original place, but simply "copied" to the subscribed location. This mechanism requires updating all copies of a memory block whenever it is modified by any of the cores. This cannot be enforced only using the cache coherence protocol but requires invalidating memory copies whenever a cache block is invalidated for a shared block. This introduces a lot of additional complexity in designing the protocol, in addition to significant validation and verification effort. However, this architecture would significantly reduce the cost of subscription and allow for better performance on even more workloads.

Finally, DAMOV's implementation does not consider the scenario when the input buffer of a memory request's destination vault is full, and assume that the memory request is placed in the input buffer of its destination vault immediately upon issuance. This may cause the performance result to be slightly different from the real hardware.

# Bibliography


[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, 2017.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 336–348, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. *SIGARCH Comput. Archit. News*, 43(3S):336–348, jun 2015.

[5] Shaahin Angizi and Deliang Fan. Accelerating bulk bit-wise x(n)or operation in processing-in-dram platform, 2019.

[6] Shaahin Angizi, Zhezhi He, and Deliang Fan. Pima-logic: A novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[7] Shaahin Angizi, Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. Cmp-pim: An energy-efficient comparator-based processing-in-memory neural network accelerator. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[8] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. Aligns: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[9] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[10] Aurelia Augusta and Stratos Idreos. Jafar: Near-data processing for databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management*

*of Data*, SIGMOD '15, page 2069–2070, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1754–1766, 2015.

[12] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. Revamp: Reram based vliw architecture for in-memory computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 782–787, 2017.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[14] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. *SIGPLAN Not.*, 53(2):316–331, mar 2018.

[15] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 629–642, 2019.

[16] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 16(1):46–50, 2017.

[17] F. Nisa Bostancı, Ataberk Olgun, Lois Orosa, A. Giray Yağlıkçı, Jeremie S. Kim, Hasan Hassan, Oğuz Ergin, and Onur Mutlu. Dr-strange: End-to-end system design for dram-based true random number generators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1141–1155, 2022.

[18] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 951–966, 2020.

[19] Kevin K. Chang, Prashant J. Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K. Qureshi, and Onur Mutlu. Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–580, 2016.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[21] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[22] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Candy: Enabling coherent dram caches for multi-node systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[23] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*, pages 213–224, 2015.

[24] F. Dahlgren and J. Torrellas. Cache-only memory architectures. *Computer*, 32(6):72–79, 1999.

[25] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2019.

[26] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.

[27] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. *Knoxville, Tennessee*, 42, 2015.

[28] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 383–396, 2018.

[29] Babak Falsafi and David A. Wood. Reactive numa: A design for unifying s-coma and cc-numa. *SIGARCH Comput. Archit. News*, 25(2):229–240, May 1997.

[30] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, 2015.

[31] Ivan Fernandez, Aditya Manglik, Christina Giannoula, Ricardo Quislant, Nika Mansouri Ghiasi, Juan Gómez-Luna, Eladio Gutierrez, Oscar Plata, and Onur Mutlu. Accelerating time series analysis via processing using non-volatile memories, 2022.

[32] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality Cache for Data Parallel Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 397–410, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Pierre-Emmanuel Gaillardon, Luca Amarú, Anne Siemon, Eike Linn, Rainer Waser, Anupam Chattopadhyay, and Giovanni De Micheli. The programmable logic-in-memory (plim) computer. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 427–432, 2016.

[34] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 113–124, 2015.

[35] Mingyu Gao and Christos Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137, 2016.

[36] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Nika Mansouri Ghiasi, Nandita Vijaykumar, Geraldo F. Oliveira, Lois Orosa, Ivan Fernandez, Mohammad Sadrosadati, Konstantinos Kanellopoulos, Nastaran Hajinazar, Juan Gómez Luna, and Onur Mutlu. Alp: Alleviating cpu-memory data movement overheads in memory-centric systems. *IEEE Transactions on Emerging Topics in Computing*, 11(2):388–403, 2023.

[38] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019.

[39] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1), feb 2022.

[40] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016.

[41] Nagendra Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 38–50, 2014.

[42] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze-Meng Low, Lawrence Pileggi, James C Hoe, and Franz Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design, 2014.

[43] Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, and Tajana Rosing. Rapid: A reram processing in-memory architecture for dna sequence alignment. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2019.

[44] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. An experimental evaluation of machine learning training on a real processing-in-memory system, 2023.

[45] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 43–54, 2017.

[46] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10:52565–52608, 2022.

[47] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: A framework for bit-serial simd processing using dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 329–345, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Said Hamdioui, Shahar Kvatinsky, Gert Cauwenberghs, Lei Xie, Nimrod Wald, Siddharth Joshi, Hesham Mostafa Elsayed, Henk Corporaal, and Koen Bertels. Memristor for computing: Myth or reality? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 722–731, 2017.

[49] Said Hamdioui, Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, Henk Corporaal, Hailong Jiao, Francky Catthoor, Dirk Wouters, Linn Eike, and Jan van Lunteren. Memristor based computation-in-memory architecture for data-intensive applications. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1718–1725, 2015.

[50] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 184–195, New York, NY, USA, 2009. Association for Computing Machinery.

[51] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Accelerating dependent cache misses with an enhanced memory controller. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 444–455, 2016.

[52] Milad Hashemi, Onur Mutlu, and Yale N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *2016 49th Annual*

*IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[53] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32, 2016.

[54] Cheng-Chieh Huang and Vijay Nagarajan. Atcache: Reducing dram cache latency via a small sram tag cache. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 51–60, 2014.

[55] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. Radar: A 3d-reram based dna alignment accelerator architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[56] Maurus Item, Geraldo F. Oliveira, Juan Gómez-Luna, Mohammad Sadrosadati, Yuxin Guo, and Onur Mutlu. Transpimlib: Efficient transcendental functions for processing-in-memory systems. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 235–247, 2023.

[57] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. An energy-efficient vlsi architecture for pattern recognition via deep embedding of computation in sram. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8326–8330, 2014.

[58] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro*, 42(1):116–127, 2022.

[59] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.

[60] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies. *BMC genomics*, 19(2):23–40, 2018.

[61] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[62] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

[63] Shahar Kvatinsky, Avinoam Kolodny, Uri C. Weiser, and Eby G. Friedman. Memristor-based imply logic design procedure. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 142–147, 2011.

[64] Shahar Kvatinsky, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. Memristor-based material implication (imply) logic: Design principles and methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2054–2066, 2014.

[65] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. A 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 350–352, 2021.

[66] James Laudon and Daniel Lenoski. The sgi origin: A ccnuma highly scalable server. *SIGARCH Comput. Archit. News*, 25(2):241–251, May 1997.

[67] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 241–252, 2015.

[68] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for pim based on commercial dram technology : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56, 2021.

[69] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. A fully associative, tagless dram cache. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 211–222, 2015.

[70] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(2SI):148–159, may 1990.

[71] Yifat Levy, Jehoshua Bruck, Yuval Cassuto, Eby G. Friedman, Avinoam Kolodny, Eitan Yaakobi, and Shahar Kvatinsky. Logic operations in memory using a memristive akers array. *Microelectronics Journal*, 45(11):1429–1437, 2014.

[72] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301, 2017.

[73] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[74] Gabriel H. Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 201–212, 2009.

[75] Gabriel H. Loh and Mark D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 454–464, 2011.

[76] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. Genstore: A high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 635–654, New York, NY, USA, 2022. Association for Computing Machinery.

[77] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[78] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, page 162, New York, NY, USA, 2004. Association for Computing Machinery.

[79] Micron Technology, Inc. Hybrid Memory Cube – HMC Gen2.

[80] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. Gps: A global publish-subscribe model for multi-gpu memory management. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–58, 2021.

[81] Geraldo F Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks. *IEEE Access*, 9:134457–134502, 2021.

[82] Geraldo F. Oliveira, Juan Gómez-Luna, Saugata Ghose, Amirali Boroumand, and Onur Mutlu. Accelerating neural network inference with processing-in-dram: From the edge to the cloud. *IEEE Micro*, 42(6):25–38, 2022.

[83] Jisung Park, Roknoddin Azizi, Geraldo F. Oliveira, Mohammad Sadrosadati, Rakesh Nadig, David Novo, Juan Gómez-Luna, Myungsuk Kim, and Onur Mutlu. Flash-cosmos: In-flash bulk bitwise operations using inherent computation capability of nand flash memory. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 937–955, 2022.

[84] Louis-Noel Pouchet. Polybench: The polyhedral benchmark suite. https://www.cs.colostate.edu/~pouchet/software/polybench/, 2011–2012.

[85] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, 2012.

[86] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 167–178, USA, 2006. IEEE Computer Society.

[87] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[88] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

[89] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the memory wall: The case for processor/memory integration. *SIGARCH Comput. Archit. News*, 24(2):90–101, May 1996.

[90] Vivek Seshadri, Kevin Hsieh, Amirali Boroum, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Fast bulk bitwise and and or in dram. *IEEE Computer Architecture Letters*, 14(2):127–131, 2015.

[91] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 185–197, 2013.

[92] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Buddy-ram: Improving the performance and efficiency of bulk bitwise operations using dram, 2016.

[93] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287, 2017.

[94] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A. Kozuch, and Todd C Mowry. Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 267–280, 2015.

[95] Vivek Seshadri and Onur Mutlu. The processing using memory paradigm:in-dram bulk copy, initialization, bitwise and and or, 2016.

[96] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[97] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, Feb 2013.

[98] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–257, 2012.

[99] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. Fpga-based near-memory acceleration of modern data-intensive applications. *IEEE Micro*, 41(4):39–48, 2021.

[100] Gagandeep Singh, Dionysios Diamantopoulos, Juan Gómez-Luna, Christoph Hagleitner, Sander Stuijk, Henk Corporaal, and Onur Mutlu. Accelerating weather prediction using near-memory reconfigurable fabric. *ACM Trans. Reconfigurable Technol. Syst.*, 15(4), jun 2022.

[101] Gagandeep Singh, Juan Gómez-Luna, Giovanni Mariani, Geraldo F. Oliveira, Stefano Corda, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Napel: Near-memory computing application performance prediction via ensemble learning. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[102] Gagandeep Singh, Alireza Khodamoradi, Kristof Denolf, Jack Lo, Juan Gomez-Luna, Joseph Melber, Andra Bisca, Henk Corporaal, and Onur Mutlu. Sparta: Spatial acceleration for efficient and scalable horizontal diffusion weather stencil computation. In *Proceedings of the 37th International Conference on Supercomputing*, ICS '23, page 463–476, New York, NY, USA, 2023. Association for Computing Machinery.

[103] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 80–91, 1992.

[104] Boyu Tian, Qihang Chen, and Mingyu Gao. Abndp: Co-optimizing data access and load balance in near-data processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 3–17, 2023.

[105] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[106] Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Said Hamdioui, and Koen Bertels. Fast boolean logic mapped on memristor crossbar. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 335–342, 2015.

[107] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583, 2021.

[108] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, 2009.

[109] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 328–339, 2018.

[110] Jintao Yu, Hoang Anh Du Nguyen, Lei Xie, Mottaqiallah Taouil, and Said Hamdioui. Memristive devices for computation-in-memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1646–1651, 2018.

[111] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient dram caching via software/hardware cooperation. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2017.

[112] Jia Zhan, Itir Akgun, Jishen Zhao, Al Davis, Paolo Faraboschi, Yuangang Wang, and Yuan Xie. A unified memory network architecture for in-memory computing in commodity servers. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2016.

[113] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, 2018.

[114] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 712–725, New York, NY, USA, 2019. Association for Computing Machinery.