TÉCNICO LISBOA

Ambix: Rethinking Linux's Page Management to Support the new Intel Optane DC Persistent Memory

Miguel Soares Marques

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Pedro Faria Mendonça Barreto

Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes

Supervisor: Prof. João Pedro Faria Mendonça Barreto

Member of the Committee: Prof. Paolo Romano

June 2021

Acknowledgments

First and foremost, I would like to express my deep gratitude to my thesis supervisors, professors João Barreto and Rodrigo Rodrigues, and to professor José Carlos Monteiro, who provided invaluable guidance and advice throughout my dissertation. To my family, who has always provided and supported me throughout my academic journey. To my friends, who motivated me to continue and improve not only at an academic but also at a personal level.

To each and every one of you – May this thesis make you proud – Thank you.

Abstract

Intel Optane™ DC persistent memory module (DCPMM) is an emergent non-volatile memory (NVM) technology that is promising due to its byte-addressability, high density, and similar performance to DRAM.

Prior literature explores a new architectural paradigm, coined hybrid memory architecture (HMA), which results of the configuration of NVM as a memory tier between DRAM and storage. HMAs have the potential to improve applications by enabling them to place a larger working set in fast memory, and thus reduce the need of evicting data to slow block-based storage. HMAs also mitigate the well-known memory scalability problem, common in a plethora of large servers and supercomputers. These systems cannot deploy more physical memory due to size, energy or cost limitations, all of which are alleviated by NVM integration.

However, most NVM research explores its non-volatility as an enabler to faster data persistence, neglecting the scalability benefit offered by NVM integration, in the HMA scenario. Conversely, existing NVM research in the data placement field precedes the commercial availability of NVM, testing HMAs in often-inaccurate simulation-based environments, inferring NVM's performance from outdated technologies.

Our thesis proposes Ambix, the first published solution tested on a real system running DCPMM, which decides page placement dynamically in a Linux system. We extensively discuss how different memory policies and distributions affect throughput and energy consumption in a DRAM-DCPMM system, leveraging the conclusions to guide Ambix 's design. We show that Ambix has an up to 10x speedup in HPC-dedicated benchmarks, compared to the default memory policy in Linux.

Keywords: Hybrid Memory Architecture, Persistent Memory, Intel Optane, Dynamic Placement.

Resumo

O módulo de memória persistente Intel Optane DC (DCPMM) é uma tecnologia NVM emergente que é promissora devido à sua capacidade de ser endereçável ao byte, alta densidade, e desempenho semelhante ao da DRAM.

Trabalhos nesta área exploram uma arquitetura de memória híbrida (HMA), que introduz a NVM como uma camada de memória entre a DRAM e o armazenamento. Esta arquitetura tem o potencial de melhorar o desempenho de aplicações, permitindo que seja colocado um maior número de dados em memória, reduzindo assim a necessidade de despejar dados para o armazenamento. As HMAs também atenuam o problema de escalabilidade da memória, comum em servidores e supercomputadores atuais.

Contudo, a maioria das publicações sobre NVM exploram o seu aspeto não volátil como um meio de persistir dados mais rapidamente, não tendo em conta o benefício de escalabilidade que a integração da NVM mostra oferecer, no cenário da arquitetura híbrida. Por outro lado, trabalhos na área de colocação de dados precedem a comercialização da NVM, testando HMAs em ambientes de simulação imprecisos, inferindo o desempenho da NVM com base em tecnologias desatualizadas.

Nesta dissertação propomos o Ambix, que será a primeira solução de placement dinâmico testada num sistema real configurado com DCPMM. Discutimos extensivamente como diferentes políticas e distribuições de memória afetam tanto o throughput como o consumo energético num sistema DRAM-DCPMM, usando as conclusões na implementação do Ambix. Mostramos que o Ambix tem um speedup de até 10x em benchmarks HPC, comparado com a política de memória default do Linux.

Palavras-chave: Arquitetura de Memória Híbrida, Memória Persistente, Intel Optane, Placement Dinâmico.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Memory scalability is an ever more relevant concern in the era of Exascale computing. Workloads running on Exascale supercomputers will most likely be characterized by a high degree of data complexity and parallelism, as well as a growing demand for increased memory capacity [1, 2]. Dynamic random access memory (DRAM) is the current de facto standard in a system's memory and an integral part of the current architecture. However, when considering scaling up the amount of DRAM in supercomputers or large servers, multiple problems arise, from power limitations and cooling, to area constraints and cost [3].

Non-volatile memory (NVM[1]) is an emerging class of memory that is able to mitigate these issues [4]. It is byte-addressable, performs similarly to DRAM, and keeps its data on the event of a power loss, akin to persistent disk or drive-based storage. Compared to DRAM, NVM does not require power to refresh data periodically, therefore having a lower static energy consumption, is denser, and has a lower cost per GB. These three advantages mitigate the three main problems of memory scalability: energy use, area constraints and budget limitations, respectively. However, NVM has undesirable characteristics: while latest NVM technologies perform only slightly worse compared to DRAM in read latency, write latencies are at best $\sim$10x slower, under idle conditions; secondly, writes to NVM consume more energy, counteracting its static energy advantages in write-intensive workloads; thirdly, NVM bandwidth is inferior, which limits the amount of concurrent accesses to it; and lastly, while DRAM endurance could be considered infinite, NVM can only endure a limited number of writes before failing.

Multiple NVM technologies were developed, varying in density, cost and performance [5]. Most notably, phase-change memory (PCM) [6–10], spin-transfer torque RAM (STT-RAM) [11–13], ferroelectric field-effect transistor (FeFET) [14, 15] and resistive random-access memory (RRAM) [16] have all been previously studied as possible candidates for replacing or complementing DRAM.

Intel recently made commercially available a byte-addressable NVM named Intel's Optane Data Center Persistent Memory Module (DCPMM), based on 3D XPoint non-volatile memory technology. Compared to contemporary block-based non-volatile memory technologies, DCPMM significantly narrows down the performance gap to volatile memory, while improving in terms of density and endurance

[1] NVM has been given multiple nomenclatures in previous literature: non-volatile RAM (NVRAM), persistent memory (PM) and storage-class memory (SCM) all relate to the study of persistent memory. We will use NVM when referring to persistent memory.

[17, 18]. DCPMMs are up to 4x denser than DRAM, ranging from 128 to 512GB, and are compatible with DDR4 DIMM slots. Its price per GB is estimated to be significantly cheaper than that of current gen DRAM, making it a viable option for replacing costly DRAM sticks with higher density non-volatile ones. For the above reasons, the Aurora system, currently projected to be one of the first Exascale supercomputers in the US, will include both DRAM and DCPMM in order to overcome the aforementioned memory scalability challenges at Exascale-level [19].

1.1 Motivation

When DCPMM is configured to extend main memory, applications are able to run larger data sets without needing to swap or evict data to storage due to main memory limitations. However, due to its latency and bandwidth limitations [18], allocating data to the appropriate memory tier is most relevant, as placing intensive and especially write-intensive data in it can severely hinder an application's performance.

DCPMM provides a novel mode of operation that configures DRAM as a cache, and DCPMM as the sole memory tier. In this configuration, the most intensive data is cached in DRAM in a way seamless to the programmer. Moreover, DCPMM also allows a more traditional configuration, where the system has two separate memory tiers, for DRAM and DCPMM. This configuration increases total memory further, as DRAM is directly accessible, and provides the system and user with the ability to decide where data should be placed.

The latter configuration is known as an hybrid memory architecture (HMA)[2], due to the heterogeneity of memories in the system. Multiple solutions have been proposed in the past, which place or migrate data to a memory tier based on its intensiveness and access pattern, taking into account the performance differences provided in each available tier.

The commercial availability of large-scale NVM constitutes a notable opportunity to revisit previously proposed techniques for data placement in hybrid DRAM-NVM systems and to devise new implementations that are tailored to the idiosyncrasies of the real NVM hardware. In this paper, we explore such path to propose Ambix, a dynamic page placement for off-the-shelf Linux-based systems equipped with DCPMM.

However, due to DCPMM's recentness, there is no published real-world implementation of a data placement algorithm that considers a DCPMM-equipped system. The existing studies are mostly theoretical or simulation-based, due to the memory's limited availability. As far as we know, only a single unpublished solution, which is being developed in parallel with this work, is tested on a real machine running DCPMM [20]. We see this as an opportunity to contribute to the NVM research field, with the first work on a data placement solution specifically designed with DCPMM's characteristics in mind, which leverages the increased memory scalability offered by DCPMM to improve relevant large data set workloads, tested on a real NVM-equipped commodity system.

We may also compare HMAs to a non-uniform memory access (NUMA) architecture composed of only DRAM. These architectures provide different access latencies depending on the memory accessed

[2]We use the term HMA to refer to architectures that use a multi-tiered memory configuration

by a CPU, similarly to how NVM provides a slower access latency when compared to DRAM. A plethora of literature proposes data placement algorithms that consider same-memory NUMA systems. Furthermore, the current Linux kernel already has support for NUMA balancing, and can distribute a process' pages, according to factors such as page intensiveness and memory distance to the computing node.

Nevertheless, previous NUMA-aware solutions, including Linux's implementation, are unsuitable for DCPMM and other NVM technologies, as they do not consider factors such as NVM's higher write asymmetry, lower bandwidth and active energy consumption, leading to suboptimal page distributions in machines that are equipped with the emergent memory.

1.2 Objectives

The main goal of our work is to complement the existing Linux's page management mechanisms with a solution that considers an heterogeneous memory configuration, consisting of both DRAM and DCPMM. We focus our efforts on a solution that requires minimal changes to the Linux kernel, and expands existing page placement mechanisms in order to accommodate the integration of DCPMM. We show that we are able to determine a performant page distribution dynamically, with low overhead.

The contributions of this thesis are summarized as follows:

- As a **first contribution**, we start by empirically studying some fundamental performance properties of DCPMM that are relevant to the design of dynamic page placement solutions. This allows us to reach a set of design guidelines, from which we then build Ambix. It is worth noting that our observations invalidate some key design choices of several previous proposals in literature.

- As a **second contribution**, we leverage the guidelines to design and implement a dynamic page placement solution, Ambix, as a complement to the existing Linux page management mechanisms. In a nutshell, Ambix considers the disparity in performance between DRAM and DCPMM, and decides new page distributions that ultimately lead to a higher application throughput and lower energy consumption. In order to achieve this, Ambix periodically identifies frequently modified and referenced pages, which benefit the most from the fast, but limited, DRAM tier. As a result, a larger portion of the workload's accesses are fulfilled by the DRAM tier, while DCPMM serves sporadically-read pages. We focus our efforts on a solution that requires minimal changes to the Linux kernel, and expands existing page placement mechanisms in order to accommodate the integration of DCPMM.

- As a **third contribution**, we evaluate Ambix, as well as relevant page placement alternatives, with several benchmarks from the NAS Parallel Benchmark (NPB) [21, 22] and GAP suites [23, 24]. To the best of our knowledge, this is the most comprehensive experimental evaluation of dynamic page placement solutions on a real system equipped with DCPMM memory. We show that Ambix outperforms both solutions proposed in past literature and placement options that are currently available in off-the-shelf DCPMM-equipped Linux systems, with an average speedup of 3.6x in

large footprint workloads, reaching a peak improvement of 10x, compared to the default memory policy in Linux.

1.3 Thesis Outline

The remainder of this work is organized as follows:

- Chapter 2 provides an overview of Linux's memory management subsystem, introducing key concepts and data replacement algorithms proposed in the past, for single-socket, NUMA, and HMA architectures.
- Chapter 3 describes how DCPMM is organized internally and configured, and presents a set of guidelines derived from devised benchmarks, which define how data should be placed in DRAM-DCPMM systems. It also discusses literature related to our work, based on the fulfillment of each guideline.
- Chapter 4 starts by outlining the main goals attained by our proposed solution, Ambix, and then describes its architecture and implementation.
- Chapter 5 details and discusses the evaluation of Ambix, comparing it to HMA-aware dynamic placement solutions proposed in past literature, and placement options that are currently available in off-the-shelf DCPMM-equipped Linux systems.
- Chapter 6 concludes the work, summarizing the main findings and discussing some directions for future work.

Chapter 2

Background

As the world transitions to the era of Exascale computing, we see NVM as a technology that presents us with an interesting proposition: "How can we capitalize on the improved scalability potential NVM offers, while mitigating its disadvantages?".

We introduce four main architectural designs that result from the integration of NVM, and that could be used to answer the proposition:

- Single memory tier with byte-addressable NVM, fully replacing DRAM. This configuration is seldom implemented, as current NVM technologies cannot fully replace DRAM in practice due to their limited endurance and higher read/write latencies.
- Single memory tier consisting of byte-addressable NVM and DRAM configured as last-level cache (LLC). In this architecture, DRAM extends the CPU caches in order to provide faster access to a larger set of pages, placed in the NVM. However, the DRAM cache is hardware-managed, and as such, applications have no control over which pages reside in it at any given time.
- Single memory tier with DRAM, and NVM configured as fast block-based storage served over either PCIe or NVMe. This architecture considers block-based NVM which, while still relevant as it is much faster than previous storage devices, does not take advantage of the byte-addressability NVM can offer.
- An HMA consisting of a multi-tiered memory architecture with DRAM and byte-addressable NVM in a linear or separate address space. In this architecture, both levels can be directly accessed by the programmer, making it the most attractive architecture for NVM-related work. As such, most research related to our work focuses on this configuration.

Previous literature that focuses on the described DRAM-NVM HMAs is divided in two major areas. The first area studies how systems can integrate a slower but larger level of memory for data placement [20, 25–46]. This involves rethinking traditional memory management strategies to consider an heterogeneous memory configuration [18, 47].

The second area leverages NVM's non-volatility in order to implement faster durability techniques for data objects. Literature in this area adapts traditional checkpointing algorithms and transactional systems to consider the lower access latencies offered by NVM, when compared to storage [48, 49].

Our work is focused on the first area. We will start by describing how Linux manages memory at a basic level in Section 2.1, and then advance to more complex NUMA- and HMA-aware data placement algorithms in Sections 2.2 and 2.3, respectively. We provide a critical overview of previous work, highlighting why it is unsuitable to effectively support NVM-equipped HMAs, along with an historical evolution of the memory management mechanisms provided by Linux, which we ultimately found to still be lacking when considering NUMA and hybrid memory architectures, compared to the SotA in these areas.

2.1 Memory Management

In this section, we describe the memory management mechanisms implemented in current OSes. We decide to focus on Linux, as it is the most widely-used OS for solutions related to our work. Moreover, our algorithm is also designed with Linux's mechanisms as its base.

Memory management is a critical part of an OS' design. Main memory is a scarce resource, and without memory management mechanisms processes: (i) compete for the available physical memory in order to leverage its lower latencies, and (ii) need to be aware of where they could write in main memory, since processes could inadvertently or maliciously write over other process' data.

Linux divides a process' address space in blocks, usually 4KB in size, called pages. It assigns each process its own virtual address space, which allows processes to perceive available memory as an exclusive, large, and contiguous structure, even though, in actuality, their data might physically reside in different memories or in storage.

Linux implements demand paging. In demand paging, all of a process' data is initially in storage. When it tries to access a virtual address, the kernel tries to find the respective page in a structure, called the page table. If it is not found, a page fault occurs, the page is placed in main memory, and the page table is updated with the the page's physical mapping.

Other paging mechanisms exist, such as anticipatory paging. In anticipatory paging, the OS estimates which pages are likely to be referenced in the near future and places them in physical memory before they are accessed. This technique reduces the amount of page faults at the cost of a greater memory usage, as it might place pages in memory that are not accessed. Nevertheless, commodity OSes default to demand paging, as it is seen as more performant in most use cases, due to its reduced memory and computational overhead.

Since processes perceive available memory as a value much higher than the existing physical memory capacity, they can allocate more data than can physically fit in memory. As such, Linux provides a special partition created in storage, called swap. The swap partition serves as a cache for pages recently evicted from memory. When physical memory is depleted or near depletion, the OS selects pages that are not currently in use and swaps them out to this partition.

Linux uses a parameter called *swappinness*, which defines at which rate and in which conditions the

kernel tries to evict pages from physical memory to swap (swapping). Higher values of *swappinness* correlate to more aggressive swapping, meaning that the kernel swaps pages more eagerly. In contrast, with a *swappinness* of 0, the kernel only swap pages when needed, i.e., when there is no available space in main memory. Performing swapping eagerly is relevant due to a set of principles called *locality of reference*.

2.1.1 Locality of Reference Principles

By default, the Linux kernel sets *swappinness* to a value higher than 0. This is done to swap out pages eagerly, before physical memory is depleted. When a page that is in storage is accessed, the kernel tries to place it in main memory. If physical memory does not have sufficient space, a page that the kernel expects to not be accessed soon is reclaimed to make space for the new one. This introduces a computational overhead compared to directly allocating the page when there is free space.

Eager swapping prevents the initial condition from happening, by evicting pages that are not expected to be accessed soon before memory is depleted, and therefore always allow new pages to find free space in memory. Although this comes at the cost of not fully utilizing memory capacity, placing the recently accessed page in memory is important, as processes tend to access the same memory locations frequently over a short period of time. This principle is known as *temporal locality*.

Another important locality of reference principle is called *spatial locality*. The principle dictates that an access to a virtual address is usually followed by accesses to addresses within its vicinity. This is visible in many algorithms which perform common operations, such as array initialisations and iterations, where accesses are either sequential or have a repetitive pattern.

The *spatial locality* principle applies in paging mechanisms, as an access to a virtual address causes a full block of contiguous data, i.e., page, to be brought into physical memory. Moreover, anticipatory paging mechanisms additionally bring pages within the accessed one's vicinity, giving a greater relevance to the principle.

2.1.2 Memory Management Unit

The memory management unit (MMU) is a memory controller chip, which is integrated into the CPU.

The MMU is responsible for translating virtual into physical addresses upon request by a running thread. This is done by accessing an OS-maintained page table.

The MMU also stores and accesses an associative cache, known as the translation lookaside buffer (TLB). The TLB improves translation overhead by storing the page table entries (PTEs) of recently referenced pages.

Translation Lookaside Buffer

Before accessing the page table, the MMU tries to find a requested virtual address' respective PTE in the TLB. If the MMU is able to find the PTE, a TLB hit occurs, and the corresponding physical address

is sent back. Otherwise, a TLB miss occurs, which causes the MMU to iterate over the page table, and add the found entry to the TLB.

Iterating over a page table involves traversing multiple entries, which can become prohibitively expensive in memory-intensive scenarios. Due to the *temporal locality* principle, the TLB speeds up translation greatly, since it is predicted that the virtual address was recently requested, and therefore its translation is expected to be present in it.

Page Table

The page table contains entries which store virtual to physical address translations. Each PTE also stores bits that provide information on page protection, and a present, dirty and reference bit, all of which are maintained by the kernel. Since virtual addresses are not unique, page tables are exclusive to a single process.

The present bit indicates if a page is currently in memory and has a valid mapping. If the present bit is unset, the current information in the PTE is invalid, and as such its contents are ignored.

The dirty, or modified, bit is set on page modifications, and is used to indicate that a page was written to since placed in memory. When page has its dirty bit set, storage contains an older value, and, as such, the updated value must be written back before it is evicted.

The reference bit is set on page accesses. It is used to indicate that a page was accessed recently, and used by multiple page replacement algorithms, namely the one operating in the Linux kernel, to decide which pages to evict.

When an address translation is requested and a TLB miss occurs, the MMU iterates over the requesting process' page table until it finds the mapping it is looking for. If it is unable to find a valid translation in the page table, it raises an exception, known as a page fault, which is handled by the kernel.

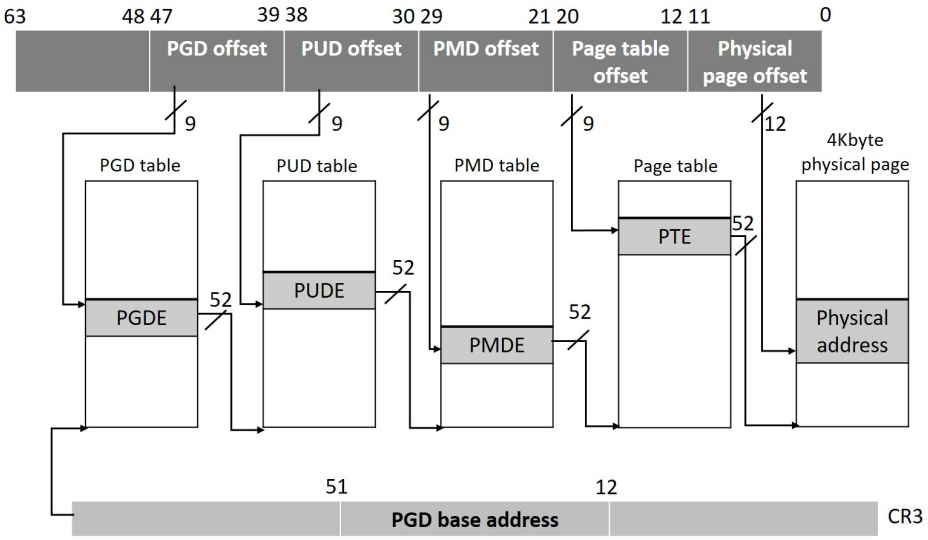


Figure 2.1: Linux's page table layout. [50]

Instead of creating a single page table containing a PTE for each possible virtual address, Linux

instead implements a multi-level structure, containing page directories, where the lowest-level directory points to multiple page tables [51]. This is known as a multi-level page table.

Linux subdivides a virtual address into multiple parts, which yield offsets. The offsets are used to navigate multiple structures, until the physical translation of a virtual address is found.

This mechanism is illustrated in Figure 2.1, which depicts a 4-level implementation. This implementation creates a hierarchical structure, where initially only the upper level table, known as the page global directory (PGD), is allocated.

If the MMU raises a page fault, the kernel iterates over the offsets of the faulted page's virtual address, using each offset to locate the entry that contains the root of the next level directory. If the entry is empty, the kernel allocates the lower level directory and fills it with the new directory's root address. When it reaches the last level, i.e., the page table, it then creates a PTE with the faulted page's physical address.

Conversely, in a scenario where a thread requests a virtual address that has a valid translation, the MMU is eventually able to find its respective PTE. Then, it uses the least significant bit offset of the address to locate the requested data inside the physical page, and returns it to the requesting thread.

Since in multi-level page tables the lower levels are only mapped when a page is placed in memory, the memory requirements are lower than in a single-level approach, when processes only use a part of their virtual address range. This is a common scenario in 64-bit architectures, where each process is able to allocate multiple terabytes of virtual memory (2^{48} pages). However, this comes at the cost of performing multiple memory accesses to find a requested translation.

Page Fault

Handling a page fault depends on why it was raised by the MMU, i.e., its cause.
Two main types of page faults exist:

- Major: A major, or hard, page fault results from referencing a page that is not currently in memory. Major page faults are common when applying demand paging, and are especially frequent when a process starts or tries to access new pages. When a major page fault occurs, the requested page is fetched from backing storage into main memory, and the page table is updated with the new information.

- Minor: A minor, or soft, page fault occurs when the requested page is in memory, yet the requesting process has no valid mapping in its page table. Common scenarios for minor page faults are when a process tries to access a page: from a loaded shared library, for which it does not currently have a mapping; that is marked protected for the type of request performed; or that was recently selected for being reclaimed but has not yet been cleared from memory. Since the page is already present in memory, minor page faults are computationally cheaper to handle. In this scenario, the page table is directly updated with the new mapping, without resorting to backing storage.

When a process tries to access an address outside of its virtual address space, or an unallocated one, an invalid page fault occurs, since the page is not found neither in memory or in storage. In this scenario, the kernel handles the page fault by raising a segmentation fault, which results in the offending

process' termination. Invalid page faults are generally related to programmer errors, and as such, are not common during normal execution.

2.1.3 Page Replacement Algorithms

Multiple page replacement algorithms have been proposed in past literature. They define which pages should be swapped out when needed, with varying degrees of complexity. We will describe the main algorithms proposed, as well as some variants that optimize them. All algorithms described in this section are thought out for a non-HMA architecture, consisting of a DRAM memory and swap cache.

Bélády's Algorithm

We mention Bélády's algorithm [52], also known as OPT, as it provides the theoretical best page replacement policy.

When a page needs to be swapped out, the OS selects the page that would be accessed the farthest from the current point in time. It cannot be implemented in general purpose OSes, as the algorithm requires either predicting future accesses or being able to perform a static analysis of all processes running in the system, therefore knowing beforehand which and when pages would be accessed.

Nevertheless, OPT can be used as a benchmark for other page replacement algorithms. If an algorithm performs closely to OPT i.e., the theoretical best, then it is considered to be near-optimal.

Least Recently Used

LRU is a family of traditional placement algorithms. The most basic implementation of the algorithm consists of a linked-list, or stack, that stores pages. In this implementation, a page access causes the OS to promote the page entry to the front of the list. When a page needs to be evicted, the page at the back of the list is selected, i.e., the least recently used page.

Optimized LRU implementations differ but usually rely on a logical clock implementation, incremented at every page access. The clock value, sometimes called *age*, is kept for each page. *Age* stores the logical clock at the last page access, meaning that pages with a lower *age* have not been accessed for longer. The OS replaces the page with the lowest *age* from memory. This avoids reordering the list at every page access, but still requires updating the page's *age* at every page access, which is both expensive to the CPU and requires special hardware routines that are able to intercept every access. Furthermore, the *age* entry should be stored in multiple bits, to avoid clock overflow, which increases the memory requirements of the algorithm. Although the pages selected by the algorithm are near optimal, LRU implementations rely on intercepting every page access, which causes significant performance degradation in current architectures. As such, commodity OSes, such as Linux, implement algorithms which are approximated to LRU, but instead rely on periodically updating the pages' information.

A modification to base LRU, named LRU-K [53] tracks the time of the last K accesses, and decides to keep the most frequently accessed pages in memory, by estimating the reuse distance based on the intervals between these times. The reuse distance is a metric that allows estimating the access

frequency of a page, where lower intervals between accesses correlate to pages with a lower reuse distance, i.e., higher access frequency.

For example, for K=2, the algorithm has been shown to improve the base LRU strategy by deciding to reclaim less frequently accessed pages with a more recent last reference time instead of the least recently referenced page. For K $>$ 2, the algorithm performs well for stable access patterns, when compared to OPT. However, larger K values introduce additional tracking data, which leads to a greater memory and computational requirements. Adding that to the fact that LRU-K still implements a logical clock in order to track the pages' age, where CLOCK or second-chance have been shown to incur less overhead while performing near OPT, it is currently not implemented by any commodity OS.

First-in First-out

FIFO is a simple algorithm that places pages in a queue at first access. When a page needs to be swapped out, the OS selects the page that was placed first in the queue, i.e., the oldest page. The algorithm performs poorly in real-world scenarios, as a page hit does not promote a page to the top of the queue. This means that the bottom page is always selected for replacement, even if it was accessed more recently than newer pages.

FIFO is still used in scenarios where the hardware is limited, such as in older systems, but has since been made less relevant by hardware improvements over the past decades. For example, past versions of Windows implemented a FIFO variant in multiprocessor systems [54], due to its simplicity when trying to maintain coherency in the TLB caches.

Second-chance

The Second-chance algorithm is a modified version of the FIFO algorithm. It improves FIFO by providing a second-chance to the oldest page in the queue, that would otherwise be selected for replacement in the traditional FIFO algorithm. Instead, when trying to select a page from the queue, it checks the reference bit of the page. If the reference bit is set, the algorithm unsets the reference bit and places it at the front of the queue, giving the page a second-chance. The algorithm proceeds to check the reference bit of the next oldest page until it finds one with the bit unset, at which point it selects the page for replacement.

For example, in a simple scenario where all pages have their reference bit set and no further accesses are performed, it will go through the queue once, ending up selecting the original oldest page, as it will now have its reference bit unset.

Second-chance algorithms combine the queue mechanism presented in traditional FIFO with each page's reference bit, which performs well against OPT.

CLOCK

CLOCK [55] is a variant of the second-chance algorithm. Similarly to second-chance, it implements a reference bit per page, which is set when the corresponding page is accessed.

When a page needs to be evicted from memory, CLOCK iterates over a circular list of pages and unsets their reference bits until it encounters the first page with the reference bit unset. At this point, the page is considered the least recently used and selected for eviction.

As CLOCK implements a circular list of pages, it has no need for reordering. Instead, CLOCK saves the index of the page that was last selected for eviction and uses it as the start for the next eviction operation.

Although CLOCK was first introduced over 40 years ago, it is still relevant and widely used today, namely by Windows 10 [56] and the latest Linux kernel [56, 57].

CLOCK-Pro

Rik van Riel proposed a kernel patch to use an optimized version of CLOCK to manage OS-level page placement [58], named CLOCK-Pro [59]. It implements reuse distance by only using the reference bit of each page, an improvement over LRU-K. The reasoning behind the patch is that the implemented LRU-approximation algorithm is inefficient for common data operations, such as array initialisations and one-use operations, which set the accessed page's reference bit and fill up physical memory with pages that might not be used in the near future. LRU and the base variant of CLOCK operate under the assumption that a page that was referenced recently will be referenced again in the near future, a phenomenon called temporal locality. Instead, the reuse distance implemented by CLOCK-Pro is able to detect which pages are more frequently accessed, and opt to keep them in memory in lieu of pages that were either accessed only once or which second to last reference occurred longer ago.

The CLOCK-Pro[59] algorithm separates pages into 3 categories: **hot** pages, which are pages that are frequently accessed; **resident cold** pages, which are pages that have not been accessed since the algorithm changed their category from hot to cold; and **non-resident cold** pages, which are cold pages that were previously resident but have since been evicted from memory.

Keeping non-resident pages in the circular list allows the algorithm to give a second-chance to pages that have been recently evicted from memory, if they are accessed shortly after being reclaimed. Otherwise, the algorithm eventually clears them from the list.

Also, marking pages as hot allows the implementation of the reuse distance. A page is only evicted if it is cold and has its reference bit unset. As such, the algorithm must first alter a page's category to cold before deciding to finally evict it.

Not Frequently Used

The NFU algorithm tracks page accesses in a time frame. It periodically increments a per-page access counter for every page accessed in the last interval. When a page must be replaced, the algorithm selects the page with the lowest number of accesses.

NFU is not implemented in commodity OSes for two major reasons:

- Incrementing an access counter for each page access is both expensive to the CPU and requires the system to have a special hardware counter for the effect, similarly to the logical clock imple-

mentation of LRU,. On the other hand, using a reference bit has been proven to provide a good estimate of the optimal page for replacement, and it is already implemented in mainstream architectures.

- It is unable to estimate if a page was accessed recently, as it provides an absolute count of accesses since it was first placed in memory. This leads to poor performance. For example, if a page is accessed many times in a short time frame and then never accessed again, newer active pages with a lower access count will be preferred for replacement.

Aging

In order to mitigate the second issue of the NFU algorithm, the Aging algorithm was proposed, which provides some information on the temporal access pattern of a page. It differs in implementation in how the access counter is updated on page accesses. Instead of simply incrementing the counter, the Aging algorithm first performs a shift left, which divides the counter by 2, after which it is then incremented to reflect the new page access. The selection process is equivalent to the NFU algorithm.

Although Aging is able to solve the temporal problem of the NFU algorithm, it is still rather expensive to the CPU and requires special hardware counters. As such, it is not frequently implemented in current OSes.

Not Recently Used

The NRU algorithm uses the reference and dirty bits in order to select which pages to evict. It periodically clears the reference bit of each page. When a page needs to be evicted, it selects the first page found that has both its reference and dirty bits unset. Otherwise, it looks for less optimal pages, relaxing the criterion each time no pages are found:

- Reference bit unset and dirty bit set
- Reference bit set and dirty bit unset

If no pages that meet the previous criteria were found, it randomly selects a page with both bits set.

2.1.4 Virtual Memory Management in Linux

Current Linux versions use an LRU-approximation page replacement algorithm in order to manage virtual memory [56, 57]. Even though it is considered to be LRU-based, the algorithm provides multiple improvements over the computational and architectural requirements of the LRU implementations, by using the CLOCK algorithm to reduce the overhead of the mechanism and allow efficient page selection without the need to intercept every access, like on the logical clock LRU implementation.

Linux divides pages in two lists, an active and an inactive list. The active list maintains pages that are considered to be currently in use by one or more running processes, while the inactive list maintains pages that have not been accessed recently, and as such are suitable for being reclaimed.

Linux keeps a balance between the length of the lists, called *inactive_ratio*. The ratio is set based off of total available memory and represents the target proportion between the size of both lists. For example, if the system has a total memory of 1GB, the ratio is 3:1, meaning that 25% of the pages should be in the inactive list.

Whenever a page is first referenced, it is placed in the active list. When the ratio of the inactive list falls below the configured threshold, the least recently used pages of the active list migrate to the inactive list. This is achieved by iterating over the active list's pages and checking if each page has been recently referenced. If it has, the kernel maintains the corresponding page in the active list, therefore giving it a second chance. Otherwise, it unmaps the page and demotes it to the inactive list. If a page placed in the inactive list is accessed again, a minor page fault occurs, which causes the kernel to remap the page and promote it to the active list.

If the current free memory falls below a threshold, based off the configured *swappinness* value, the kernel launches a daemon, called *kswapd*. The daemon reclaims as many pages from the inactive list as necessary until it restores the threshold, starting from the older ones, i.e., the pages at the bottom of the list. These pages can either be swapped out, if the system has a configured swap cache or simply evicted from main memory. The inactive list improves reclaim performance, as when memory needs to be freed there already exists a subset of pages that have been demoted to this list, and therefore can be freed from memory without the overhead of the CLOCK algorithm. If all pages in the inactive list have been reclaimed but were insufficient to restore the threshold, the kernel then decides to directly reclaim pages from the active list that have not been recently accessed, following the same criteria as in the active to inactive list demotion.

2.2 NUMA-Aware Mechanisms

In this section, we describe the memory management mechanisms provided by Linux when a system has multiple sockets and physically distributed memory (NUMA).

We choose to introduce and discuss the NUMA architecture, as it shows how a novel memory architecture can impact an OS' design. NUMA architectures were first proposed in the 90s, and are still widely used today, most notably in servers and supercomputers dedicated to HPC.

NUMA provides fundamental improvements over the single-socket architecture, but relies on both OS and hardware support in order to achieve its fullest potential. Its impact drove Linux, as well as the majority of widely-used OSes, to extend their memory management mechanisms in order to consider the possibility of a multi-socket system, where each socket's CPU can directly communicate with all system memory, with different latencies dependent on the accessed memory. Still to this date, solutions that propose efficient memory management techniques specifically tailored to NUMA architectures are being published. We consider this area to be one of the main inspirations to the data placement mechanisms proposed in recent literature dedicated to HMAs.

Moreover, HMAs are a natural extension to NUMA architectures, with HMA-equipped systems maintaining the same multi-socket and decentralized memory characteristics of NUMA, with the added com-

plexity of multiple memory technologies within each socket. Therefore, even though the concepts introduced in this section were designed for NUMA architectures composed of identical memory, usually populated with only DRAM, our overview in this section will serve as a base to introduce HMAs, later in this chapter.

We will start this section by providing an overview of NUMA-related concept, and then introduce important NUMA-aware data placement literature.

2.2.1 NUMA Architecture

Shared-memory multiprocessor systems are composed of multiple processors which share access to all memory in the system [60]. In these systems, main memory can be organized in two configurations: (i) centralized, which offers an uniform access latency to all CPUs, called uniform memory access (UMA); and (ii) distributed, where each CPU is associated to a subset of the system's memory, called non-uniform memory access (NUMA).

The vast majority of current NUMA architectures are cache-coherent (ccNUMA). Cache coherency in a NUMA system is achieved by implementing a write-invalidate cache coherence protocol, which maintains a consistent view of memory addresses cached in multiple CPUs [61]. The following overview of NUMA always assumes a ccNUMA architecture.

In NUMA architectures, although main memory is physically distributed, CPUs can still access all memory in the system. In these systems, Linux implements a logical node view, where a node is characterized by a CPU and the memory banks closest to it, usually configured as a node per socket. As the name indicates, memory access latency can vary, depending on the location of the accessed memory relative to the processor. If a CPU core accesses memory from within the same node, it is defined as a local access. If the core instead accesses another node's memory, the access is classified as remote.

Communication between a CPU and memory remote to it is achieved via point-to-point interlinks, such as Intel's QuickPath Interconnect [62] or AMD's HyperTransport [63], which allow inter-socket data operations.

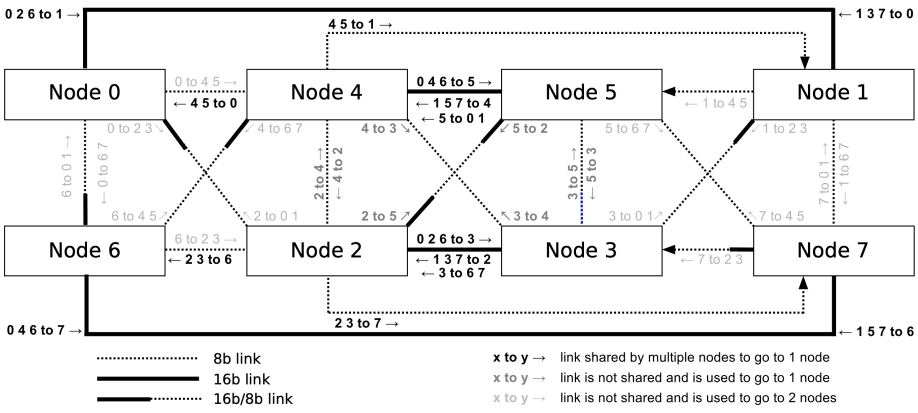


Figure 2.2: Asymmetric NUMA system [64].

Figure 2.2 shows the routing of the NUMA interlinks and their respective bandwidth in an 8-node system. We can see that some buses are unidirectional (represented by the arrows), some are limited to 8-bit bandwidth and some even present different bandwidths depending on the direction of the connection. Furthermore, we see that some interlinks are shared between different node pairs. For example, the bi-directional 16-bit bus between nodes 0 and 1 is also shared for requests to node 1 by nodes 2 and 6, and requests to node 0 by nodes 3 and 7.

Linux defines the latency difference between node pairs as *distance*. *Distance* values range from 10 to 255, where 10 is the base latency, which is the latency of a node's local accesses, and 255 represents that there is no connection between the node pair, i.e., the nodes do not have a bidirectional interlink path between them. For a distance d, between 10 and 254, the access latency between the node pairs is estimated to be $d/10$x higher than the latency of a local access. For example, if node 0 has a *distance* of 20 to node 1, then accesses between the CPU of node 0 and the memory of node 1 are $\sim$2x slower than accesses to its local memory. The *distance* values are hardcoded by default, and are provided by a firmware-dependent structure in the Advanced Configuration and Power Interface (ACPI) called System Locality Information Table (SLIT).

These values are not intended to provide an accurate measurement of a node's latency difference to another node. In fact, memory latency tools and benchmarks such as Intel's Memory Latency Checker (MLC) [65] often produce significantly different values than those given by the SLIT table. Nevertheless, the default values provide an estimate of a remote memory's performance and as such are used by both Linux and NUMA-aware data placement algorithms when deciding where data should be placed in a system.

Memory access latency is not only influenced by the distance between the CPU where a process is running and the physical memory DIMM it accesses, but also due to the amount of accesses performed simultaneously with the same interlinks, to DIMMs connected by the same memory channel, or to each individual DIMM.

Memory channels connect physical memory to a memory controller, providing data flow between the CPU and memory, in both directions. Each channel is responsible for a subset of the system's memory. While a memory controller can be associated to multiple memory channels, each memory DIMM can only be associated to one memory channel.

When many accesses are performed in parallel, the channel's data rate, or throughput, stops increasing. This is due to data congestion on the channel that processes the data flow in both directions, and leads to a drop in average access latency. Similarly, each DIMM and interlink has a maximum throughput, and may become saturated when too many accesses are requested to it.

The hardware that limits throughput from increasing defines the memory bandwidth. Bandwidth is also dependent on the access pattern of a tested workload. For example, MLC measures the sustained memory bandwidth of different access patterns by injecting data into multiple system nodes, with the goal of finding the distribution that maximizes throughput.

Modern motherboard designs often implement multiple memory channels, or buses. Adding more memory channels result in an increased bandwidth between a CPU and its local memory. Multi-channel

architectures reduce each channel's bandwidth stress, by allocating data evenly across channels. Furthermore, data may also be evenly distributed across each individual DIMM, or memory bank, within each channel. However, a more scalable solution is to schedule a process and its data to idle or unstressed remote nodes.

2.2.2 From Latency-centric to Contention-aware Data placement in NUMA

In the past, NUMA-aware data placement mechanisms for commodity OSes, such as Linux, relied on minimizing the amount of remote accesses to determine where in a system a process should run and have its data placed. This is known as a locality-based approach, since the goal is to maximize the local access ratio of the running threads. In scenarios where multiple large footprint applications run in parallel, these systems would schedule applications to different sockets in a way that a higher volume of their data is accessed locally. This is due to the massive latency and bandwidth difference between local and remote accesses in older systems. These systems benefit from maximizing the ratio of local accesses for all applications, as they can serve data requests faster, even if the utilization of remote nodes is lower in comparison.

Remote access latency has since been greatly improved. While in the 90s we could expect a remote access to incur a latency overhead between 4 and 10x higher [66], modern NUMA systems reduce the penalty to less than 30% [67, 68].

The faster remote access latencies lead to a shift in the what was perceived as the optimal data placement strategy. Due to the reduced remote access overhead, other metrics, which were less significant in comparison, e.g., channel, bank, and CPU load imbalance, became significant factors when deciding where to place both data and threads in a system. Modern NUMA-aware data placement algorithms perform these decisions not only taking into account the remote latency overhead, but also metrics that relate to bus contention, such as MMU load imbalance, interconnect link usage, and other bandwidth-related metrics, which have been shown to deteriorate the throughput of an application.

Recent literature defines contention in the memory channels and MMUs as one of the major bottlenecks in performance [64, 67, 69]. As such, these solutions may intentionally place a subset of a thread's pages in nodes remote to it, as, despite these pages incurring a latency penalty compared to local accesses, the new page distribution can ultimately improve total throughput if the thread's local memory is saturated.

These contention-aware mechanisms have been shown to greatly outperform locality-based mechanisms in multi-threaded workloads which heavily share resources. Despite this fact, the current NUMA mechanisms implemented in Linux still consider data locality as the primary factor when deciding page and thread placement.

2.2.3 Linux NUMA Subsystem

Linux manages memory independently for each NUMA node, with the mechanisms described in Section 2.1.4. When a node is near to depletion, *kswapd* evicts pages allocated in it, not affecting the other

nodes' pages.

NUMA Memory Policy

Linux allows users to change the memory policy at different scopes: global, which defines the default allocation policy for all running processes; process-level, which allows defining the policy for a single process; and ranges of a process' virtual memory, called virtual memory areas (VMAs). This can be achieved by manually changing the kernel's configuration, resorting to *numactl*'s command-line interface, or programmatically via the *libnuma* library [70].

By default, the global memory policy Linux follows for running applications is *node local*. The *node local* policy prioritizes the allocation of a new page in the memory local to the requesting thread, as it provides the lowest latency and highest bandwidth values in ideal conditions. The policy favors local accesses, i.e., it is a locality-based approach, and tries to minimize the use of interconnection links. If there is no space available, the policy defaults to allocating pages on the least-distant node with available memory.

Other available memory policies are defined as follows:

- *Interleave*. The *interleave* policy distributes pages among all specified nodes evenly. The allocations are performed round-robin, meaning that the kernel alternates the allocation node at every access, leading to an even distribution of the pages. Interleaving pages can outperform the *node local* policy in scenarios where allocating all data to the local node causes significant contention on a single memory controller. In this case, the added throughput of the unstressed remote nodes can prove beneficial to the performance of the workload [67]. However, even though pages are balanced across nodes, there is no guarantee that its accesses will be, as most workloads access some pages more frequently than others. Moreover, the ideal page ratio is seldom even, and in many workloads varying the percentage of pages allocated in each node outperforms the even distribution [69].

- *Bind* and *preferred*. Both policies allocate data to the first specified node that has free space. The *bind* policy restricts page allocation to the specified nodes, while the *preferred* policy allows allocation to other nodes if no free memory is available in the specified ones. These policies are not frequently used, as *node local* or *interleave* provide better performance in most scenarios. Nevertheless, they can be useful for debug or benchmarking tools, as they allow testing the performance of different node configurations.

Figure 2.3 shows the traffic pattern of a multi-threaded execution of the *streamcluster* benchmark, part of the PARSEC suite, which is a widely used suite for HPC performance measurements. It represents the ideal scenario for the *interleave* policy, where data requests, visualized in percentage, are perfectly distributed across nodes. In this scenario, *interleave* outperforms the *node local* policy by a factor greater than 2 [67]. The *interleave* policy is able to decrease the imbalance of the NUMA interlinks substantially (seen by the thickness of the arrows), which improves overall performance considerably.

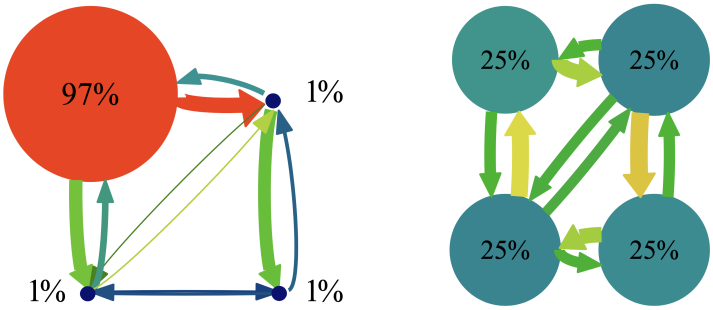


Figure 2.3: Traffic imbalance under *node local* (left) and *interleave* (right) policies for the PARSEC's [71] *streamcluster* benchmark [67].

However, *node local* has been shown to perform better for other workloads, including the majority of the benchmarks in the PARSEC suite [67]. This means that the memory policy choice is not straightforward and that the user is left with the task of tuning the memory policy that performs best for certain tasks, at the process or even VMA level.

AutoNUMA

The latest Linux kernels provide an automatic page and thread balancer, called AutoNUMA [72]. It presents several optimizations on the memory policy-tuning approach, by managing both pages and thread allocation automatically, without the need for user interaction.

The algorithm periodically scans every page table, clearing the present bits and setting a protection bit (_PAGE_PROTNONE), which makes the MMU view the page as resident but not accessible, therefore causing a minor page fault on the first subsequent access. When the page fault occurs, AutoNUMA registers both the node of the accessing thread and the node where the page is currently placed, and the page is marked as present again.

Two structures are kept: the first is a per-thread structure, which stores the nodes of the pages last accessed by the thread. The second is a per-page structure which simply stores the node of the thread that accessed it last.

The algorithm consists of two main mechanisms:

- The first mechanism observes the node access pattern of each thread. If a thread accesses a remote node's pages more frequently than the local ones, then the thread is scheduled to migrate to that node. Since thread migration incurs significant CPU overhead, AutoNUMA only migrates a thread when it predicts that placing the thread on the node where it performs the most accesses will ultimately improve performance, due to the decrease in remote accesses.

- The second mechanism is related to page migration. If a remote node's thread causes a page fault, the accessed page is queued to migrate to that node. If another node accesses a queued page before the routine migrates it, then the page is cleared from the queue.

For some applications, enabling AutoNUMA provides significant performance benefits without the

need for manual tuning [67, 69]. However, it performs worse when compared to contention-aware approaches, in workloads prone to high contention [67].

2.2.4 Contention-Aware Approaches

In this section we describe in describe two approaches which extend locality-based principle with metrics that are related to bus contention, although we identify other contention-aware solutions, such as DINO [73] and N-MASS [74], as well as studies [75] [76] related to how contention affects a system's performance.

Carrefour [67] and *AsymSched* [64] detail the importance of contention management in modern NUMA systems. *AsymSched* further improves contention management by consider the NUMA interlinks' bandwidth asymmetry, which is common in modern systems.

Carrefour

Carrefour [67] manages traffic, i.e., movement of data in a NUMA machine, to improve the system and its running applications overall performance when compared to approaches that only consider locality. Locality still plays a role in the algorithm, as local accesses incur a lower idle access latency. However, the algorithm's capability of detecting channel imbalances allows it to prefer remote nodes in these scenarios for placing or migrating data. *Carrefour* also schedules threads in a way that minimizes contention.

The algorithm has 4 main mechanisms.

- Page Co-location: Similarly to the *node local* policy, the algorithm prefers local accesses by relocating the pages to the node where they are most accessed.

- Page Interleaving: When the algorithm detects MMU imbalance, it distributes the pages across multiple nodes, such that the accesses are physically distributed. This is achieved at the VMA-level, which means that a process might only have part of its pages interleaved.

- Page Replication: In scenarios where there is sufficient memory available, the algorithm replicates a process' pages across several nodes. Replication not only improves latency, since a page's contents are accessible locally by threads running on different nodes, but also has the added benefit of reducing interlink communications in read-intensive workloads. This comes at the cost of maintaining data coherency, since writes performed to a local copy must be replicated to all the nodes where it is replicated.

- Thread Clustering: The algorithm prefers scheduling threads that access the same data on the same node. This is applied only if the grouped threads do not exacerbate contention, meaning the selected thread configuration does not cause a significant MMU imbalance.

Carrefour is only enabled in memory-intensive scenarios, due to its sampling overhead. If memory traffic surpasses the defined threshold, then the replication, interleave and co-location mechanisms are also enabled based on the characteristics of the workload.

Replication is only enabled in read-intensive scenarios if sufficient memory is available. The replication decision is dynamic, meaning that if a system suddenly no longer meets the free memory criteria, then replication is disabled.

Interleaving is applied whenever memory channel imbalance surpasses a defined threshold. If the imbalance is low, then applications do not benefit from interleaving. However, interleaving can significantly improve performance in applications that would otherwise cause high memory channel imbalance, as seen by the default *interleave* policy in Linux [77].

The third and final decision is on whether on not to enable co-location. The mechanism is only enabled when the average local access ratio is below a threshold. When active, pages that are accessed exclusively from a single node are relocated to that node, therefore reducing the percentage of remote accesses. Since thread clustering balances threads across nodes in a way that bus contention is not prominent, co-location does not aggravate MMU imbalance.

With these mechanisms, *Carrefour* is able to adapt to different workloads dynamically, which eliminates the need for fine-tuning an application's NUMA policy, while improving performance greatly compared to AutoNUMA or manual policy tuning.

AsymSched

AsymSched [64] further improves the contention-aware mechanisms implemented in *Carrefour* by also considering how nodes are connected in a NUMA system. The NUMA interlinks in current and future systems may be asymmetric, meaning that the communication between different node pairs provide different bandwidth values. The asymmetric nature of these systems makes traditional thread scheduling and contention-related metrics proposed in past literature suboptimal or insufficient for these systems.

The algorithm adapts to the asymmetric interlink configurations presented in Figure 2.2 by placing communication intensive threads in well-connected nodes, and preferring placing their data locally or on nodes connected with a high-bandwidth path. Interlink contention is also considered, as data requests may require traversing interlinks shared with other remote threads.

Like *Carrefour* and previous contention-aware algorithms, *AsymSched* dynamically groups threads into clusters. These clusters are composed of threads that heavily share data among them. A weight value, given based on the amount of remote accesses is given to each cluster.

The algorithm then decides where to place each cluster, considering the maximum bandwidth that the placement decision can provide and the connectedness of the nodes. The clusters with higher weights are placed in well-connected nodes, leveraging the higher-bandwidth buses offered by them.

The selected placement is only applied if the migration overhead of the new placement is estimated to be compensated by the performance benefits in the long run. This is achieved by projecting the time it will take for the system to migrate the threads and their data to the new configuration and comparing it to the total running time of the workload, at the point of the decision. If the ratio is below a defined

threshold then the new placement is put in effect, and the threads migrate to the new configuration.

After the threads are migrated, the algorithm begins migrating a subset of each cluster's most intensive pages to the nodes defined in the new configuration. If, after a defined time, most accesses are still performed to nodes in the old configuration, then the algorithms performs a full memory migration, which migrates the remaining pages according to the placement decision.

2.3 Data Placement in HMAs

In HMAs, similarly to multi-socket NUMA systems, defining where data should be placed is not trivial. Each memory tier provides different characteristics, and can vary in terms of size, latency, bandwidth, and energy consumption.

Placing an application's data in the fastest tier has obvious benefits, such as a reduced average access latency, and overall increased throughput, due to its lower latency and/or higher bandwidth. On the other hand, less frequently accessed data may be better suited to a slower tier, when space is scarce. Moreover, if the fastest tier's bandwidth is saturated, distributing frequently accessed pages between the other available tiers may increase aggregate memory throughput, even if they offer slower latency. Bandwidth-aware placement can be achieved with mechanisms similar to those presented in the contention-aware solutions, presented in Section 2.2.4.

In order to allocate data to the appropriate memory tier, a plethora of viable solutions exist. We divide these solutions in two areas:

- Static placement: Approaches in this area profile an application's data accesses based on their access pattern, by executing one or more test runs. Then, a data distribution that maximizes a chosen performance-related metric is decided, which can be manually or automatically applied to the application, only then making it ready for an HMA-aware execution.

- Dynamic placement: Dynamic algorithms classify and migrate data between tiers online, i.e., during execution, without a priori knowledge about an application or its access pattern.

While dynamic placement is most commonly decided at page-level [20, 25–39], static placement solutions are usually implemented at object-level [40–46].

Existing HMA-aware solutions that are specifically designed for NVM-equipped systems, considering their asymmetry, are still uncommon in current literature [25–32, 37, 39]. Most commonly, these solutions also, or only, consider multi-channel DRAM (MCDRAM) [78], which is an on-package memory that fits above DRAM in the HMA tier hierarchy. Compared to DRAM, MCDRAM offers a smaller capacity (up to 16GB) but an increased bandwidth (up to 4x). Most importantly, its latency is similar to that of DRAM, with no read/write asymmetry.

Therefore, in trying to be as architecture-agnostic as possible, existing solutions that propose placement in HMAs are seldom aware of NVM's asymmetric read/write performance [33–35, 38, 40–46], as they are tailored to function with any tier configuration, e.g., MCDRAM-DRAM-NVM [33]. Even though we consider these solutions suboptimal for our chosen architecture, they present relevant mechanisms

that could be applied in asymmetry-aware solutions, where the majority identifies NVM as a possible configured tier [33–35, 37, 38, 40, 41, 43, 45, 46].

In the following sections, we start by considering generic HMAs, and present a high-level discussion about the common challenges and proposed mechanisms applied to all architectures composed of multiple memory tiers. Since the focus of this thesis is on DRAM-NVM HMAs, we then narrow the discussion to a more detailed survey of proposals for such HMAs, in Section 2.3.4.

2.3.1 Static Placement

Static placement solutions identify an application's frequently accessed data objects and define where each object should be placed, such that the application's throughput is maximized.

These solutions start by identifying and separating the application's data into multiple objects. Then, the objects are profiled, and sorted according to their access frequency, or a similar memory-related metric. Finally, the solution calculates where each object should be allocated, such that the decided distribution maximizes a chosen metric, such as throughput or energy efficiency.

Object Identification

A simple approach for static solutions to identify and separate an application's data into multiple object groups is to manually modify allocation sites, e.g., `malloc()` calls, with similar calls to the solution's API. For example, Unimem [41] requires allocation site replacement. However, this leaves the developers with the laborious task of individually identifying and replacing every allocation-related instruction in a possible huge code base.

Instead, more evolved approaches are able to identify the same allocation sites automatically, resorting to: (i) a static compilation pass, or (ii) an instrumented run. The former approach (i) can be achieved with the low level virtual machine (LLVM) compiler infrastructure [79], for instance, which enables solutions to pinpoint allocation and deallocation instructions in an application's source code. Membrain [42], Olson et al. [43], and OAM [46] use LLVM to identify and associate memory references to data objects. Solutions that identify objects through instrumented runs (ii) can use the PIN [80] framework, which is able to intercept allocation and deallocation calls and associate them to regions, or objects. PIN-based instrumentation is applied in the approaches proposed by Effler et al. [44], and X-Mem [45]. Servat et al. [40], also apply (ii) but instead use an in-house instrumentation tool that automatically finds and replaces allocation and deallocation instructions with their solution's API calls.

In all methods, allocation sites are associated to a unique handler, such as a tag, which identifies each allocated object group.

Object Profiling

Object profiling consists of characterizing an object, or object group, by quantifying: (i) the number of accesses that it causes to physical memory, usually measured by the number of last-level cache (LLC)

misses; (ii) its maximum resident size; and optionally (iii) the time that takes for the object to be freed after allocation, or lifetime.

We identify two main approaches to profiling: PIN-based instrumentation, and hardware-based sampling.

Besides allowing allocation and deallocation call identification, PIN enables measuring (i-iii) and is also leveraged in the same approaches that rely on allocation site identification via PIN [44, 45].

Solutions may also leverage hardware-based sampling mechanisms, such as Intel's Processor-Event Based Sampling (PEBS) [81], or AMD's Instruction-based sampling (IBS) [82], both of which are widely common in Intel and AMD CPUs, respectively. Similarly, these mechanisms can be used to capture all three metrics and associate them to the objects identified via LLVM.

The main benefit of the PEBS/IBS approach as opposed to PIN is a reduced test run execution time, at the cost of precision due to its sampling-based nature. For example, PIN instrumentation has been shown to increase execution time by up to 40x in X-Mem [45], while PEBS has a negligible overhead with lower sampling rates.

Placement Decision

After objects are profiled, the solution is left with a list of allocation sites (objects) that have an associated LLC miss count, lifetime, and size. At this point, the solution must sort and decide which data to allocate in each tier.

All identified solutions solve some variant of the classic 0/1 knapsack combinatorial optimization problem [83]. Simply put, the problem presents a knapsack with a defined weight limit, that must be filled with elements of varying weight and value. The goal of the problem is to insert a subset of elements that maximize the knapsack's aggregate value, without exceeding its weight limit.

Applied to the data placement scenario, we can consider the faster tier as the knapsack and the tier's size as its weight. Similarly, each object also has a defined weight, defined by its size.

However, when trying to quantify each object's value, two main approaches exist:

- Density: Calculated from the object's LLC misses divided by its size.
- Bandwidth: Divides an object's LLC misses by its lifetime instead, i.e., time between allocation and free calls, calculating its total bandwidth demand.

In both cases, the solution then calculates the best subset of objects to place in the faster tier.

Multi-Phase Profiling

One of the pitfalls of static placement strategies are their inability to react to workload changes. One way to mitigate this is to divide an application's execution into multiple phases, as proposed in [41, 44, 46]. These solutions profile and determine where data should be placed at each phase. This gives the algorithms a dynamic-like nature, as an object's value may change throughout the phases and cause it to be migrated to a different tier. Migration is done at the start of each phase by leveraging OS mechanisms [44], or with a helper thread [41, 46].

2.3.2 Dynamic Placement

Whilst the previous solutions decide where to place data by profiling the workload and identifying the most cost-effective data at a global or per-phase level before execution, approaches in this section make the same tier placement decision dynamically during execution, i.e., online [20, 25–39]. Dynamic placement introduces new challenges, such as adapting to workload changes during runtime in a way that maximizes throughput with minimal overhead, with little to no a priori knowledge about the workload.

Without prior profiling or allocation site identification, existing dynamic solutions are implemented at a page-level granularity, and commonly leverage the long established page management mechanisms in commodity OSes, described in Section 2.1.

In contrast to the identified static placement solutions, the majority of the identified dynamic placement literature is designed with tier asymmetry in mind, and are specifically proposed for DRAM-NVM architectures [20, 25–32, 37, 39].

Therefore, we continue this section by presenting a high-level overview of the mechanisms proposed in the identified online placement literature, detailing how the solutions tackle the additional challenges common to any HMA configuration, and later detail the NVM-focused solutions' implementation, in the following section.

Placement Overview

Page management policies for HMAs include, but are not limited to deciding where a new page should be initially placed, and when to migrate or evict resident pages.

When a page is initialized and fetched from backing storage, solutions must decide where to insert it. Pages recently placed in memory should be prioritized in the faster tier, due to a higher probability of being heavily accessed in the near future, according to the *temporal locality* principle. However, as new pages are inserted, the faster tier may become full.

Therefore, dynamic placement algorithms may keep a buffer of free space in the faster tier, demoting pages eagerly when free space falls below a threshold [20]. This is applied in the Linux kernel's active-inactive lists mechanism, which not only pre-selects pages that it deems inactive, or cold, but also demotes them eagerly.

Most commonly, HMA-aware solutions do not implement a buffer mechanism. Instead, when fast memory is full, these solutions exchange a new page with a resident one, demoting the latter to a slower tier [26–28, 32, 35, 36].

We also identify approaches that do not strictly follow the *temporal locality* principle. From these, we identify four alternatives to initial placement: (i) follow the default allocation policy, placing pages in the fastest tier with available space [31, 39]; (ii) knowing each tier's maximum bandwidth for a certain workload, distribute pages proportionally [38]; (iii) place the new page in a faster space-depleted tier only if a cold page can be found [29, 30, 33, 34]; (iv) place pages directly in the slower tier and promote them based on their access frequency [37].

In contrast to placement algorithms designed for the traditional DRAM-only architecture, approaches

for HMAs can choose to demote pages in a space-depleted faster tier to a slower tier, deciding not to evict it to the swap partition while still freeing up space from the faster tier. Similarly, pages may also be promoted, and thus HMA algorithms must define a migration policy.

In dynamic algorithms, pages can be chosen to migrate between tiers based on the fulfillment of one or more criteria [20, 25–37, 39]. The proposed criteria varies in existing literature, but frequently relies on measuring the absolute or temporal relevance of a page, based on its access pattern, where the most frequently accessed pages are commonly prioritized for being promoted or kept in the fastest tier. We elaborate on some existing page classification techniques in Section 2.3.2.

If all memory tiers in the HMA become full, algorithms must decide which pages to evict to secondary storage. Most proposals only evict pages from the slowest tier [20, 25–31]. Still, proposals such as AC-CLOCK [32] may also evict pages that currently reside in the fastest tier if their access pattern does not justify retaining them in memory.

Page Classification

In this section, we present an overview of existing mechanisms proposed to classify pages We will shortly discuss how these techniques are applied to steer data placement, although we leave implementation details to be comprehensively described in Section 2.3.4, which narrows the scope to DRAM-NVM HMAs.

Prior literature presents two main page classification techniques:

- **Access Tracking**: Approaches that propose access tracking rely on intercepting a workload's accesses so as to order pages by their access frequency or trigger them to migrate after surpassing some defined threshold. We divide access tracking methods into three subareas:
  - Read/Write Tracking [20, 25–35].
  - TLB Miss Interception [36, 37].
  - LLC Miss Sampling [39].
- **PTE Monitoring**: Existing solutions may also leverage access-related PTE bits, such as the reference and dirty bits [25, 26, 29, 30, 32, 37].

These techniques are sometimes combined. For example, existing solutions use both read/write access tracking and PTE bit information in order to classify pages [25, 26, 29, 30, 32]. Memos [37] also leverages PTE bits, but instead compounds it with TLB miss information.

Solutions that exclusively implement access tracking are often based on the LRU algorithm [20, 27, 28, 31, 33–35]. In its traditional implementation, accesses to a page promote it to the top of the LRU queue, and the pages at the bottom are evicted first. In HMAs however, multiple tiers exist, and thus, the identified solutions propose per-tier LRU queues. In each LRU queue, pages that are placed closer to the top are considered to be more ideal to either retain or promote to a faster tier, and therefore are classified as more intensive than pages at the bottom.

Tracking R/W accesses to a page can be achieved with hardware modifications. Namely, DualStack [31] proposes MMU changes, in which the chip keeps track of all requested page translations, associating each page to a read and a write access counters. The same purpose can also be achieved without relying on modified hardware. For instance, one can intentionally cause minor page faults, which can be done by clearing a page's present bit in the PTE, flushing its entry from the TLB, and instrument the page fault handler. A simple access tracking mechanism is applied in the kernel's active-inactive lists, in which an access to a page in the inactive list causes a page fault, and consequentially its reinsertion into the active list.

A possible extension to this technique could lead to multiple consecutive access being intercepted. Namely, if shortly after a tracked page that caused a page fault was marked invalid again, i.e., had its present bit cleared, this would result in the following access being intercepted. However, this technique would suffer from lack of precision, since accesses during the window between a page's fault and subsequent invalidation would not be tracked, therefore causing the method to have a sampling-based nature.

Another access tracking-based option to classify pages is to intercept TLB misses. TLB misses can be used to estimate a page's number of LLC misses, by determining the number of times the MMU failed to find its translation in the TLB. Both Memos [37] and Thermostat [36] leverage BadgerTrap [84], which instruments TLB misses online.

Salkhordeh et al. [39] present a third access tracking alternative, which leverages PEBS in order to monitor page access frequency by collecting information on the number of LLC misses caused by each page. Unlike the static solutions introduced in Section 2.3.1, the proposed solution profiles the workload online, and therefore requires no prior test runs to steer placement.

An alternative, or complement, to access tracking is to leverage PTE information. For instance, the per-page reference and dirty bits, which already exist in the PTEs of commodity systems, allow solutions to have a binary classification on whether or not a page was recently accessed and/or modified. The main benefit to this approach is that the bits are managed by the MMU, and therefore enable pages to be classified at a lower overhead than access tracking.

Solutions that also, or exclusively leverage PTE information extend the traditional CLOCK algorithm to consider a multi-tiered system [25, 26, 29, 30, 32, 37].

In order to have further information about a page's access pattern, the algorithms proposed in [25, 26, 29, 30, 32] propose new per-page bits to be added. CLOCK-DWF [25] suggests a write access counter, incremented when it finds a dirty page, and a counter which tracks the number of times a clean page was kept in the faster tier. M-CLOCK [26] proposes a *lazy* bit, which prevents pages from being promoted to a space-depleted tier when a write access is intercepted. AIMR [29] introduces the *suggest* bit, set when a faster tier-resident dirty page is modified. CLOCK-HM [30], proposed by the same authors as AIMR, re-implements the *suggest* bit, alongside with two other bits which characterize pages based on past CLOCK iterations, which enables the algorithm to detect and store more information about the pages' access pattern and guide placement accordingly. AC-CLOCK [32] implements state-based classification, which separates pages into four different intensity levels, ranging from very cold to very

hot, based on the number of read/write accesses between CLOCK iterations, therefore also requiring additional PTE bits to be added and monitored.

We can speculate that solutions which are designed to intercept every or the majority of accesses [20, 27–29, 32, 34, 35] would require hardware changes in order to be competitive with DualStack, or with solutions that do not rely on access tracking. However, as these solutions are tested in a trace- or simulation-driven environment, the details on how access tracking would be implemented in a real system are not specified. CLOCK-DWF [25] and M-CLOCK [26] only apply access tracking for page promotion, where M-CLOCK intercepts at most two write accesses to a page, and CLOCK-DWF defines an access threshold which causes pages above the threshold to be promoted. Therefore, we consider that these algorithms could be implemented without hardware changes with a lesser penalty to overhead, without much loss to precision.

Intercepting TLB misses comes as a faster alternative, as the mechanism does not impact the normal behavior of the TLB. However, since TLB hits are not measured, the mechanism only tracks some of a workload's accesses, with its precision being dependent on how ineffective the TLB is for a certain workload. Although BadgerTrap could potentially track every TLB miss, doing so has a large overhead, slowing down applications by up to 40x [84].

Therefore, solutions that apply BadgerTrap only use it to monitor subset of a workload's pages. Memos [37] instruments TLB misses for the slower tier-resident pages, in order to separate them into hot and cold regions, and then leverages the PTE's access bits, with a CLOCK-based algorithm, to further distinguish the hot regions into read- and write-dominated. Thermostat [36] proposes to choose sample pages, at equidistant intervals, and intercept TLB misses to them. The sampled pages' TLB miss information is then used to classify their own and their neighbor's intensiveness.

The main benefits of applying an LLC Miss sampling approach vs. the aforementioned access R/W tracking method are: (i) the possibility of achieving a solution without kernel or hardware modifications, since it relies on existing hardware counters which information is available at user-level; (ii) having a lower classification overhead, due to its sampling-based nature. However, the latter comes with the downside of reduced precision, where if the sampling interval is set too low, the collected LLC miss information might misrepresent the workload's access pattern. When compared to TLB miss tracking, the LLC-based approach has the benefit of being capable to additionally observe accesses that caused a TLB hit, and therefore capture accesses to pages that frequently cause TLB hits.

Thrashing

Dynamic solutions may suffer from a problem known as thrashing. Thrashing results from cyclic migrations when the faster memory tier is full and the solution tries to promote a page to it, causing a page to be demoted.

CLOCK algorithms are usually more prone to thrashing, due to their binary page selection mechanism. As CLOCK iterates over a circular list until it finds a suitable page to be replaced, it may end up selecting a frequently accessed page on the second round, after clearing its bits on the first. However, the demoted page will likely fulfill the promotion criteria in the next iteration, and thus trigger another

promotion-demotion cycle to occur, and so on, leading to an over-utilization of CPU resources for migration.

In order to mitigate thrashing, CLOCK-based solutions may decide to migrate data lazily [26], or require a less intensive page to be found in the faster tier, in solutions that implement access frequency counters or other non-binary classification technique [25, 30, 32, 37]. Lazy migration consists of intentionally retaining a page that meets the promotion criteria for one or more iterations, before deciding to promote it to a space-depleted tier, which would cause CLOCK to choose a page to demote.

In contrast, LRU-based implementations inherently mitigate thrashing, as they are always implemented with a non-binary classification scheme. Pages at the top of the LRU queue are, on principle, more frequently accessed than pages at the bottom. Therefore, when a page must be replaced, the algorithm chooses the page at the bottom of the queue, which is less likely to be accessed in the near future.

Bandwidth-Aware Page Distribution

While the solutions described thus far try to maximize the fast memory utilization by placing the most frequently accessed data in it until its capacity is depleted or near to depletion, we identify two solutions which consider bandwidth as a metric to guide placement.

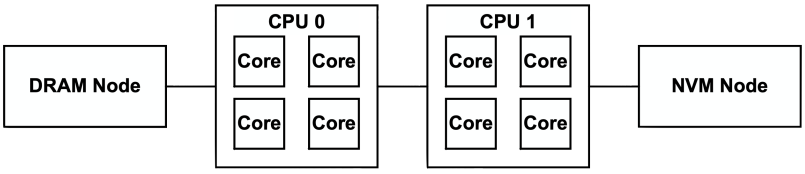


Figure 2.4: Single tier heterogeneous memory architecture [38].

Yu et al. [38] propose a solution that initially allocates data proportionally to each tier's bandwidth, tailored to a system with similar properties to an HMA. The authors propose that each socket is populated with a single memory tier, and therefore each tier is local to a single CPU, and seen as a NUMA node. Although the solution is designed for a DRAM-NVM system, it can be extended to consider any other memory pair, e.g., MCDRAM-DRAM. We illustrate the proposed architecture in Figure 2.4.

For example, if the slower tier is estimated to be 5x slower, Yu et al. propose that for every 5 pages allocated in the faster tier, one should be placed in the slower one. A locality-based migration mechanism is also proposed, which dynamically migrates pages to a different tier if the migration would result in more accesses being fulfilled locally. However, in order to preserve the bandwidth-proportional placement ratio, the method relies on keeping the number of migrations in each direction identical.

The main drawback of the solution is that it assumes a non-standard memory configuration, where the CPU associated to the slower memory will have a lower memory bandwidth available as it will either: (i) perform a local access, incurring the latency and throughput penalty of slower memory; or (ii) access fast memory via the limited-bandwidth interlink that connects both NUMA nodes.

Memos [37] instead assumes the standard multi-tiered HMA configuration. The solution starts by placing every page in the slower tier and then promotes the most intensive pages, as long as the migration results in a combined throughput gain. The number of pages to migrate is dynamic, and depends on the effects of the previous migration on the tiers' throughput. If, after a migration, the observed faster tier throughput gain was less than in the previous cycle, Memos lowers the pages to migrate in the next epoch. Moreover, if the faster tier's throughput increased less than what the slower tier's decreased, the solution instead demotes intensive pages, therefore restoring the previous epoch's distribution. The solution eventually reaches a point where it stops migrating pages, assuming that it has reached an optimal bandwidth-proportional page distribution.

Unlike the solution proposed by Yu et al., which defines a static BW-aware page distribution, Memos targets the same goal observing how throughput reacts to a migration.

We will further discuss these solutions in the next section, hypothesizing how they would apply to a DRAM-NVM HMA.

2.3.3 NUMA-Aware Solutions Compatibility

Besides the solution proposed by Yu et al., which by design always considers a multi-socket architecture, Intel's AutoNUMA patch [20] is also attuned to work with multiple sockets, each with a multi-tiered memory configuration. The solution extends the locality-based inter-socket approach of the base AutoNUMA with mechanisms that adapt to systems equipped with DRAM-DCPMM HMAs.

We see an adaptability of similar contention-aware algorithms to this architecture. Namely, if we define the NVM node as a memory extension to its local CPU and DRAM node, then contention-aware algorithms could consider inter-socket communication with the base co-location, interleaving and other contention-management mechanisms, extending the HMA data placement approach within the socket with an inter-socket contention-aware solution.

2.3.4 Placement in DRAM-NVM Architectures

When considering HMAs that contain an NVM tier, read/write asymmetry is also a factor that should be taken into consideration.

For example, in DRAM-DCPMM systems, the slower non-volatile tier is larger, but has asymmetric latency and bandwidth values depending on the operation performed. Thus, in order to allow performant placement decisions in these systems, the proposed solution must be able to classify and prioritize frequently modified data in DRAM.

The identified static placement literature fails to consider asymmetry, although most assume NVM as a possible configured tier. In contrast, the majority of the identified dynamic placement works not only consider a DRAM-NVM architecture, but provide a design that is specifically designed for it, almost always considering the system to be asymmetric.

DCPMM was recently introduced into the market as the first widely available NVM technology that enables scaling the memory of existing systems past the capacity imposed by DRAM-only configura-

tions. However, as the majority of the dynamic literature precedes the commercial availability of NVM, the proposed solutions are tested recurring to simulation- or emulation-based techniques, and often assume inaccurate latency and throughput values. Furthermore, these values are based on theoretical evaluations of PCM [6–10], which was regarded as the most likely candidate to overthrow DRAM-only architectures before DCPMM was introduced.

This section will discuss dynamic placement techniques for DRAM-NVM HMAs. In the next chapter, we will also conclude our overview of past literature with a discussion on how these solutions would perform if applied to a DRAM-DCPMM system.

Asymmetry-aware Policies

Another approach to the aforementioned initial placement policies in DRAM-NVM HMAs is to allocate data to DRAM on write operations but directly to NVM on read operations [25]. This allows write-dominated pages to fit in DRAM more often, at the cost of placing read-intensive pages in the limited latency/bandwidth NVM tier. In workloads that do not have a working set of write-intensive pages which fully saturate or occupy DRAM, this leads to a suboptimal page distribution, that could be improved by placing more read-dominated pages in the faster tier.

Instead, the more common approach is to always prefer DRAM allocation, and design a migration mechanism that reacts to its lack of space, by demoting cold and read-dominated pages to the slower tier, when a write-intensive candidate is found to promote. In order to achieve this, most algorithms propose to extend CLOCK and LRU-based algorithms in order to further distinguish pages not only based on their access frequency, but also access nature, between read- and write-dominated.

CLOCK Variants

Lee et al. propose M-CLOCK [26], a page-level approach thought out for a DRAM-NVM architecture, which reduces the number of unnecessary migrations between NVM and DRAM, by introducing a lazy migration scheme. The authors divide the algorithm into two components: *Classification of Write-intensive pages in DRAM*, and *Lazy Migration*:

- The first component operates in the DRAM and consists of 2 pointers, iterating over a circular list of pages. The pointers are similar to the one used in CLOCK, with the added functionality of also considering each page's dirty bit, which adds information on whether a page was written after the last explicit cache flush. D-hand, one of the pointers, manages pages that were referenced and written recently. These pages are ideally kept in DRAM as long as their write-intensive nature is maintained. C-hand, the other pointer, considers all other pages, called candidate pages. When the DRAM is full, the algorithm is initiated, and a candidate page is chosen for migration to the NVM, or simply reclaimed by the MMU. A candidate page does not migrate to the NVM, and is instead evicted, when it has both the reference and dirty bits unset, which indicates it has not been accessed in a long time, and therefore its presence in memory is no longer useful.

- The second component operates in the NVM and decides, after a write operation, whether or not the page written should migrate to DRAM. A new per-page bit is introduced in this component, called the lazy bit. This bit is used to force a migration when the DRAM is full, and is set when the algorithm chooses not to migrate a written page. The main purpose of the lazy bit is to prevent thrashing.

With this approach, M-CLOCK is able to effectively reduce the number of NVM writes.

A similar adaptation of CLOCK, CLOCK-DWF [25], surfaced in the same time frame of M-CLOCK. The algorithm tracks the total number of writes for each NVM-resident page in a software-level structure. The structure is used to select pages above a write threshold to promote to DRAM.

Demotion in CLOCK-DWF tracks a per-page write counter, for pages in DRAM, which is incremented when the algorithm iterates over a dirtied page. The authors define a write count threshold, above which pages are considered to be intensive. Another counter (*overlooked*), tracks the number of times a page was retained in its tier, and has a similar threshold, called *expiration*.

When the demotion algorithm iterates over a page, it checks its dirty bit. If the bit is set, it increments the write counter, resets the dirty bit and *overlooked* counter, and retains the page in memory. Otherwise, it checks if the page's current write counter is above the write threshold, and additionally if its *overlooked* counter is below the *expiration* threshold. If so, it increments the *overlooked* counter and decides to retain the page. Otherwise, it chooses the page for demotion.

AC-CLOCK [32] tracks accesses to all pages. It presents a state-based implementation, which is used to distinguish DRAM pages into four intensiveness levels (very cold (0) – very hot (3)). When it iterates over these pages, it checks the number of read and write accesses that each page had since the last iteration, and increases or lowers their state accordingly. When a page in NVM is modified, it decides to promote it, and tries to find a DRAM page with a state of 0, retaining the page if no candidate was found. If the same page is accessed a second time, the algorithm relaxes the goal state, and tries to find a replacement in DRAM with a state of 2 or below.

AIMR [29] and CLOCK-HM [30], both devised by the same authors, combine CLOCK with an LRU of recently evicted pages. If a page is evicted from main memory and then accessed again, the algorithms check if the page is still in the LRU and was once write-intensive. If so, the page is always placed in DRAM, forcing a demotion if needed. Otherwise, the algorithms instead try to find a suitable DRAM page to be replaced, but may place the page in NVM if no candidate is found.

Memos [37] initially places every page in NVM and intercepts TLB misses to them, in order to identify hot and cold regions. The authors propose a CLOCK mechanism for page promotion, which runs over the hot regions' pages, separating them into read- and write-dominated. The algorithm then promotes pages, prioritizing the latter. When DRAM is full, a CLOCK algorithm is also used, which simply checks the page's reference bit, in order to identify cold pages. Memos improves the classification precision of prior approaches by not only applying TLB miss interception in order to accurately portray an application's address space, but also by running multiple CLOCK iterations, which reduce the chance of a page being promoted due to being modified shortly before the CLOCK iteration, without relying on access tracking, similarly to the aforementioned approaches.

LRU Variants

Seok et al. developed a page-level algorithm [27, 28] based on LRU. The objective is similar to the previous solutions: maximizing NVM endurance, which is achieved through minimizing the number of writes to NVM. The solution integrates a module that predicts the read/write ratio for each page. The module keeps track of each page's weight (W_{cur}) at software-level. The weight is updated at every page request based on the previous weight (W_{prev}) and the type of the request (RT). The request type variable takes the value 1 if the operation is a write and -1 if it is a read. The weight is updated as follows:

$$W_{cur} = \alpha * W_{prev} + (1 - \alpha) * RT : \alpha \in [0, 1]$$

The constant α can be changed in order to give more or less relevance to the previous weight, and enables the module to adapt to workloads with different characteristics. The algorithm then chooses where to place pages based on the page's weight, setting a weight threshold, where pages above the threshold ($W_{cur} > threshold$) are considered more likely to be written in the near future and therefore fit for volatile memory and vice-versa.

DualStack [31] proposes two MMU-managed LRU lists, for each tier, using the reuse distance concept from LIRS [85], which is an LRU-based cache replacement algorithm design for the traditional architecture. Similarly to LIRS, the interval between accesses to a page is tracked, giving two possible classifications to each page: Low Inter-reference Recency (LIR) or High Inter-reference Recency (HIR), kept in one of the LRU lists for each tier. Moreover, another LRU list tracks the number of read and write requests to a page.

The algorithm sets a read and a write threshold. For pages in DRAM, it chooses to demote LIR pages that exceed the read threshold, as they are considered infrequently accessed (LIR) and read-dominated. Each page's read counter is reset whenever they are written to, therefore retaining them in DRAM for longer.

Similarly, the algorithm chooses to promote NVM-resident HIR pages upon exceeding a write threshold. Unlike the read counter for pages in DRAM, the write counter is not reset on a read operation. Instead, if the NVM-resident page is above its read threshold, further read accesses decrement the write counter, therefore choosing to retain the page in NVM for longer.

A patch for Linux's AutoNUMA [20] is currently being developed in parallel with this work, and is tested with real DCPMM. The algorithm is built on top of the thread and page migration techniques used by the base version of AutoNUMA, extending it to also enable page migration between local DRAM and DCPMM memories.

Promotion from DCPMM to DRAM is achieved by leveraging the existing page fault mechanisms in AutoNUMA, to determine a page's access frequency. The patch adapts the page tables' scanning routine to clear the pages' present bit, therefore causing a page fault on the subsequent access, similarly to the BadgerTrap [84] approach. When accessed, the algorithm calculates the time it took between the clearing the present bit and the page fault. If the time is below a defined threshold, it considers the page to be intensive, and therefore promotes it to DRAM. The threshold depends on the type of operation that

caused the page fault. If the page fault was caused by a write operation, then the threshold is multiplied by a set number (defaults to 2), making it easier for write-intensive pages to be promoted to DRAM.

Demotion from DRAM to NVM is achieved by modifying Linux's swapping routine, which is changed to demote a page to DCPMM instead of directly swapping or reclaiming it. Demotion runs under the basic Linux's swapper thread, which triggers when free memory falls below a threshold.

TwoLRU [35] presents two different LRU lists, one for each tier, and always places new pages in DRAM. The NVM LRU list tracks per-page read and write accesses, and the respective read and write percentage. Pages with a higher write access count and percentage are ordered closer to the top of the LRU, and are promoted when they exceed a threshold. Similarly, read accesses may also trigger a promotion, but have an higher threshold, and therefore are less likely to promote than write-intensive pages. The DRAM LRU promotes pages to the top of the LRU when accessed, either for a read or write operation, and demotes pages at the bottom of the LRU list when a new page is allocated, or an NVM-resident page is chosen to promote. This demotion mechanism is similar to the one proposed in Memos, as it does not prioritize read over write-dominated pages.

Summary

Article	Design Assumptions	Implementation	Evaluation	Modifications	
	HMA tiers	Classification Algorithm	Real System	HW	OS
CLOCK-DWF [25]	DRAM-PCM	CLOCK	✗	✗	✓
M-CLOCK [26]	DRAM-PCM	CLOCK	✗	✗	✓
AC-CLOCK [32]	DRAM-PCM	CLOCK	✗	✓	✓
AIMR [29]	DRAM-NVM	CLOCK+LRU	✗	✓	✓
CLOCK-HM [30]	DRAM-PCM	CLOCK+LRU	✗	✓	✓
Seok et al. [27, 28]	DRAM-PCM	LRU	✗	✓	✓
DualStack [31]	DRAM-PCM	LRU	✗	✓	✓
HeteroOS [33]	MCDRAM-DRAM-NVM	LRU	✗	✓	✓
UIMigrate [34]	DRAM-PCM	LRU	✗	✓	✓
TwoLRU [35]	DRAM-PCM	LRU	✗	✓	✓
AutoNUMA patch [20]	DRAM-DCPMM	LRU	✓	✗	✓
Thermostat [36]	DRAM-DCPMM	TLB Misses	✗	✗	✓
Memos [37]	DRAM-NVM	TLB Misses+CLOCK	✗	✗	✓
Yu et al. [38]	DRAM-PCM	N/A (Locality-based)	✗	✗	✗
Salkhordeh et al. [39]	DRAM-PCM	PEBS	✗	✗	✗

Table 2.1: Comparison of NVM-aware related work.

Table 2.1 presents an overview of all identified dynamic placement literature that considers an NVM-equipped HMA.

From the identified solutions, the vast majority consider PCM as the secondary tier, with some not specifying the NVM tier technology (represented simply with NVM). Due to the lack of commercially available NVM at the time, these solutions' evaluation is conducted in a simulation- or trace-driven environment, either emulating a full DRAM-NVM system, or by introducing an artificial access delay to a region of DRAM's address space, considering it to present similar performance to a real NVM tier, as proposed in Thermostat [36]. Even though Thermostat intends to simulate a DRAM-DCPMM sys-

tem, it fails to consider read/write asymmetry, setting a uniform access delay. In contrast, the currently in-development AutoNUMA patch [20] is not only tailored to DCPMM, but has also presented some experiments conducted on a real DCPMM-equipped system.

Many of the solutions discussed require significant HW and/or OS changes to be made to the system. We consider that solutions which require tracking every or the majority of accesses [27–35] require hardware modifications, even though the majority leaves the implementation details unspecified. An alternative is to apply PEBS-based LLC miss sampling, as proposed in Salkhordeh et al. [39], which comes with an overhead vs. precision trade-off, where lower sampling rates provide more accurate results at the cost of more CPU/memory resources. Therefore, we consider that these solutions are not viable candidates for being implemented in our DRAM-DCPMM system with current hardware. From the remaining solutions [20, 25, 26, 36–39], only the BW-aware approach by Yu et al. and Salkhordeh's PEBS-based algorithm do not require significant changes to the kernel.

Chapter 3

Tailoring Placement to a DRAM-DCPMM Architecture

In this chapter, we first describe how DCPMM is organized internally and configured within a system, and then characterize its performance with different workload types in different physical and logical configurations. We also present benchmarks that extensively evaluate a selected configuration. We use these benchmarks in order to outline the optimal data placement strategy in DCPMM-equipped HMAs through a set of guidelines.

We then compare related work based presented in Section 2.3.4 based on the fulfillment of these guidelines, in a discussion section. The discussion section will hypothesize how the presented solutions, proposed for older NVM technologies and/or with a inherently inaccurate simulation- or trace-driven evaluation would perform when applied to a real system running DCPMM, such as ours.

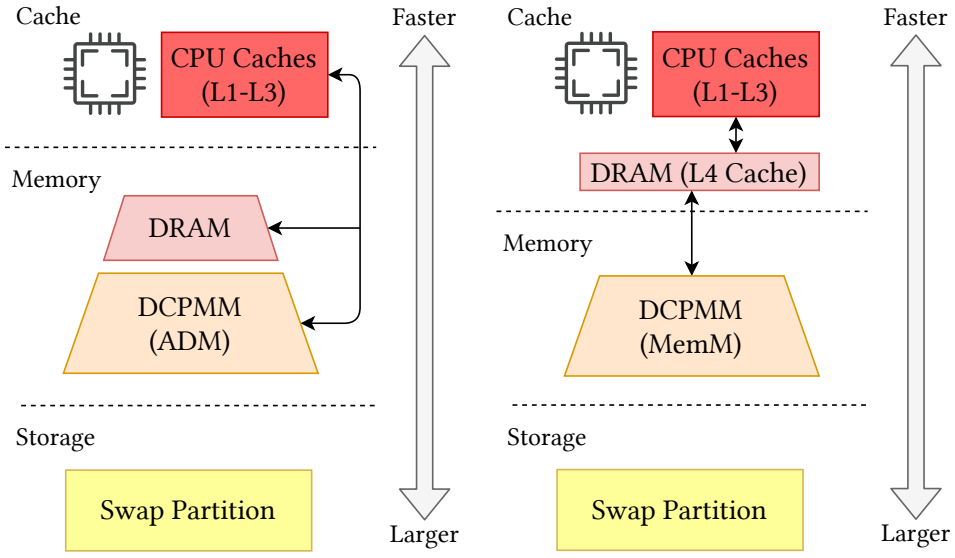
3.1 DCPMM Internals and Configuration

DCPMM is delivered as DIMMs that are compatible with DDR4 sockets. The current capacity of DCPMM modules range from 128GB to 512GB, which represents up to a 4x increase in per-module capacity compared to DDR4 DRAM. Currently, DCPMM modules can be used with Intel's Cascade Lake CPUs with large memory support, either in single-socket or multi-socket machines.

In this setup, each CPU contains 2 integrated memory controllers (iMC), each supporting up to 3 memory channels. Each iMC uses the DDR-T protocol to communicate with DCPMM. Like DDR4, DDR-T operates at cache-line granularity (usually 64B). Internally, each DCPMM module caches 256B blocks (called XPLines), with an associated prefetcher. This cache also serves as a write-combining buffer for adjacent stores. Due to the granularity mismatch between DDR4 and XPLines, random stores incur in costly read-modify-write cycles. Similarly to SSDs, DCPMM uses logical addressing for minimizing wear-leveling, leveraging an internal address indirection table.

Current systems with DCPMM are HMAs, where different DIMM configurations are possible, with varying DRAM-DCPMM capacity ratios. However, these systems have several restrictions. Firstly, each

iMC needs to be populated with at least one DRAM module. Secondly, each memory channel can be either served by DRAM only (up to 2 modules), DCPMM only (1 module) or, in some models, both DCPMM and DRAM (1 module each).



(a) ADM Configuration (b) MemM Configuration

Figure 3.1: ADM and MemM DCPMM architectures within a socket.

DCPMM can be configured in two modes: App Direct mode (ADM), and Memory mode (MemM), both of which are illustrated in Figure 3.1.

In ADM, DCPMM can be seen as a natural extension of the address space, effectively increasing the system's available memory to the capacity of DRAM and DCPMM combined. In this configuration, both DRAM and DCPMM are directly exposed to the OS as two distinct memory nodes. Each one can be directly accessed through load and store operations. The DCPMM node does not have any CPUs associated to it, therefore being considered remote to all running threads. However, the closest CPU's NUMA *distance* is always lower than that of "true" remote DRAM nodes. Thus, Linux views DCPMM within a socket as the closest NUMA node after local DRAM, and prioritizes data allocation in it when configured with the default *node local* policy.

ADM DCPMM can also be configured as a directly-mapped persistent storage medium, which can be used to store files and user data, similarly to traditional storage. This latter configuration is referred to as "Storage over App Direct Mode".

Throughout this work, we factor out the aspect of persistence guarantees, since our main focus is on how to leverage this new layer of the memory hierarchy to improve system performance, and persistence can be seen as an orthogonal issue. Therefore, we will be referring to the NUMA configuration when discussing ADM DCPMM.

In MemM, the socket's DRAM is configured as an internal last-level cache (LLC), which interposes every access to the local DCPMM memory node. In this configuration, processes allocate pages directly to DCPMM, while the hardware manages which pages are cached in DRAM, seamlessly to the programmer. This architecture presents the OS with an hardware-managed data placement algorithm,

where a single NUMA node, in the form of DCPMM, exists within each socket.

DCPMM and DRAM can populate the memory slots in different physical dispositions, observing the limitations imposed at the iMC and memory channel levels. Prior studies evaluate the performance impact of different DIMM configurations, and DCPMM modes [18, 47, 86–88].

Lenovo [87] shows that a 2-2-1 configuration consisting of 1 DRAM DIMM per channel, and an additional DCPMM configured in MemM in the first and second channels outperforms a 2-1-1 (3x DRAM and 1x MemM DCPMM in the first channel) configuration in AAL by as much as 40% in read-only workloads and 50% in "2 reads 1 write" (2R1W) workloads, while only providing an additional persistent DIMM.

Furthermore, studies found that DCPMM suffers an exponential increase in latency when saturated, i.e., loaded latency, while DRAM is able to remain performant under full load [18, 86]. This is relevant for data placement algorithms, as it is shown that DCPMM performs best in near idle conditions.

3.2 Placement Guidelines for DRAM-DCPMM

This provides multiple insights on how pages should be distributed in DRAM-DCPMM architectures. We leverage these conclusions in order to guide Ambix's design and implementation, and hypothesize how the discussed NVM-aware literature would apply to systems equipped with this architecture.

We answer one main question: Given the idiosyncrasies of DCPMM, which page placement strategy provides the best possible throughput and lowest energy consumption?

We devise two benchmarks, which test how different page placement strategies affect throughput and energy consumption, evaluating workloads with: (i) a uniform access pattern and small working sets, and (ii) non-uniform patterns with arbitrarily-large working sets.

3.2.1 Experimental Methodology

We populate a socket with a total of 32GB of DRAM and 256GB of DCPMM, in a 1-1 configuration (i.e., each memory channel contains a single DRAM and DCPMM DIMMs), where only 2 out of the 3 available memory channels are used. The configured CPU is an Intel® Xeon® Gold 5218 CPU, running at 2.30GHz, with 16 physical cores (32 threads).

Due to idle system resources, DRAM utilization is limited to 27GB, or $\sim$0.84x its effective size, in the benchmarks we run.

Both benchmarks generate multi-threaded workloads that allocate a test array, and then iterate over it for a fixed runtime, after which they output the number of accesses performed to the array per second (throughput). The workloads have a sequential access pattern, where accesses have a large enough stride between them, such that each page is referenced by a single access. The test array is also parametrized to always have a large enough number of pages, i.e., size, causing each entry's reuse distance to be much larger than the LLC size. This leads to an accessed entry being evicted from the cache before accessed again in the next iteration, therefore maximizing the percentage of memory

accesses.

Each thread continuously loops over a unique range of entries, i.e., no two threads ever access the same entry. For example, in a N-threads scenario, the nth thread : $n \in \{0..N-1\}$ starts each array iteration on the $n * stride$ entry, and then jumps to the $2 * n * stride$ entry, and so on, until it reaches the end of the array, at which point it starts a new iteration. The stride is dependent on the page size, which is set to 4KB in our system.

Since we intend to study page placement between memory tiers, we isolate the benchmarks to a single NUMA socket. In order to achieve this, we use the *numactl* CLI, where we isolate the benchmark threads to a single CPU node and restrict allocation to the local DRAM and DCPMM memory nodes.

When a test array entry is modified, it is cached (load) and eventually written back (store). Due to the configured stride and large reuse distance, we can assume that every cached entry is evicted before referenced again. Therefore, write-only workloads are denominated 1R1W, and two accesses are counted when an array entry is modified.

Besides measuring throughput, we also leverage *perf*, in order to collect memory energy consumption (excludes processing- and IO-related energy consumption) of the timed portion of the workload, i.e., without the allocation phase, in Joules, with the following command:

```
perf stat -e power/energy-ram/ [workload]
```

Although cold pages, i.e., pages that are initialized but seldom referenced, are common in real workloads, we do not consider these in our benchmarks, where all pages are equally intensive, varying only between read- and write-dominated. We assume that cold pages should never be prioritized in DRAM over intensive pages. This assumption is consistent with the design of current page or cache replacement algorithms for DRAM-only architectures, in which cold pages are evicted first.

3.2.2 Small Working Set Study – Balancing Approach

In order to study the throughput and energy consumption effect of distributing pages over the DRAM and DCPMM tiers, with and without DRAM bandwidth saturation, we devise the Interleave Weighted Benchmark (IWB). IWB tests if and by how much a workload's throughput can be increased by distributing pages between memory tiers when bandwidth saturation is detected.

The benchmark allocates a 24GB array of fixed size, which fully fits in DRAM. It is parameterized with a varying number of threads (1-32), and page distribution, from all pages in DCPMM to all pages in DRAM.

IWB is configured to allocate N% of pages DRAM, where N varies between 0 and 100. It first allocates the full array in DRAM and then separates the array's pages into 100 page groups, migrating the last 100-N pages from each group to DCPMM. This is done with `move_pages()`, which is imported from the *libnuma* library.

We benchmark two workloads: (i) Read-only (RO); and (ii) Write-only (WO).

In Figures 3.2 to 3.5 we present two throughput and two energy heat maps from the RO and WO parametrizations of IWB, respectively.

Pages Placed in DRAM (%)	Memory Access Demand (# Threads)									
	1	2	4	8	12	16	20	24	28	32
0	21,93	32,28	39,84	54,79	55,10	55,11	54,96	55,14	55,17	55,11
5	22,25	33,68	41,96	58,15	58,23	58,13	57,78	58,18	58,19	58,03
10	22,79	35,88	44,54	60,77	61,34	61,32	61,70	61,37	61,38	61,31
15	23,59	36,05	48,14	64,95	65,29	65,07	66,20	65,11	65,12	64,41
20	24,29	37,06	52,05	68,27	69,24	69,10	68,88	69,14	69,14	69,07
25	25,24	39,15	53,63	73,53	74,26	73,96	73,51	73,95	73,96	73,87
30	26,09	39,19	56,59	77,80	79,46	79,17	79,92	79,16	79,19	79,09
35	27,40	42,08	61,05	84,61	85,95	85,67	87,49	85,54	85,51	85,47
40	28,61	43,05	63,24	90,31	93,14	92,73	92,27	92,60	92,64	92,59
45	29,80	45,51	65,93	99,42	101,94	101,87	101,33	101,53	101,44	101,45
50	31,31	46,62	72,72	106,76	112,22	111,94	113,80	111,59	111,58	111,56
55	33,50	48,95	82,02	119,28	124,76	125,31	129,44	124,91	124,81	123,84
60	34,85	50,67	79,37	127,31	140,69	141,20	140,20	140,54	140,43	140,51
65	36,63	52,92	93,00	143,20	160,77	162,37	164,06	162,38	162,30	162,34
70	38,85	56,46	96,05	152,15	187,26	189,84	198,49	190,06	189,99	189,84
75	41,03	59,05	104,77	170,02	220,79	227,02	244,37	229,67	229,89	229,70
80	43,10	61,79	110,34	180,85	246,18	266,35	272,35	273,81	279,39	276,67
85	47,08	64,08	117,16	194,83	241,35	257,77	273,18	270,07	276,95	264,23
90	50,77	67,03	122,44	200,68	226,40	239,33	255,94	256,16	263,92	245,90
95	52,88	71,04	128,36	213,15	214,35	225,57	243,50	241,10	249,22	230,93
100	63,46	81,09	136,55	209,58	204,12	216,70	239,13	236,20	241,67	217,64

Figure 3.2: IWB throughput heat map (M accesses/s). Read-only workload.

Pages Placed in DRAM (%)	Memory Access Demand (# Threads)									
	1	2	4	8	12	16	20	24	28	32
0	15,52	19,63	20,80	20,93	20,87	20,94	20,82	20,88	20,91	20,91
5	16,08	21,46	22,53	22,36	22,27	22,20	21,98	21,94	21,50	21,94
10	16,85	21,53	23,16	23,19	23,21	23,22	23,20	23,23	23,26	23,24
15	17,65	23,82	25,36	25,16	24,82	25,04	25,12	24,76	24,19	24,33
20	18,59	23,85	26,08	26,05	26,11	26,11	26,05	26,09	26,12	26,12
25	19,64	26,79	28,83	28,41	28,43	28,26	28,08	27,95	28,09	28,11
30	20,87	26,69	29,88	29,80	29,79	29,79	29,87	29,80	29,85	29,84
35	22,37	30,81	33,40	33,06	32,70	32,80	32,87	32,62	31,98	32,72
40	23,87	30,90	35,08	34,84	34,89	34,78	34,67	34,79	34,83	34,84
45	25,83	36,02	39,63	39,10	38,97	38,76	37,76	38,54	38,40	38,20
50	28,10	36,98	42,20	41,95	41,98	41,85	41,92	41,77	41,81	41,78
55	30,86	42,73	48,75	48,19	47,78	47,57	48,69	47,42	47,20	47,43
60	34,29	45,73	53,35	52,97	52,67	52,49	52,18	52,39	52,39	52,44
65	38,03	53,55	63,25	62,40	61,92	61,53	60,28	61,56	61,38	59,79
70	41,94	58,82	71,76	71,29	70,90	70,80	71,19	70,60	70,47	70,69
75	47,29	67,49	86,82	88,63	88,03	87,26	90,64	87,40	87,12	86,76
80	52,71	78,35	105,58	108,17	107,88	108,13	109,75	107,66	106,94	107,77
85	58,68	93,95	126,19	149,74	155,35	150,56	154,60	151,08	150,17	149,43
90	67,81	101,20	151,87	183,23	195,19	195,90	211,65	200,67	202,45	206,29
95	73,54	110,06	166,44	195,90	200,38	190,39	201,15	199,94	201,73	203,45
100	83,47	127,33	169,20	183,05	191,09	181,91	193,95	188,97	192,66	196,84

Figure 3.3: IWB throughput heat map (M accesses/s). Write-only workload.

Pages Placed in DRAM (%)	Memory Access Demand (# Threads)									
	1	2	4	8	12	16	20	24	28	32
0	120,66	83,88	69,11	51,37	51,00	51,10	51,60	51,52	51,47	50,94
5	118,83	81,07	66,48	48,55	48,31	48,40	49,38	49,27	48,65	48,90
10	116,26	75,78	61,80	46,61	46,18	46,41	47,01	46,87	46,71	46,33
15	112,18	75,46	58,01	43,58	43,71	43,63	44,05	44,24	43,59	44,14
20	109,37	73,43	52,86	41,78	41,60	41,72	42,25	42,12	42,16	41,57
25	105,42	69,75	52,50	39,22	38,88	39,34	40,36	39,94	39,79	39,26
30	102,16	69,60	48,86	37,39	36,78	37,07	37,08	37,29	37,35	37,15
35	97,62	64,99	46,45	34,49	34,36	34,26	34,43	34,87	34,84	34,53
40	93,36	63,45	43,94	32,40	31,87	32,56	32,62	32,88	32,63	32,58
45	90,01	60,14	43,55	30,04	29,76	30,15	30,52	30,56	30,21	30,14
50	85,49	58,67	38,94	28,11	26,99	27,21	27,10	27,60	27,55	27,51
55	80,28	56,02	35,27	25,44	24,92	24,91	24,50	25,39	25,17	25,18
60	77,28	54,25	36,21	23,97	22,45	22,64	22,66	23,12	22,80	22,88
65	73,68	52,07	31,39	21,80	20,17	20,16	20,11	20,71	20,32	20,46
70	69,40	48,99	30,59	20,58	17,72	17,64	17,17	17,89	17,78	17,88
75	65,92	47,07	28,21	18,65	15,17	15,08	14,34	15,06	15,13	15,02
80	62,74	45,29	26,75	17,69	13,66	12,92	12,80	12,76	12,60	12,59
85	57,57	43,72	25,20	16,35	13,56	12,78	12,56	12,61	12,39	12,54
90	53,61	41,86	24,13	15,77	14,16	13,54	13,14	13,22	12,80	13,25
95	51,51	39,60	23,04	14,74	14,70	14,12	13,50	13,78	13,32	13,86
100	43,65	34,79	21,85	14,84	15,25	14,46	13,56	13,85	13,66	14,47

Figure 3.4: IWB energy heat map (J/M accesses). Read-only workload.

Pages Placed in DRAM (%)	Memory Access Demand (# Threads)									
	1	2	4	8	12	16	20	24	28	32
0	170,34	136,11	128,66	128,05	129,23	129,18	129,27	129,41	128,87	128,77
5	164,54	124,02	118,94	119,69	121,26	120,92	122,33	123,11	125,06	122,60
10	157,12	124,11	115,98	116,44	116,59	116,83	116,45	116,79	116,27	116,49
15	149,49	111,56	106,22	107,14	109,46	108,36	107,77	109,67	111,94	111,22
20	142,97	112,48	103,76	104,21	104,36	104,49	104,41	104,59	104,22	104,03
25	135,26	100,78	93,95	95,92	96,05	96,05	96,73	97,92	97,08	96,66
30	127,91	101,20	91,20	91,57	92,06	91,95	91,56	92,00	91,62	91,32
35	119,71	88,08	81,80	82,77	84,07	83,33	83,36	83,95	85,60	83,68
40	112,54	87,78	77,99	78,65	78,99	79,15	79,21	79,16	78,85	78,92
45	104,49	75,72	69,21	70,49	70,87	70,83	72,98	71,60	71,93	71,96
50	95,74	73,71	65,48	66,09	66,56	66,59	66,27	66,58	66,31	65,81
55	87,81	64,43	57,51	57,86	58,59	58,57	57,57	59,05	59,36	58,67
60	79,09	60,44	52,94	53,55	54,35	53,89	54,27	54,35	54,10	53,50
65	71,63	52,37	45,33	46,00	46,16	46,24	47,47	46,46	46,63	47,54
70	65,12	48,00	40,39	40,59	41,10	40,63	40,92	41,09	41,13	40,60
75	58,15	42,20	34,05	33,20	33,60	33,76	32,82	33,82	34,00	33,85
80	52,54	36,77	28,30	27,83	28,44	27,74	27,83	28,36	28,42	27,78
85	47,57	31,25	24,18	20,97	20,60	20,92	20,84	21,10	21,21	20,98
90	41,51	29,00	20,37	17,30	16,80	16,71	15,90	16,44	16,42	15,74
95	38,33	26,80	18,59	16,21	16,24	17,07	16,20	16,31	16,25	15,73
100	33,90	23,54	18,07	17,60	16,88	17,60	16,62	17,01	16,76	16,05

Figure 3.5: IWB energy heat map (J/M accesses). Write-only workload.

In the throughput heat maps, each cell displays the average throughput, in million accesses per second, of a given page distribution and memory access demand, represented by the percentage of pages allocated in DRAM and number of running threads, respectively. The cells are colored in a red to green gradient, where the green cells have the highest possible throughput in a given thread configuration (column-wise). In the energy heat maps, the cells instead contain the average per-access energy consumption of each distribution-demand pair, represented in Joules per million accesses, where green cells contain the lowest possible values, which correlate to the highest energy efficiency.

In all heat maps, we highlight the best page distribution for a given access demand with thicker borders, i.e., the greenest cells.

By design, IWB has a sequential access pattern, where all pages have an identical access frequency. Therefore, the page distribution also represents the distribution of accesses for any given parametrization.

In the read-only workload throughput heat map (Figure 3.2), we observe both DRAM and DCPMM saturation at around 8 threads. Before this point, allocating all pages in DRAM grants the best possible throughput and energy efficiency. At 8 threads, the optimal throughput-wise distribution shifts towards 95%, and subsequently stabilizes around the 80% range when we further increase memory demand. In this workload, the 32 thread configuration observes a 27% increase in throughput when 20% of pages are allocated in DCPMM (80%), compared to all pages in DRAM (100%). As observed in the workload's energy heat map (Figure 3.4), the optimal energy-wise distribution stabilizes at around the 85% range, with a 13% lower per-access energy consumption in the 32 thread configuration.

In the write-only workload throughput heat map (Figure 3.3), the same saturation point is observed for DRAM, but DCPMM is saturated earlier, at 4 threads. In this case, we see the optimal throughput-wise distribution stabilize at 90% after DRAM is saturated. However, the observed throughput gain in the 32 thread configuration is much smaller, with only a 5% increase in the 90% vs. 100% distributions. We attribute this smaller throughput difference to DCPMM's lower write throughput. Similarly to what was observed in the read-only workload, we see that the best energy-wise distribution stabilizes closer

to 100%, in the 95% range. In this scenario, the energy improvement compared to all pages in DRAM is smaller, at around 2%.

By comparing throughput in the 0% and 100% distributions after each tier is saturated, we can also observe DCPMM's read/write asymmetry, compared to DRAM. In the all pages in DCPMM scenarios (0%), at 8-32 threads, throughput is 62% lower on average in the write-intensive workload compared to the read-only one. In comparison, the all pages in DRAM (100%) throughput drops by only 15%. We find that these results are expected, as they confirm prior studies on DCPMM's performance [18, 89].

3.2.3 Large Working Set Study – Placement Policy

The previous study assumed data sets with uniform access patterns, testing workloads with either all read- or write-dominated pages. However, workloads frequently allocate pages which differ in read- and write-dominance. In this scenario, if the working set is unable to fully fit in DRAM, some pages must be placed in DCPMM.

In order to study which pages benefit the most out of being allocated in DRAM, we devise a policy benchmark (PB) which tests different memory policies at varying array sizes, from workloads that fully fit in DRAM, to workloads that are more than twice as large. For simplicity of presentation, PB does not consider the optimal BW-aware ratios defined in IWB, and instead focuses on memory capacity as a trigger for page placement.

PB allocates an array with two equally sized and intensive read- and write-dominated portions. We assume the write-heavy portion to be 1R1W, and the read-intensive one to be 1R. Therefore, PB generates 2R1W workloads.

The benchmark is parameterized to allocate the array following one out of four defined memory policies:

- Write first-touch (WrFT): Allocates as many pages as possible in DRAM, initializing the write-intensive segment first. If any remaining pages exist, those are allocated in DCPMM.
- Read first-touch (RdFT): Same as above, but prioritizes read-intensive pages in DRAM.
- Interleave: Follows the default interleave policy, which performs round-robin allocations between both DRAM and DCPMM nodes, leading to an even page distribution.
- DCPMM: Limits allocation to the DCPMM tier.

We parameterize PB with different array sizes, ranging from 0.5x-2.5x DRAM size, for each policy. A PB run launches 32 threads, which fully utilize the local CPU's available cores.

In Figure 3.6 we see multiple parameterizations of PB, with the four aforementioned memory policies, at different workload sizes. The bars represent throughput, in million page accesses per second, and are represented by the left axis. The lines present the per-access energy consumption, in Joules per million accesses, and are associated to the right axis. The horizontal axis indicates the workload size in comparison to DRAM size (recall that DRAM has around 84% effective capacity due to idle system resources).

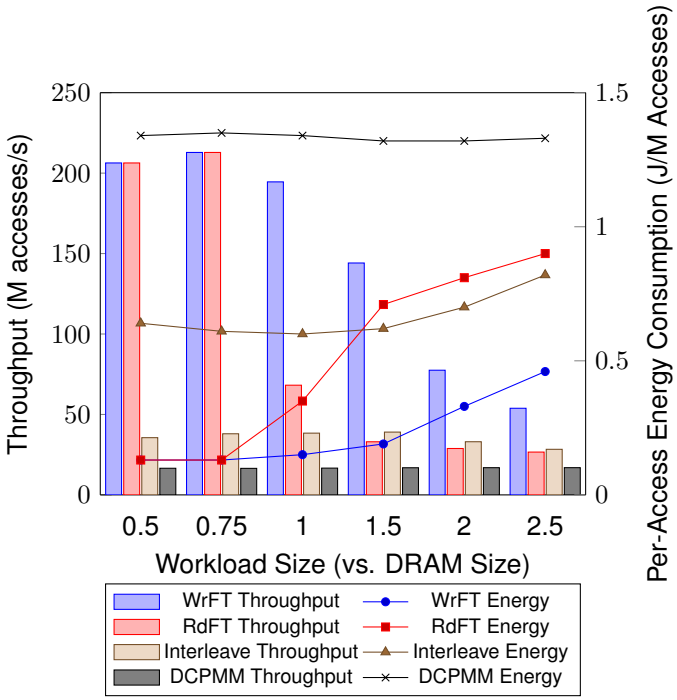


Figure 3.6: PB plot. 2R1W, 32 threads

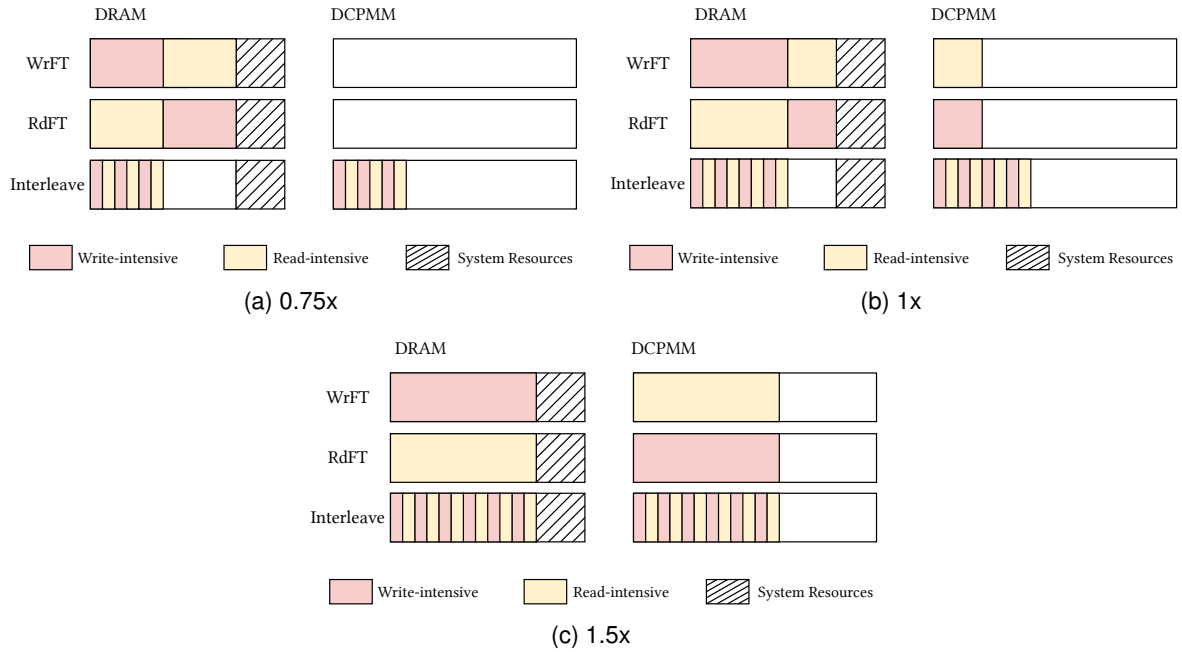


Figure 3.7: Policy page allocation at 0.75x, 1x, and 1.5x workload size. DRAM-DCPMM relative capacity not to scale.

The interleave and DCPMM policies serve as a baseline for the WrFT and RdFT policies, where the interleave policy places both read and write intensive portions evenly, and the DCPMM policy defines the worst throughput and energy efficiency values at each workload size.

As expected, the DCPMM policy throughput and per-access energy consumption results are constant in all workload sizes. The same is true for the interleave policy up to the 1.5x DRAM (1.5x) workload size. However, in the 2x and 2.5x scenarios, the interleave policy depletes DRAM space and therefore allocates a larger percentage of pages in DCPMM, which negatively impacts both throughput and energy efficiency.

In Figure 3.7 we describe how the WrFT, RdFT and interleave policies allocate pages at different workload sizes (0.75x, 1x, and 1.5x). For simplicity of presentation, we simplify DRAM capacity to 24GB (0.75x its capacity), and do not represent the tiers' capacity to scale.

In the 0.5x, and 0.75x workloads, the full test array fits in DRAM. Since the WrFT and RdFT policies prioritize DRAM allocation, they place the full array in DRAM (Figure 3.7a), and thus output the same throughput and energy efficiency results. At 1x (Figure 3.7b), the test array is larger than the available DRAM capacity, and we see a 68% throughput drop and a 177% increased energy consumption per access in the RdFT policy. In contrast, the WrFT policy remains performant until the 2x size workload, at which point the policy places as many pages from the write-intensive portion as the RdFT policy in the 1x workload. Both cases exhibit identical throughput and energy consumption even though the workload is twice as big in the WrFT case.

The 1.5x workload (Figure 3.7c) showcases the scenario where the RdFT and WrFT policies fully allocate each array portion to opposite tiers. In this parameterization, the WrFT policy's throughput is around 5x higher than the RdFT one, which indicates a 5x benefit in prioritizing a write-dominated over a read-dominated page in DRAM.

Overall, prioritizing write-intensive page allocation in DRAM leads to the best throughput and energy efficiency values at every workload size. If the workload is twice as large as DRAM's effective capacity, the interleave policy comes as a second best out of the four. Otherwise, any other policy which prioritizes DRAM allocation over an even page distribution, such as RdFT, grants higher performance.

3.2.4 Insights

From both IWB and PB we can draw two main guidelines which we will leverage to steer page placement in DRAM-DCPMM systems:

- **Prioritize DRAM Allocation**: Before DRAM capacity is full, every accessed page should be placed in DRAM.

- **Asymmetry-aware Migration**: When DRAM is at capacity, cold pages should be prioritized and, if still needed, read-intensive pages. Similarly, if write-intensive pages are detected in DCPMM, these pages should be promoted to DRAM, and exchanged with cold or read-intensive pages, if needed.

IWB additionally shows that, when DRAM bandwidth is saturated, the optimal throughput-wise access distribution lies somewhere between 80 and 90%, with the energy-wise distributions being in the 85 to 95% range. We find that these results prove that a BW-aware component is superfluous in a dynamic HMA-aware placement algorithm due to two main reasons.

Firstly, the main use case for a DRAM-DCPMM HMA is to allow the system to run larger footprint workloads without resorting to remote memory or data eviction. Even in a very read-intensive workload, where the ideal throughput-wise distribution would be around 80% in favor of DRAM, the working set would have to be lower than 1.25x DRAM size in order for this distribution to be possible. For example, if the workload was larger, all additional pages would need to be allocated in DCPMM, shifting the ratio towards the slower tier. Conversely, at smaller or identical workload sizes, a DRAM-only configuration would be able to fit the full workload in two DRAM DIMMs, while still having at least 0.75x free space in the second DIMM, or 37.5% in both, assuming an unweighted interleave distribution.

Secondly, while IWB assumes a sequential-like access pattern where pages are equally intensive, common workloads have the added complexity of accessing some pages more frequently than others, with some being more read- or write-dominated. Moreover, it is also common for a workload's access pattern to change, leading to some pages which were (i) intensive, or (a) write-dominated; becoming (ii) colder or (b) read-dominated, and vice versa. However, as dynamic solutions do not profile the workload a priori, the access pattern is not well known, and therefore the appropriate balance should be found via adaptive methods. Such a balancing mechanism would likely be implemented on a trial and error approach, such as in Memos [37] (see Section 2.3.2), which requires both: (i) the access pattern to remain stable after a migration, in order to correctly associate the resultant throughput difference to the migration; and (ii) the future access pattern of the workload to compensate the migration cost.

Combining both motives, we find that the complexity added by implementing a BW-aware approach would seldom be beneficial in systems which integrate DCPMM. Therefore, we will design Ambix without a BW-aware component.

3.3 Discussion of Related Proposals

Table 3.1 extends the related work summary table in the last chapter (Table 2.1) with the fulfillment of the guidelines set in Section 3.2. The information on the table allow us to hypothesize how the identified solutions would perform on a real system equipped with a DRAM-DCPMM HMA.

The majority of solutions consider NVM asymmetry, and thus prioritize write-dominated pages in DRAM. TwoLRU [35] and Memos [37] only design an asymmetry-aware mechanism for page promotion. Similarly, the AutoNUMA patch designs a page fault mechanism for promotion, giving a higher priority to write-dominated pages, but applies the base active-inactive list management implemented in the kernel for demotion, which only considers page references.

Only a small subset of solutions fail to saturate DRAM capacity. Those that do not fulfill the guideline: (i) allocate pages in NVM when their first-touch is caused by a read operation [25]; (ii) restrict initial allocation to NVM [37]; or (iii) distribute pages proportionally to the tiers' bandwidth [38].

Article	Design Assumptions	Placement Strategy				Implementation	Evaluation	Modifications	
	HMA tiers	Asymmetry-aware Migration		Prioritizes DRAM	BW-Aware	Classification Algorithm	Real System	HW	OS
		Promotion	Demotion						
CLOCK-DWF [25]	DRAM-PCM	✓	✓	✗	✗	CLOCK	✗	✗	✓
M-CLOCK [26]	DRAM-PCM	✓	✓	✓	✗	CLOCK	✗	✗	✓
AC-CLOCK [32]	DRAM-PCM	✓	✓	✓	✗	CLOCK	✗	✓	✓
AIMR [29]	DRAM-NVM	✓	✓	✓	✗	CLOCK+LRU	✗	✓	✓
CLOCK-HM [30]	DRAM-PCM	✓	✓	✓	✗	CLOCK+LRU	✗	✓	✓
Seok et al. [27, 28]	DRAM-PCM	✓	✓	✓	✗	LRU	✗	✓	✓
DualStack [31]	DRAM-PCM	✓	✓	✓	✗	LRU	✗	✓	✓
HeteroOS [33]	MCDRAM-DRAM-NVM	✗	✗	✓	✗	LRU	✗	✓	✓
UIMigrate [34]	DRAM-PCM	✗	✗	✓	✗	LRU	✗	✓	✓
TwoLRU [35]	DRAM-PCM	✓	✗	✓	✗	LRU	✗	✓	✓
AutoNUMA patch [20]	DRAM-DCPMM	✓	✗	✓	✗	LRU	✓	✗	✓
Thermostat [36]	DRAM-DCPMM	✗	✗	✓	✗	TLB Misses	✗	✗	✓
Memos [37]	DRAM-NVM	✓	✗	✗	✓	TLB Misses+CLOCK	✗	✗	✓
Yu et al. [38]	DRAM-PCM	N/A	N/A	✗	✓	N/A	✗	✗	✗
Salkhordeh et al. [39]	DRAM-PCM	✓	✓	✓	✗	PEBS	✗	✗	✗
Ambix	DRAM-DCPMM	✓	✓	✓	✗	CLOCK+PCMon [90]	✓	✗	✓

Table 3.1: Comparison of related work – Guideline fulfillment overview.

Despite our arguments against a BW-aware solution, we identify two proposed solutions – Memos and Yu et al.'s – which consider bandwidth as a metric to guide placement.

Yu et al. [38] propose that pages should always be distributed based on the proportion between to tiers' maximum bandwidth. However, the solution fails to prioritize DRAM saturation in its initial placement, as pages are always distributed according to the tiers' bandwidth ratio. This solution would also be incompatible with DRAM-DCPMM architectures, as current motherboards with DCPMM support require at least one DRAM module per memory channel, whereas the authors assume a single memory tier per socket.

On the other hand, Memos [37] is compatible with DRAM-DCPMM HMAs and presents an asymmetry- and BW-aware promotion mechanism, which migrates pages until it no longer benefits throughput, prioritizing write-intensive pages. However, the solution initially places every page in NVM, and therefore does not prioritize DRAM allocation. Moreover, the proposed demotion mechanism uses only the reference bit in order to select which pages to migrate to NVM.

Our proposed solution, Ambix, will be the first solution to apply the observed guidelines on a real system configured with a DRAM-DCPMM HMA. Ambix requires no hardware changes and only a single line of code added to the kernel. Our main goal is to present a lightweight approach that is as competitive as possible against OS modification-heavy or HW-dependent solutions.

In order to evaluate Ambix, we compare it to Memos and the AutoNUMA patch. Although Memos does not prioritize DRAM saturation, its migration mechanism should give some insight on how a dynamic solution would perform when leveraging bandwidth information to guide placement decisions. Furthermore, Memos considers DCPMM as a possible NVM tier. Moreover, we also add the AutoNUMA patch [20] to the comparison, since it is proposed for our DRAM-DCPMM architecture, and its code is readily available online, with promising results.

Chapter 4

Ambix

In this chapter, we describe Ambix, our dynamic page placement algorithm tailored to a DRAM-ADM DCPMM architecture, as illustrated in Figure 3.1a. We start by summarizing our main goals, which influenced our design and implementation decisions. Then, we provide a theoretical overview of the mechanisms used in our solution, and discuss how we implemented them.

4.1 Goals

Linux's LRU-approximation and base AutoNUMA algorithms perform placement decisions based only on the frequency at which a page is accessed, not taking into account the nature of its accesses. As such, some scenarios emerge, where these algorithms decide to keep read-intensive pages in the faster tier, while evicting less intensive pages, which are modified more often. While this is not a concern in systems populated with latency- and bandwidth-symmetric memories, it becomes relevant in DRAM-DCPMM architectures.

A judicious data placement strategy for these HMAs should be able to leverage the increased memory size of the DCPMM tier, while minimizing writes to it. However, there is currently no mechanism in the Linux kernel that allows write-mitigation to a specific tier, or NUMA node.

Thus, our main goal is to complement the page migration and classification mechanisms currently existing in Linux with a dynamic placement algorithm that is able to classify pages based not only on the frequency of their accesses but also on the accesses' read/write dominance, and use the latter metric to improve data placement in NVM-equipped systems. Moreover, the added metric should only require hardware components that are widely available in commodity architectures, i.e., be as compatible as the existing mechanisms.

We intend to implement the algorithm with as few changes as possible to the Linux kernel. The AutoNUMA patch [20], the only other implemented dynamic solution thought out for a multi-tiered DCPMM-equipped system, relies on extensive kernel modifications in order to function. This introduce multiple limitations we see as avoidable. Firstly, as it rewrites or extends large sections of a chosen kernel, conflicts in future kernel versions are unavoidable, as the memory management subsystem suffers changes

over time. Secondly, even if no conflicts are detected, the solution must be re-patched into every kernel. This means that, if a user requires a specific driver or module introduced in or only compatible with a certain kernel, it is up to the original developer or the user, through time-consuming reverse engineering, to patch the desired kernel. Moreover, if the kernel is patched poorly, the memory management subsystem can be compromised, which can result in system instability or even lead to a full crash. In order to mitigate this, Ambix must offload as many critical components from kernel-space as possible, all the while not sacrificing competitive placement decisions compared to prior HMA-aware placement solutions or DCPMM's hardware-based caching implementation.

We also want to allow programmers to decide whether or not to use Ambix in a per-application basis, rather than operating in the whole system. Furthermore, the solution should be able to adapt to changes in the memory management subsystem, by retaining all of its functionalities, such as page swapping or memory policy management. A situation arises in MemM hardware-based caching and global-scope HMA-aware solutions, where a performance-critical application has some of its accesses met by the slower tier due to a global lack of space in DRAM, which is shared among all running processes. By letting users select which applications should benefit from our algorithm, one can manually override tier allocation to DRAM via a process-scope *bind* memory policy or a similar alternative. However, this comes at the implication of guaranteeing there is a buffer of free space in fast memory for such scenarios. Thus, Ambix must implement a mechanism that foresees these situations.

4.2 Ambix in Theory

HMA systems consisting of DRAM and NVM require an algorithm that selects where to place each process' data, in order to maximize the potential offered by NVM-integration. Ambix considers a DCPMM-equipped system, and leverages the increased memory size DCPMM offers when configured in ADM, while mitigating its limitations related to latency and bandwidth.

We define two main decisions one needs to take before implementing a data placement algorithm for a DCPMM-equipped system:

- At which scope and for which processes should data migration be performed.
- How to decide which data belongs in which tier.

Migration Scope

Since one of our main goals is to minimize changes to the Linux kernel, Ambix leverages the existing migration and classification routines implemented in the Linux kernel, which are performed at page-level.

The algorithm manages page placement within a socket with two memory tiers. Thread balancing and swapping can be optionally configured, working in parallel with Ambix, and may decide to move pages in and out of the monitored socket in order to mitigate load imbalance or free up memory resources. Thus, pages migrated to other unmonitored sockets or evicted to the swap partition are excluded from Ambix's future placement decisions.

The solution must be explicitly bound to running applications in order for them to be affected by our placement decisions. This decision contrasts with the global scope of the AutoNUMA patch and DCPMM's MemM, in which all processes in the system are affected by their placement decisions, as discussed in Section 4.1.

Tier Migration and Page Classification

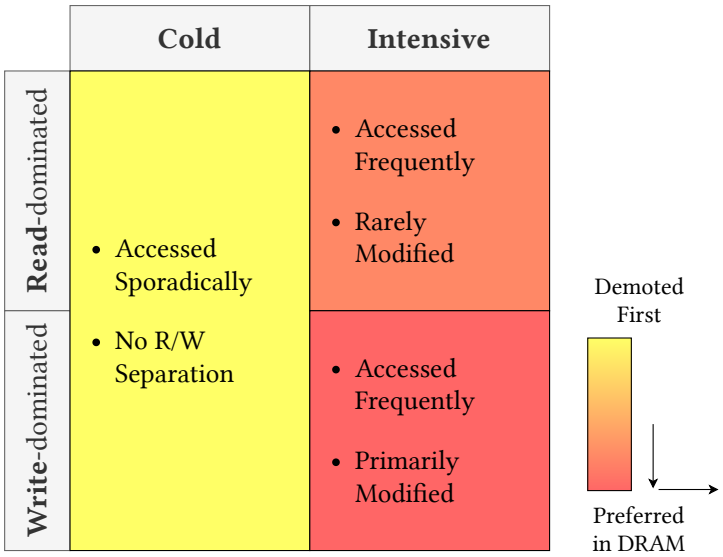


Figure 4.1: Ambix page classification.

Ambix follows the guidelines set in Section 3.2, namely: (i) **Prioritize DRAM Allocation**, and (ii) **Asymmetry-aware Migration**.

In order to fulfill (i), Ambix is combined with Linux's *node local* policy. Under the default *node local* memory policy, a new page is placed in the closest node with available memory, therefore fully saturating DRAM. Furthermore, we extend the guideline to follow the *temporal locality* principle, maximizing the percentage of allocated pages in DRAM. In order to achieve this, Ambix eagerly demotes pages to the slower tier when utilization surpasses a defined threshold. Inversely, when there DRAM has free space, Ambix decides to send pages in the slower tier that are better suited to the faster one.

In order to select which pages to evict, the page replacement algorithm implemented in Linux classifies pages into two categories: frequently referenced, or intensive, and cold. Similarly to prior dynamic placement solutions for NVM HMAs, Ambix adds an additional axis to this classification by separating pages between read and write-dominated, therefore enabling it to have an asymmetry-aware migration mechanism (ii).

In Figure 4.1 we see the two classification axis used by Ambix, where intensive pages are preferred in DRAM, while cold pages are demoted first when DRAM is full. Similarly, write-prone pages are preferred in DRAM over read-prone ones, due to DCPMM's write asymmetry.

Our implementation manipulates the page table entries' reference and dirty (R/D) bits, for each bound process. We decided to minimize page classification overhead as much as possible, relying only on the binary nature of a PTE's R/D bits, and its MMU-managed implementation to classify a page. An

alternative would be to devise a more costly weight- or age-based algorithm, which would need to rely on PTE unmapping or similar mechanism as to induce minor page faults in order to quantify the access frequency of a page. We see these alternatives as no less intrusive than PTE bit manipulation, but with a greater overhead.

In order to manipulate these bits, we leverage a set of existing kernel-space routines related to page table management. This is the only caveat that inhibits Ambix from being fully implemented in user space.

4.3 Ambix in Practice

In this section, we start by outlining the architecture of our solution, specifying its components' role and interaction, and then provide a detailed overview of their individual behavior.

4.3.1 Architecture

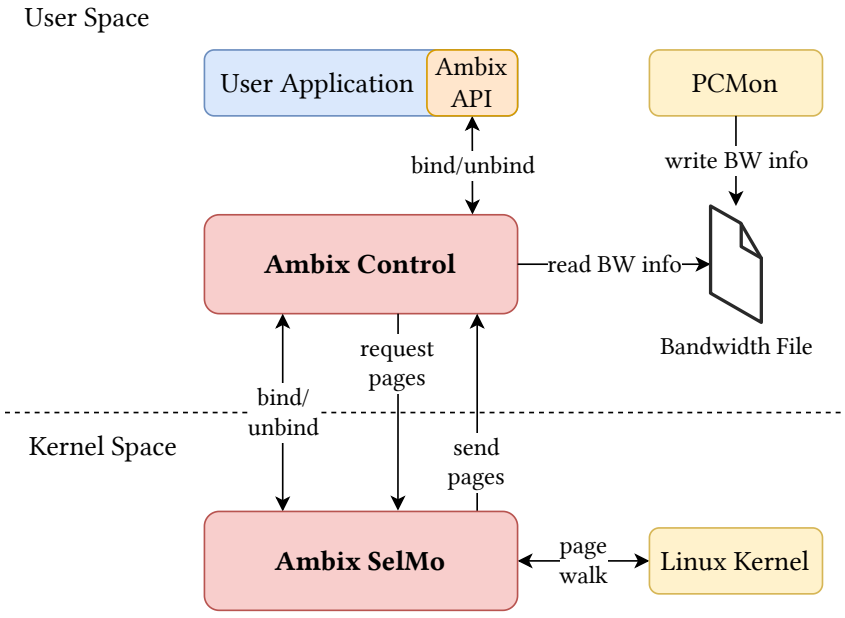


Figure 4.2: Ambix architecture overview.

Figure 4.2 provides an overview of Ambix, presenting its components and how they interact. Our solution consists of two main components: `Control` and the Page Selection Module (`SelMo`).

`Control` is an elevated process running in user-space, which is responsible for formulating and putting into effect new placement decisions. In order to formulate new decisions, the component leverages Processor Counter Monitor (`PCMon`) [90], which periodically outputs the current throughput per node to a shared text file.

To use kernel-implemented mechanisms, Ambix integrates a kernel-space module, named `SelMo`. The module selects pages belonging to bound processes, in order to carry out `Control`'s decisions.

`Control` is Ambix's entry point, being responsible for binding and unbinding applications to our solution. Although Ambix provides specific APIs for binding/unbinding user applications written in C, C++, or Fortran, it supports virtually any target binary using alternative methods, such as user input via `Control` or a provided C wrapper.

The dual-component implementation achieves a small footprint within the kernel, since all mechanisms related to devising and effectuating new placement decisions are offloaded to the user-space component.

4.3.2 SelMo

`SelMo` is the first component launched in Ambix. During its initialization, it creates a netlink socket, which is used to establish bidirectional communication with `Control`.

Then, `Control` is launched, which starts by binding itself to `SelMo`. When `Control` formulates a new placement decision, it sends a `PageFind` request to `SelMo`. The request gives the module information about which and how many pages it needs to find. The module then sends back a list of selected pages that best fit the request.

`SelMo` provides no direct user input, and only accepts defined request structures over the netlink socket. Thus, both components work in a semi-closed loop, where no output related to page information is sent to external processes.

Request

The request structure is defined in a shared header file that is imported by both `Control` and `SelMo`. `Control` sends a request structure over netlink when it wants the module to perform an operation.

The structure has 3 fields:

- `OP CODE`: An integer that characterizes the type of request. There are 3 defined request types: bind, unbind and `PageFind`.
- `PID_N`: Related to the OP code, an integer that may contain a PID for bind and unbind operations, or the number of pages to find for `PageFind` requests.
- `MODE`: Only used for `PageFind` requests. Specifies which pages `Control` is requesting.

Bind/Unbind Request

When a process binds itself to Ambix, it communicates with `Control`, which then forwards the request to `SelMo` with the process' PID. At this point, future placement decisions affect the bound process' page distribution within the socket. A process may also decide to, prematurely or at the end of its execution, unbind itself.

`SelMo` stores the bound processes' information in an array of `task_struct` pointers, which are defined and managed by the Linux kernel. The structure contains the PID of a process and a pointer

Mode	Tier Scope	Goal
DEMOTE	DRAM	Demote cold pages
PROMOTE	DCPMM	Promote pages
PROMOTE_INT	DCPMM	Promote only intensive pages
SWITCH	Bidirectional	Switch intensive with cold pages
DCPMM_CLEAR	DCPMM	Clear the R/D bits from all resident pages

Table 4.1: PageFind modes and goal.

that allows access to its page table, among other information. When a process ends, its task_struct is freed by the kernel. Therefore, before any PageFind request, SelMo updates the array, removing all entries from processes that are no longer running from it.

During a bind request, the module iterates over all tasks, using the kernel's `for_each_process()` macro in order to determine if the process to bind is valid and currently running in the system. If the process is found, its respective task_struct pointer is added to the array.

During an unbind operation, SelMo iterates over the array until it finds a PID match, at which point it removes the entry from the array.

PageFind

When Control devises a new placement decision, it sends a PageFind request to the module. The request is characterized by the number of pages to find and the *mode* of the PageFind operation. The *mode* influences both the page selection criteria and the memory tier from which the module will select pages. In Table 4.1 we summarize the multiple *modes*, describing the tier from which they select pages, and their respective goals.

When a PageFind request is received, the page selection phase begins. The phase iterates over each bound process' page table, in order to select pages according to the *mode* criteria.

When: (i) the number of requested pages exceeds the required amount set in the request; or (ii) the process has iterated over all PTEs, the page selection phase ends. At this point, the last PTE's address and PID is stored and the page selection phase ends. Then, a reply-back phase begins, which prepares a final page array to be sent back to Control.

For each tier, the module keeps two last address and PID pairs, which set the start of the next page selection phase for that tier. Thus, PTEs that have not been inspected for longer are prioritized for migration over recently seen ones. For instance, in a scenario where there is exactly one process bound to SelMo, the last recorded address is greater than 0, and SelMo is unable to find the requested number of suitable pages, then two iterations run: the first starts from the next valid address until the end of the process' address space, and a second one starts from the beginning of the page table until it reaches the last recorded address.

Before sending any PageFind request that will promote pages, Control sends a DCPMM_CLEAR PageFind to SelMo, which instructs the module to clear the R/D bits of all PTEs pointing to pages in

the DCPMM tier. Then, it waits for a configurable `delay`.

Delay

`Delay` affects the access frequency at which a page is considered intensive by `SelMo`. Pages that are accessed or modified during the `delay` interval are considered read- or write-intensive, while all others are classified as cold. A shorter `delay` leads to the promotion of only a smaller subset of frequently accessed pages, while with a more relaxed `delay`, a wider range of pages can be selected.

In general, a higher `delay` is desirable for smaller workloads, when their working set fits in the upper tier, as it leads to the promotion of sporadically accessed pages. On the other hand, a shorter `delay` is better suited to workloads that frequently access a subset of pages which cannot completely fit in the upper tier. In the latter scenario, promoting only the more intensive subset of the workload's pages can maximize the percentage of accesses served by DRAM, as sporadically accessed pages are kept in DCPMM.

Page Table Iteration

Before iterating over a page table, Ambix acquires its lock, assuring that the kernel does not modify its contents during the page selection phase.

Then, we iterate over the page table with the `walk_page_range()` routine, implemented in the Linux kernel, which iterates over a range of a process' page table.

The routine is exported from the kernel with the `EXPORT_SYMBOL()` macro. This is the only modification required to use Ambix and is critical in order for our solution to work, as it enables our solution to access the page table of any process and manage the contents of its PTEs.

`SelMo` passes a PTE callback as an argument to the routine, so that it can observe and manipulate each PTE's R/D bits, The callback is invoked whenever a non-empty PTE is found. Through the callback, the module is also able to obtain the NUMA node where the associated page frame resides and its virtual address. Since we want to perform different operations depending on the goal of the find request, we define multiple callbacks, one for each *mode*.

There are three groups of pages in all callbacks (except `DCPMM_CLEAR`, which simply has R/D bits cleared):

- Priority: The priority group contains the best candidate pages that fulfill the `PageFind` request criteria. Pages placed in this group are always sent back to `Control`.
- Backup: The backup group contains pages that meet the selection criteria, although are worse suited than those in the priority group. Backup pages may be sent to `Control` if the priority pages fail to meet the number of requested pages.
- Retain: The retain group contains pages that the algorithm decides to retain in their current memory tier, since they fail to meet the selection criteria.

The backup group is not needed in page replacement algorithms designed for traditional architectures, since in their implementation the reference bit, or an analogous metric, is sufficient to classify pages. However, after clearing the R/D bits, pages with only one bit set become indistinguishable from pages with both bits unset. Thus, the former are detached into a backup group, and can be selected to migrate when the priority pages are insufficient.

If the callback's goal is to demote pages, then it clears the R/D bits of all pages that are not in the priority group. If one such page is referenced thereafter, the memory management unit (MMU) sets its PTE's reference bit; and also its dirty bit, in the event of a store operation. In contrast, if the page is not accessed until the next page table iteration, then it is suitable for demotion.

If, on the other hand, the callback's goal is to promote pages, then PTEs are expected to have both their R/D bits unset, since they have been recently cleared by the module. In this scenario, the MMU may change the PTE's R/D bits, so that the respective page is suitable for promotion over a next page table iteration. Therefore, promotion callbacks do not directly manipulate these bits. Instead, the algorithm deems a page in DCPMM read-intensive if only referenced during the `delay` window, and write-intensive if modified.

Pages in the priority or backup groups are placed in two arrays. The arrays store entries that contain the virtual address and PID of a page, both of which are required by `Control` in order to migrate the page. The module also initializes two variables, which keep track of the number of priority and backup pages found so far.

When the number of priority pages meets the required amount set in the `PageFind` request, or the process has iterated over all PTEs, the page selection phase ends. In the latter scenario, the module was unable to find a sufficient number of pages that meet the priority criteria. In this case, the module proceeds to add the remaining entries from the backup array to the final array. If the pages from the backup array still fail to meet the demanded value, the module sends an array with less than the requested entries back to user-space, consisting of all selected priority and backup pages.

We will now describe how each callback manages PTEs, and how the priority and backup arrays are populated with pages.

PageFind Callbacks

In Algorithm 1, we present the callback associated to a `DEMOTE PageFind`. Its goal is to find cold, or, as a backup, read-intensive pages from DRAM.

The associated callback prioritizes pages that have their reference bit unset. As backup, pages which have not been modified recently but are considered read-intensive may also be selected. This is achieved by calling the `pte_young()` (line 6) and `pte_dirty()` (line 10) functions, defined in the Linux kernel, which indicate if a PTE has its R and D bits set, respectively.

In order to obtain the NUMA node in which a page is allocated, two kernel functions are used. Firstly, the PTE is converted to a page frame number (PFN), via the `pte_to_pfn()` function. Then, the `pfn_to_nid()` function outputs the node where the page frame resides (line 4). The NUMA nodes

Algorithm 1: DEMOTE Callback

```
   global_input: curr_pid, pages_array, bak_pages_array, pages_found, bak_pages_found,
                 pages_to_find, last_addr_dram
   input       : pte, address
 1 if pages_found = pages_to_find then
 2 |   last_addr_dram := address ;
 3 |   return 1 ;                                                    // end pagewalk
 4 if !pte_present(pte) or !pte_write(pte) or pfn_to_nid(pte_to_pfn(pte)) != DRAM then
   |   // pte not present, write protected, or not in DRAM
 5 |   return 0 ;                                                    // continue pagewalk
 6 if !pte_young(pte) then
 7 |   put address and curr_pid in pages_array;
 8 |   increment pages_found;
 9 |   return 0;
10 if !pte_dirty(pte) and bak_pages_found < (pages_to_find − pages_found) then
11 |   put address and curr_pid in bak_pages_array;
12 |   increment bak_pages_found;
13 old_pte := ptep_modify_prot_start(..., pte);
   // clear R bit
14 old_pte := pte_mkold(old_pte);
   // clear dirty (D) bit
15 old_pte := pte_mkclean(old_pte);
16 ptep_modify_prot_commit(..., old_pte, pte);
17 return 0;
```

specific to the DRAM and DCPMM tiers are statically defined in the shared header file, imported by both of Ambix's components.

Every observed page that does not meet the priority array criteria has its PTE's R/D bits cleared by the callback (lines 13-16). In order to achieve this, a temporary copy of the original PTE is created. Then, its bits are cleared with the `pte_mkold()` and `pte_mkclean()` routines. Finally, the temporary PTE is written over the original one, effectively changing its R/D bit information while keeping all other fields intact.

When a `PROMOTE PageFind` is requested, the module selects pages to promote to DRAM. The callback selects any page from DCPMM, with an emphasis on write-intensive pages, which are placed in the priority array. In this callback, read-intensive pages are attributed the same priority as cold pages, as the main goal of the operation is to maximize space utilization in DRAM. Unlike the `DEMOTE PageFind` callback, it does not clear R/D bits from any pages, since it relies on the `delay`-based mechanism.

In scenarios where DRAM space is scarce, `Control` sends a `PROMOTE_INT PageFind`, which is associated to a callback where only write or read-intensive pages are selected for promotion. In the `PROMOTE_INT PageFind`, the module selects N pages that were recently modified or accessed to promote to the faster tier. The associated callback places pages with both the reference and dirty bits set in the priority array. The backup array is populated with read-intensive pages, which only have their reference bit set.

The `SWITCH PageFind` differs from the previous variants, as the module is requested to find an equal number of pages to swap between both tiers. In this mode, we perform two page table iterations. The first one selects DCPMM-resident pages with the `PROMOTE_INT` callback. The module then looks

for the same number of pages in DRAM, using the DEMOTE callback. After both iterations, SelMo reconstructs the page array, such that the number of pages selected from each tier are equal. Moreover, the backup pages from a tier are only added to the array if it is matched with a page that is in the priority array of the other tier. Hence, Ambix avoids switching equally intensive pages. At this point, a separator is also added to the middle of the array, indicating Control to reverse the migration orientation for the subsequent pages.

The DCPMM_CLEAR PageFind also differs from the previous, as no pages are selected or sent back to Control. The callback precedes a find operation in DCPMM, and is used to tune the frequency at which a page is considered write- or read-intensive.

Reply-back Phase

After a bind/unbind or a PageFind request, the module creates a response structure. The response consists of an array of one or more structures, called addr_info. In the bind/unbind and DCPMM_CLEAR PageFind, the array is composed of one entry, which contains information on the success of the requested operation. In the other find operations, it additionally contains the virtual addresses and PIDs of the pages selected during the corresponding pagewalk, and a separator entry in the SWITCH PageFind.

The addr_info structure is defined as follows:

- addr: Contains the virtual address of a page.
- pid_retval: Contains the PID associated to the virtual address or the return value of the operation, if it is the last entry of the array.

The module then groups the array entries into multiple netlink packets, and sends them to Control.

4.3.3 Control

When Control starts, it tries to bind itself to the kernel module via the netlink protocol. If the bind is successful, it launches 3 threads: the Socket thread, which manages incoming bind/unbind requests from user processes; the Console thread, which processes console input; and the Placement thread, which devises and puts into effect new placement decisions.

Socket Thread

The Socket thread manages incoming requests from running processes. We leverage a stream-oriented Unix Domain Socket (UDS), which allows multiple processes to send bind and unbind requests in parallel. The created endpoint exists as a file in the path where Control runs. Once the endpoint is created, Control listens to it, expecting user requests.

The user request structure consists of two fields, which identify the type of request, which can either be a bind or unbind operation, and the PID of the process to bind.

`Control` does not maintain a list of active PIDs. As such, when a request is received, a simple sanity check is performed, which assures that the PID is within the valid range as defined in the Linux kernel. If it is valid, a netlink request is created and sent to the kernel module.

In C/C++/Fortran applications, the preferred method of associating the compiled executable to Ambix is to call the bind/unbind functions implemented in Ambix API, at the start and end of the execution, respectively. However, in other languages, Ambix currently provides no direct way of communicating with `Control` through the application. Instead, when launching a process, its PID may be passed as an argument to a piped executable written in C, which sends a bind request to `Control` with the process' PID. Similarly, when the process terminates, an unbind executable may be used.

Console Thread

The `Console` thread processes user input through command input. The defined commands allow users to manually bind/unbind processes to Ambix, as well as run multiple debug commands. The bind and unbind commands come as a third alternative to facilitate associating the process to Ambix.

The valid commands are:

- `bind`, `unbind`: Requires a PID argument. When the user inputs a bind/unbind command, `Control` creates and sends a netlink request to `SelMo`.
- `send`: Debug command which requires two arguments. The first argument specifies the migration orientation, either to promote or demote pages. The second argument specifies the number of pages to migrate. When the user inputs a valid send command, `Control` sends a find request to the `SelMo` and migrates the pages selected by it to the new tier.
- `switch`: Debug command that performs a `SWITCH PageFind`. Requires a single argument that specifies the number of pages to switch between the DRAM and DCPMM tiers.
- `toggle`: Also used for debug purposes. The command requires an argument which specifies which `Control` component to toggle. The valid arguments are `switch`, which toggles page switching, and `thresh` which toggles threshold balancing. If both components are disabled, Ambix does not migrate pages.
- `clear`: Clears the console screen.
- `exit`: Ends all threads and exits `Control`.

Placement Thread

The placement thread is responsible for periodically monitoring current memory usage and bandwidth values. Depending on the collected metrics, the thread devises a new placement decision, sends a find request to `SelMo` and migrates the selected pages to their new tier.

In order to get the current memory allocation of each tier, `Control` leverages the `libnuma` library [70], which provides per-node statistics on total node size and utilization. Bandwidth information is

obtained with PCMon, which we modify to output current per-node bandwidth information to a shared file, at the same periodicity as the placement thread.

By design, DRAM has a statically defined maximum usage threshold below its actual size. Above the threshold, Control considers that the tier is full or near to depletion. The resulting buffer should be large enough to allow newly referenced pages to fit in the faster tier, while not provoking Ambix to demote pages too eagerly, which could adversely impact the bound processes' throughput.

Similarly, DCPMM has a write throughput threshold. If DCPMM's current throughput is above the defined threshold, Control considers that the tier contains a significant amount of write-intensive pages.

New placement decisions are devised in two components: Switch, and Threshold. Switch triggers a new placement decision when the DCPMM's write throughput exceeds a threshold. Then, the Threshold component is activated, which devises a new placement decision that maximizes the DRAM utilization, while keeping a buffer of free space for new pages.

We will now describe the placement components' behavior and how the collected metrics are used by them to devise new placement decisions.

Switch

The Switch component triggers only when DCPMM is above its write throughput threshold, indicating that the tier contains write-intensive pages. Although the component's primary goal is to promote write-intensive pages to DRAM, it may also select read-intensive if it finds sufficient free space in the tier, or cold pages that could be exchanged with the promoted ones.
When Switch is triggered, Control sends a PageFind request, the *mode* of which depends on the following conditions:

- If the faster tier is below its utilization threshold, Control decides to promote as many intensive pages as possible such that the faster tier's threshold is not surpassed. In this scenario, it send a PROMOTE_INT PageFind to the module, which prioritizes write-intensive pages.

- In contrast, if the faster tier is above its usage threshold, Control decides to switch as many pages as possible between the tiers, by sending a SWITCH PageFind request to the module. In this scenario, the current memory usage in both tiers is maintained, as SelMo selects an equal number of pages from each tier, but minimizes DCPMM's write throughput, as the demoted pages either cold or read-intensive.

Threshold

The Threshold component triggers a placement decision when the DRAM is at capacity or under-provisioned. Three different placement decisions are devised based on the observed metrics:

- If DRAM is above its maximum utilization threshold but there is space available in DCPMM, Control requests cold pages from the faster tier via a DEMOTE PageFind.

- If the Switch component is toggled off via the Console thread, and DRAM is below a minimum usage threshold, a PROMOTE PageFind is requested to SelMo. In this scenario, the module may select cold pages from the slower tier, in contrast to the PROMOTE_INT or SWITCH PageFinds used by the Switch component, since the main goal of the decision is to maximize the faster tier's utilization.
- If both tiers are near to depletion, the current page allocation is maintained.

If, after a find request, SelMo sends back a non-empty array of selected pages, Control migrates them to their new tier. This is achieved with the `move_pages()` syscall, which is implemented in the Linux kernel and available to user-space applications.

4.4 Ambix's Optimizations and Limitations

We minimize the number of system calls performed during Control's migration phase, by grouping pages from the same process into a single call. In the SWITCH PageFind scenario, an additional same-goal optimization is implemented, where Control migrates as many pages as physically possible, i.e., until exhausting the destination tier's available memory, before reversing the migration orientation.

Workloads with uniform access distributions, characterized by an equally small or large reuse interval among their pages, might not benefit from Control's Switch component. In the small reuse distance scenario, Switch prevents cyclic migrations by choosing to retain an intensive page in the slower tier, when it does not find a corresponding cold page from the faster tier. However, in workloads with large reuse distances, or ephemeral accesses to their pages, SelMo chooses to promote recently accessed pages and switch them with pages that, although have not been recently referenced, will be accessed again sooner in the future. Graph searching algorithms, such as BFS or DFS, are prime examples of the latter scenario, where a graph's node is referenced, leading to the promotion of its associated pages to the faster tier, although it will only be referenced again after iterating over multiple other nodes.

Thus, when binding processes with uniform or near-uniform access distributions, the Switch component can be optionally toggled off via the Console thread. In this situation, some scenarios might emerge, where faster tier space changes during the workload's execution, as other non-bound processes allocate pages to the tier, and thus cause Control to demote pages from its bound processes. Without the Switch component, pages would not be promoted back to the upper tier as non-bound processes free their memory. Thus, the Threshold component provides a branch that only triggers when Switch is toggled off and the faster tier has available capacity, such that its utilization is maximized.

In both components, a defined variable dictates the maximum number of pages that can be requested to SelMo. The variable can be set to a maximum of 2GB worth of pages, due to a packet and payload limitation of the netlink socket. However, since migration throughput is primarily bottlenecked by the NVM's bandwidth, we find the netlink-imposed limit to be adequate for all workloads.

Chapter 5

Results

In this chapter, we present the experimental results we obtained while evaluating Ambix. We start by enumerating our evaluation's goals, then specify our experimental setup, baseline, and chosen workloads. Finally, we introduce and discuss the attained results.

5.1 Goals

Our main goal is to understand how Ambix performs in workloads with different characteristics, and comparing it against: (i) HMA-aware dynamic placement solutions proposed in past literature, and (ii) placement options that are currently available in off-the-shelf DCPMM-equipped Linux systems. We will explore how a workload's throughput is affected in each configuration, comparing workloads with varying read/write ratios, locality, and access patterns.

In (i), we choose Intel's AutoNUMA patch [20] and Memos [37], as we believe that these solutions are closest to the state of the art in dynamic placement, and present mechanisms that could be implemented with existing hardware.

In (ii), we consider: (a) DRAM and ADM DCPMM with the default *node local* NUMA policy, without any dynamic placement solution applied; and (b) MemM DCPMM.[1]. The former represents a two-tiered configuration with no tier migration, and the latter provides a hardware-managed caching algorithm, which dynamically places and evicts intensive data to and from DRAM.

Additionally, our evaluation aims at addressing the following questions:

- What is the overhead of each solution when the workload does not benefit from having having its pages distributed, such as in workloads with low footprint or that present a near-uniform access pattern.

- How effective Ambix is in maximizing the percentage of total and write accesses to DRAM.

[1] We will abbreviate these configurations as *ADM-default* and *MemM*, respectively.

5.2 Experimental Setup

5.2.1 Hardware Configuration

We configure a Supermicro X11DDW-L motherboard with four 16GB DDR4 DRAM DIMMs and four 128GB DCPMMs. The motherboard has two sockets, each running a 16c/32t Intel® Xeon® Gold 5218 CPU, at a base frequency of 2.30 GHz. It has a total of six memory channels, each comprised of two memory slots, totaling six slots and three memory channels per socket.

We populate two channels with one 16GB DRAM DIMM and one 128GB DCPMM each (1-1). This achieves a total memory of 288GB per socket when DCPMM is configured in ADM and 256GB when configured in MemM, since the DRAM modules are configured as a hardware-managed cache in the latter scenario.

We find that our selected configuration, even if only utilizing two thirds of the available memory slots, represents a good estimate of both the bandwidth and latency capabilities offered by the introduction of DCPMM, when integrated into a memory-constrained system.

We configure each socket's DCPMM in ADM, as a NUMA node, which is transparently granted a distance smaller than remote DRAM, i.e., pages are prioritized in the local DCPMM node before being allocated in remote memory. This configuration is applied in all scenarios except when testing the hardware-based caching approach, where DCPMM is configured in MemM. However, when DCPMM is configured in ADM, DRAM's usable capacity is limited to $\sim$27GB due to idle system resources, similarly to the system configuration in Section 3.2.

5.2.2 OS Configuration

Since DCPMM is a bleeding edge technology, getting new versions of drivers and memory management libraries is paramount for our work. Therefore, we choose Debian, a widely used Linux distribution, which allows us to easily modify and compile any chosen kernel.

All experiments except those with the AutoNUMA patch run on the v5.8.5 kernel.

We reconfigure some of the kernel's components in order to focus our efforts on the study of DRAM-DCPMM interaction. We choose to disable AutoNUMA balancing, and set the *swappinness* value to 0. The former disables AutoNUMA's thread scheduling and page migration between NUMA nodes, which is active by default, while the latter limits swapping to a memory depleted scenario, effectively disabling it for the workloads we run. Although Ambix was tested and works with Linux's swapping mechanism, we disable it in our experiments as it could introduce avoidable variables to our evaluation.

We run experiments with the AutoNUMA patch on a separate kernel (v5.5), since the patch makes extensive and fundamental changes to Linux's memory management subsystem. For this kernel, we leave the AutoNUMA balancing and *swappinness* values to their default values. This is an essential step, since otherwise AutoNUMA would not perform any migrations.

In all workloads we run, we limit CPU utilization to a single socket, via the *numactl* CLI, and also bind it to the DRAM and DCPMM NUMA nodes within the CPU's socket or simply to the DCPMM node in the

MemM scenario.

5.2.3 Ambix Configuration

As a prerequisite to using Ambix, we patch the v5.8.5 with a single line of code, which makes the `walk_page_range()` routine callable from kernel-level modules.

We assume that, as traditional in dynamic solutions, no information about each bound workload is known before runtime. Therefore, we configure Ambix identically for all workloads. Our chosen values come from a prior study with multiple executions of a variety of benchmarks, from pmbench [91, 92] and the Parsec [71] and NPB [21, 22] suites. We settled for values that provide the best possible throughput gain in all scenarios.

The used variables are defined as follows:

- `Control`'s periodicity, i.e., the frequency at which the user-level component performs new placement decisions, is set to 2 seconds.
- The DRAM target threshold variable is set to 0.95, therefore keeping at least 5% free space in DRAM at all times, demoting pages if needed.
- The clear mechanism `delay`, used before selecting DCPMM-resident pages to promote, is set to 50ms.
- The `Switch` component is always activated, therefore having no need to set a target threshold for DCPMM.
- DCPMM's write throughput threshold (monitored by PCMon), after which `Control` reacts to the presence of write-intensive pages in DCPMM is set to 50MB/s.

5.3 Experimental Baseline

This section presents further information about our chosen baselines, and gives some initial insights on how we think they will perform when pitted against each other.

Ambix can be directly compared to the *ADM-default*, the AutoNUMA patch, and Memos configurations, since we configured DCPMM identically. We can also extract some insight on whether or not the *MemM* configuration improves system performance when compared to the former ADM-based page placement solutions and, if so, rank it against the dynamic ones.

5.3.1 HMA-Aware Placement Solutions

Our solution is tested against Intel's AutoNUMA patch, which is the only publicly available dynamic page placement solution designed specifically for systems that integrate DCPMM. The patch relies on an ADM

configuration, where both DRAM and DCPMM are directly accessible and seen as NUMA nodes by the system [2].

The authors provide multiple patched versions, available in a Git repository [93]. We choose the *tiering-0.4* version, which is the most up-to-date documented version currently, based on the v5.5 kernel. We configure the kernel and run the post boot setup as proposed in the documentation, using the recommended settings for performance experiments.

Additionally, we incorporate Memos into the comparison, as it presents a bandwidth-aware approach, which starts by initially placing every page in NVM (see Section 2.3.2. Memos proposes a full-fledged solution that has other focuses besides page placement, such as bank imbalance, alternative migration techniques, and an in-house TLB miss profiler.

We decided to implement a simplified version, strictly focusing on the proposed placement algorithm, relying on the mechanisms implemented in Ambix to classify pages, as the solution was unfortunately not publicly available nor provided by the authors. Memos' current parametrization is suited to low footprint workloads, only migrating a maximum of 10,000 pages at each cycle, which correlates to 1MB/s. Therefore, we change it to make it as competitive as possible with our chosen workloads, as some will be multiple times larger than DRAM size ($>$27GB).

Firstly, we tighten Memos' periodicity from 40s to 4s and, in order to fit in the new 4s period, lower the number of required page classifications to a single one, sacrificing accuracy for performance. Secondly, we increase the maximum number of pages that can be migrated in a given period to 10x its original value, allowing 100,000 pages to be promoted. Both changes increase Memos' migration rate-limit a hundredfold, to 100MB/s [3].

We expect that Memos, even though it presents bandwidth- and asymmetry-aware placement mechanisms, will have poor performance in low footprint workloads, due to its initial placement policy. Furthermore, in all workloads, its throughput-dependent migration policy may become affected by a transition to different phases of execution, where throughput may vary due to other factors besides the previous migration, and thus trigger a suboptimal migration decision in the following cycle.

Since we implement Memos using Ambix's user- and kernel-level modules we decide to leave the DRAM threshold and `delay` to their default values.

5.3.2 Default Configurations

We benchmark dynamic placement solutions against the *ADM-default* and *MemM* configurations.

Although the former provides no dynamic placement decisions, we expect it to perform best in workloads with poor locality, as we expect migration or caching decisions to worsen performance when applied to random or uniformly-distributed access patterns, due to the classification and migration/caching overhead.

We expect that the *MemM* configuration will be competitive against HMA-aware algorithms in high locality and low footprint scenarios, as it can cache a workload's entire intensive working set in DRAM,

[2] This configuration will be referred to as *autonuma*.
[3] We will refer to this plain version as *memos*.

and therefore have comparable performance to HMA-aware solutions. However, its performance should fall off in workloads with higher footprint or asymmetric data accesses, as it not only does not distinguish write- and read-dominated pages, but also always caches pages on accesses, which could lead to some thrashing due to the promotion and soon-after demotion of sporadically-accessed pages.

5.4 Workloads

In order to present a comprehensive assessment of all configurations, we start our evaluation with the pmbench synthetic benchmark [91, 92], which grants us a higher degree of control over the generated workloads. We then move on to benchmarks from two well-known HPC-related suites, NAS Parallel Benchmarks (NPB) [21, 22] and GAP [23, 24].

All chosen benchmarks are based on OpenMP (OMP) [94]. We configure the `OMP_NUM_THREADS` environment variable to 32, therefore fully utilizing all available available cores in a single socket.

The next sections describe each of the benchmarks and applied parametrizations, as well as what information we want to extract for each of them. In GAP and NPB, we used `PCMon` in order to determine each workload's read to write distribution.

5.4.1 Pmbench

We choose the pmbench micro-benchmark, which was originally designed to test paging performance in systems equipped with non-volatile storage devices. We find that pmbench's configuration options are able to replicate common memory access patterns. Furthermore, it was also selected by the authors of the AutoNUMA patch in their presented evaluation [20].

Pmbench allows us to benchmark workloads with varying degrees of locality, which are generated based on a range of mathematical patterns. It also allows setting different workload sizes, and diversify the percentage of read to write operations.

Before running the timed portion of the benchmark, pmbench allocates and initializes the full test array sequentially. Thus, lower entries are placed first in DRAM in all configuration except *memos* and *MemM*, which allocate the full array in DCPMM.

We parametrize pmbench with a Gaussian distribution, varying the standard deviation (σ) parameter. Higher σ values generate workload patterns with less locality, akin to a uniform distribution, while lower σ values generate a workload where most accesses are to the pages at the center of the array and the extremities are rarely accessed.

Our chosen σ values are 12.5% and 25% of the array size, i.e., total number of allocated pages. As traditional in a Gaussian distribution, 68% of accesses are within one σ from the middle point, or mean, of the array, and 95% within two σ. This means that, for $\sigma = 12.5\%$, 95% of all accesses are to 50% of the pages, where these pages are located in the 25% to 75% range of the array. In contrast, for $\sigma = 25\%$, pages in the 0-25 and 75-100% range receive 38% of accesses, therefore generating a workload with less locality.

The percentage of read to write operations is also tuned to different values: 20%, 50%, and 80%. However, this configuration only affects the probability of a page access being a read or write operation, and thus every page has a similar read/write-dominance

We also define different workload sizes ranging from 32 to 240 GB. The chosen range always allows the workloads to always fit in a single socket.

Our chosen parametrization enables us to observe how DCPMM's higher write latency affects overall throughput, as well as compare the scalability of all dynamic configurations in scenarios with more or less locality. However, it is unable to evaluate the asymmetry-aware mechanisms existing in *memos*, *autonuma*, and *ambix*.

5.4.2 GAP

The Graph Algorithm Platform (GAP) benchmark suite emulates the behavior of graph-processing applications by presenting traverse- and compute-centric benchmarks which perform common operations over a generated or inputted graph.

We choose the breadth-first search (BFS) benchmark from GAP v1.3, which traverses all nodes of a fixed size graph. The benchmark is parametrized to generate graphs of 2^n nodes, with $n \in 27, 28, 29$, which result in a memory footprint of 35, 70 and 140GB, respectively, running for 50 iterations each. The workloads have a good balance between read and write accesses (2R:1W).

Due to the nature of the BFS algorithm, we expect the workload to have an irregular access pattern, where each node, and consequentially each page, is referenced a similar number of times, with no pages being more read- or write-dominated than others. Despite being irregular, the workload has sequential-like properties, as nodes at each depth are traversed in sequence.

Therefore, GAP BFS will give us insight on how each configuration performs when applied to a workload with a sequential access pattern, where pages have a similar read/write-dominance and access frequency. Furthermore, we hypothesize that BFS will induce a lot of thrashing in the Ambix, *memos*, *autonuma* and *MemM* configurations, as pages intensive at a certain point in time will not be referenced throughout the remainder of the current iteration. Thus, GAP BFS will also give some insight on how these configurations perform in a thrashing-heavy scenario.

5.4.3 NPB

The Numerical Aerodynamic Simulation (NAS) Parallel Benchmark suite presents multiple benchmarks which mimic common access patterns in computational fluid dynamics applications, and was designed to evaluate the performance of parallel supercomputers.

We choose 4 benchmarks from the OMP version of NPB v3.4.1, which present a good balance between computational cost and memory bandwidth requirements, and have different read/write intensity:

- BT: Block tridiagonal matrix solver. Generates read-intensive workloads (3.5R:1W).

Benchmark	Parametrization	Memory Requirements (GB)
BT	small	28.4
	medium	39.1
	large	53.9
FT	small	20
	medium	40
	large	80
MG	small	26.5
	medium	74.3
	large	131
CG	small	18
	medium	39.8
	large	150

Table 5.1: NPB's BT, FT, MG, and CG memory requirements.

- FT: Discrete Fast Fourier Transform algorithm over a three dimensional data set. Very write-intensive (1.7R:1W).
- MG: Computes a multigrid algorithm over a sequence of meshes. Read-intensive (4R:1W).
- CG: Runs the conjugate gradient method on a system of linear equations. Very read-intensive ($>$60R:1W).

Each benchmark presents up to 8 pre-defined problem sizes, where the largest require multiple terabytes of memory. We instead choose to directly modify the parametrizations the benchmarks' source code, in order to tailor it to our system's memory capacity and available processing power.

For each NPB benchmark, we evaluate small ($\sim$0.8x DRAM size), medium ($\sim$1.5x) and large ($\sim$3.5x) parametrizations, which we present in Table 5.1.

Unlike the previous, these benchmarks have write- and read-dominated pages, and have been previously used to evaluate the static solutions proposed in Servat et al.[40] and Unimem [41]. Therefore, the benchmarks represent the most common use case for an asymmetry-aware solution, when applied to a DRAM-DCPMM system.

5.5 Experimental Results

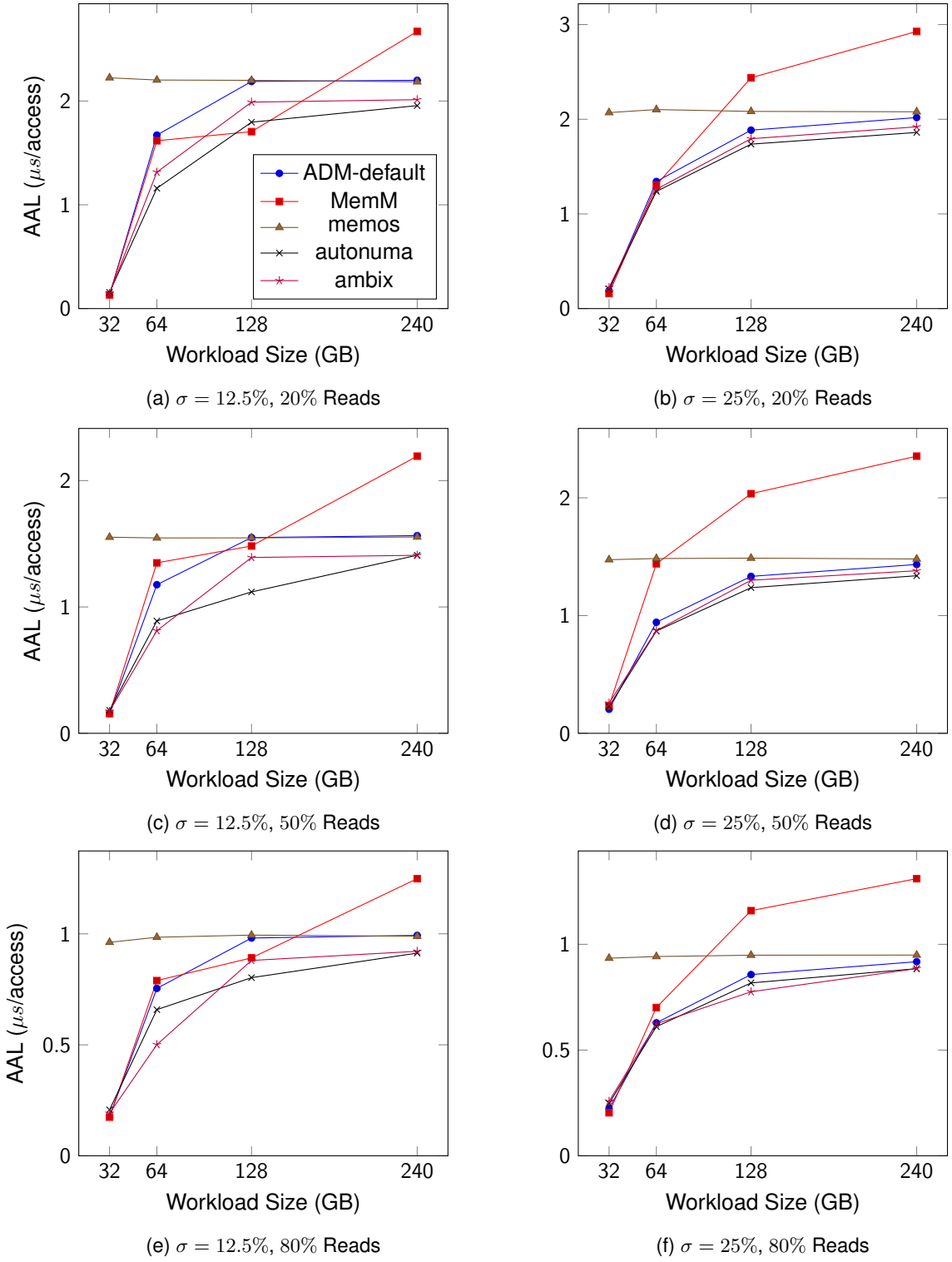
In this section, we present the experimental results obtained with all configurations. We will start by presenting all results obtained with pmbench, GAP, and NPB, respectively. For each benchmark, we compare the performance attained with each configuration, and talk over details specific to each benchmark. We then conclude the section by providing an overview of all configurations, specifying their respective benefits and weaknesses.

5.5.1 Pmbench – Scalability Potential

Pmbench generates a report at the end of a benchmark, where it outputs the average access latency (AAL) in microseconds per access, as well as the average clock cycles taken per access. We choose

the AAL metric for presenting and discussing the experimental data.

We will use the presented results to evaluate each configuration's scalability potential, or lack thereof. All selected workload sizes surpass DRAM size (27GB).



(a) $\sigma = 12.5\%$, 20% Reads

(b) $\sigma = 25\%$, 20% Reads

(c) $\sigma = 12.5\%$, 50% Reads

(d) $\sigma = 25\%$, 50% Reads

(e) $\sigma = 12.5\%$, 80% Reads

(f) $\sigma = 25\%$, 80% Reads

Figure 5.1: Pmbench plots. Varying workload size and locality, single socket, 32 threads.

In Figure 5.1, we can see the pmbench plots with varying sizes, σ, and read/write ratio.

ADM-default

In the *ADM-default* plots, we observe a logarithmic growth in AAL in all scenarios, settling around the 128GB range in the $\sigma = 12.5\%$ workloads, depicted in Figures 5.1a, 5.1c and 5.1e.

The *ADM-default* configuration initially places pages in the nearest node with free memory, and maintains them in their original node for the duration of the workload. Since pmbench initializes the page array before the workload runs, the lower entries of the array are placed in DRAM. This means that, in the 32GB workloads, the configuration only allocates the last 5GB of the array's pages in DCPMM. In the 32GB $\sigma = 12.5\%$ workloads, the DCPMM-resident pages fall outside the 2σ range, and therefore are seldom referenced.

For the 64GB $\sigma = 12.5\%$ workloads, 3GB of pages that were placed in DRAM fall within the σ range, and therefore are frequently requested. However, for both the 128 and 240GB workloads almost 100% of accesses are performed to pages placed in DCPMM, as the middle point of the array is allocated far inside DCPMM, with all DRAM-resident falling outside of the 2σ range. These configurations output similar AAL compared to each other, but worse compared to the 64GB parametrization.

In the 25% sigma scenario, the less localized nature of the workload makes it so that the percentage of DRAM Hits is greater in the 128GB workloads compared to the 240GB, and as such performance degrades further in the latter. Moreover, even in the 32GB parametrization, wherein the configuration allocates only the last 5GB of pages in DCPMM, some accesses are performed to the tail ends of the array (within 2σ). Therefore, compared to the 32GB $\sigma = 12.5\%$ workloads, AAL increases by an average of 25% across all read/write distributions, with a peak increase of 30% in the write-intensive 20% reads workload.

ADM-based Dynamic Solutions

In larger workload sizes, the *autonuma* and *ambix* solutions initially allocate the middle part of the array in DCPMM, identically to the *ADM-default* configuration, but dynamically migrate them between tiers. Moreover, the pages from the initial entries of the array, which are allocated in DRAM, will be infrequently accessed, especially in the more localized workloads. Thus, as the workload size increases, these configurations must exchange a larger amount of pages in order to improve the workload's throughput. This allows us to discuss and compare the scalability potential of both solutions.

In the 64-240GB $\sigma = 12.5\%$ workloads, we see an improvement in AAL in these configurations compared to the static *ADM-default*. In this range, the *ambix* configuration improves AAL by 16%, while *autonuma* improves by 18%. *ambix* outperforms *autonuma* in the 64GB workloads, by an average of 6.3%. However, is it outperformed by 12% in the 128GB workloads, with no performance difference in the 240GB parametrizations.

In the 64-240GB $\sigma = 25\%$ workloads, both the *autonuma* and *ambix* configurations have a lesser benefit compared to *ADM-default*, with an average improvement of 5 and 6%, respectively.

The *memos* configuration initially places every page in DCPMM, and migrates them based on the previous migration's throughput benefit. However, its implementation proves ineffective, as throughput is

affected not only by prior migrations but also due to access pattern changes, where the latter introduces noise to Memos' BW-aware mechanism, and causes suboptimal migration decisions.

In *memos*, the majority of accesses are fulfilled by DCPMM, as reflected by the identical AAL to *ADM-default* in the 240GB $\sigma = 12.5\%$ workloads, which, as discussed, have almost all accesses fulfilled by DCPMM. This conclusion is further supported by the observed DRAM and DCPMM throughput, monitored with `PCMon`, at different stages of execution. Despite some fluctuations in the DRAM and DCPMM respective throughputs due to some pages migrating between tiers, the aggregate throughput did not improve throughout the workloads' execution.

We conclude that the *autonuma* configuration scales better than *ambix* by a small margin in workloads with a footprint close to or above 128GB, but has no advantage in the 240GB parametrizations. *memos*, on the other hand, is unable to improve system throughput with its current placement strategy, and leaves the majority of all frequently accessed pages in DCPMM, therefore having a constant AAL in all scenarios.

MemM

The *MemM* configuration dynamically caches intensive data in DRAM, replacing the least recently used cache blocks when full. As such, in workloads that frequently access more pages than those that can fit in the DRAM cache, the caching algorithm will frequently replace data, leading to an increase in data movement costs associated with caching and evicting pages.

This phenomenon can be observed in all $\sigma = 12.5\%$ workloads. We see that AAL is similar in the 64 and 128GB parametrizations, but has a sharp increase in the 240GB compared to the 128GB workloads. For the 64GB workloads, almost all accesses are performed to 32GB of pages (within 2σ), which allows MemM to cache these pages and maintain them cached for the duration of the workload. For the 128GB workloads, performance does not degrade, even though the same subset of pages only receives about 68% of accesses. Thus, the *MemM* configuration is able to remain performant even when the number of frequently accessed pages surpasses DRAM size by 2x. However, in the 240GB parametrization, *MemM* performance degrades by an average of 46% in all scenarios, in which the number of frequently requested pages is about 4x larger than the DRAM cache's size.

For the $\sigma = 25\%$ workloads, increasing size always degrades performance, as all pages of the array are frequently accessed (within 2σ).

The hardware-based caching approach suffers in larger workload sizes, most notably in the lower locality $\sigma = 25\%$ workloads, in which its frequent cache replacements cause AAL to degrade past the static *ADM-default* configuration. Compared to *autonuma* and *ambix*, the configuration is only effective in the write-intensive $\sigma = 12.5\%$ 128GB workload, where its more aggressive caching strategy, which always decides to cache an accessed page, proves best among all configurations.

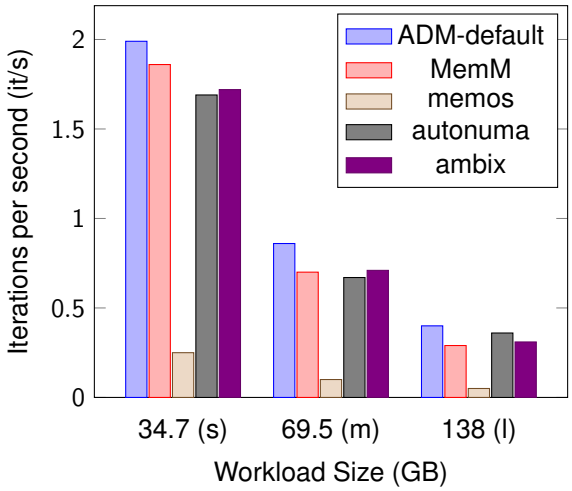


Figure 5.2: GAP BFS (2R:1W) plots. Varying workload size, Single socket, 32 threads

5.5.2 GAP – Sequential Pattern Performance

In Figure 5.2 we present the plots of a small, medium, and large parametrizations of GAP BFS, where the y axis represents throughput in iterations per second (it/s). All three parametrizations surpass DRAM size, and therefore a subset of pages must be placed in DCPMM.

As aforementioned, the BFS algorithm iterates over a generated graph of varying size, where accesses shift sequentially through the array, with no locality. As such, DCPMM-resident pages that a dynamic algorithm decides to promote do not compensate their migration cost in future accesses. Moreover, if DRAM is full and causes the algorithm to find cold pages to demote, these pages will usually be accessed sooner in the future than the promoted pages. Therefore, this study compares the dynamic solutions' capability of reducing the total number of unnecessary migrations performed, i.e., thrashing, where *ADM-default*, among the considered solutions, is expected to achieve the best throughput.

In the small parametrization, the *MemM* is best out of all dynamic configurations, with a throughput 6.8% lower than *ADM-default*, but falls off in the medium and large, where *ambix* (17% lower) and *autonuma* (10% lower) are best, respectively. On average, the *MemM*, *autonuma* and *ambix* configurations decrease throughput similarly, by 17.8%, 16%, and 17.6%, respectively. The *memos* configuration, on the other hand, lowers throughput by 87.5% in all three parametrizations.

Overall, the *autonuma* configuration provides the best sequential pattern performance out of all dynamic solutions, with *ambix* following closely behind by a 1.6% margin.

The AutoNUMA patch mitigates thrashing by measuring the time between a PTE's protected bit clear and subsequent access, relying on the mechanism existing in the base AutoNUMA algorithm. In contrast, Ambix achieves thrashing mitigation by clearing the PTE's R/D bits in DCPMM, shortly before selecting which pages to promote, therefore choosing to only promote pages accessed since they had their bits cleared. Ambix also enables programmers to proactively disable its `Switch` component, when expecting to bind an application with low locality. In this scenario (not shown in Figure 5.2), the average throughput in all three parametrizations drops by 2%, as opposed to the original 17.6%. We attribute this throughput drop to the buffer of free DRAM space Ambix decides to keep, regardless of the use of the

Benchmark	Parametrization	DRAM Hits %			DRAM Writes %		
		ADM-default	*autonuma*	*ambix*	*ADM-default*	*autonuma*	*ambix*
BT	small	100%	53%	100%	100%	35%	100%
	medium	77%	42%	95%	55%	16%	95%
	large	51%	36%	83%	26%	14%	92%
FT	small	100%	100%	100%	100%	100%	100%
	medium	29%	78%	93%	24%	79%	94%
	large	7%	82%	88%	5%	83%	89%
MG	small	100%	100%	98%	100%	100%	99%
	medium	45%	44%	48%	43%	42%	49%
	large	29%	28%	31%	28%	28%	30%
CG	small	100%	100%	100%	100%	100%	100%
	medium	28%	72%	90%	95%	96%	91%
	large	2%	54%	77%	31%	37%	50%
Average	small	100%	88%	100%	100%	84%	100%
	medium	45%	59%	82%	54%	58%	82%
	large	22%	50%	70%	23%	41%	65%

Table 5.2: NPB Benchmark's DRAM total and write Hit Rate comparison.

`Switch` component. Similarly to what was observed in pmbench, Memos' initial allocation in DCPMM and bandwidth-aware migration proves suboptimal in GAP BFS, with the vast majority of accesses being performed to DCPMM during the workload.

5.5.3 NPB – Dynamic Configuration's Performance in HPC applications

We consider that the NPB benchmarks depict common use cases for a dynamic page placement algorithm running in a HPC system populated with a DRAM-DCPMM HMA. Thus, we will evaluate all aspects of each dynamic configuration, comparing their performance in relation to *ADM-default*.

We use the small parametrizations to evaluate overhead, as *ADM-default* places every page in DRAM, which enables it to achieve the best throughput in all scenarios. In the larger parametrizations, similarly to pmbench, the *ADM-default* configuration sets the baseline for improvement.

Figure 5.3 presents four graphs which contain the average throughput of the BT (fig. 5.3a), FT (fig. 5.3b), MG (fig. 5.3c), and CG (fig. 5.3d) benchmarks, each with three different parametrizations, ordered by footprint. Furthermore, we introduce a fifth graph, which shows the geometric mean of all four chosen benchmarks, called NPBAVG (Figure 5.3e). We choose to represent the average as a geometric mean due to the large throughput difference between the tested benchmarks, which would consequentially give more relevance to read- or bandwidth-intensive workloads. In all graphs, the y axis represents throughput in million operations per second (M Ops/s).

Table 5.2 contains the percentage of total memory accesses fulfilled by the DRAM tier (DRAM Hits %), and from specifically write accesses (DRAM Writes %). The data in this table was generated in a separate run with `PCMon`, which we set to measure read and write operations to each node. Note that the collected values also include memory accesses due to page migration operations. The presented data shows how effective the *autonuma* and *ambix* configurations are at minimizing all and especially write accesses to DCPMM. We decide not to include *memos* into the table as it consistently performs

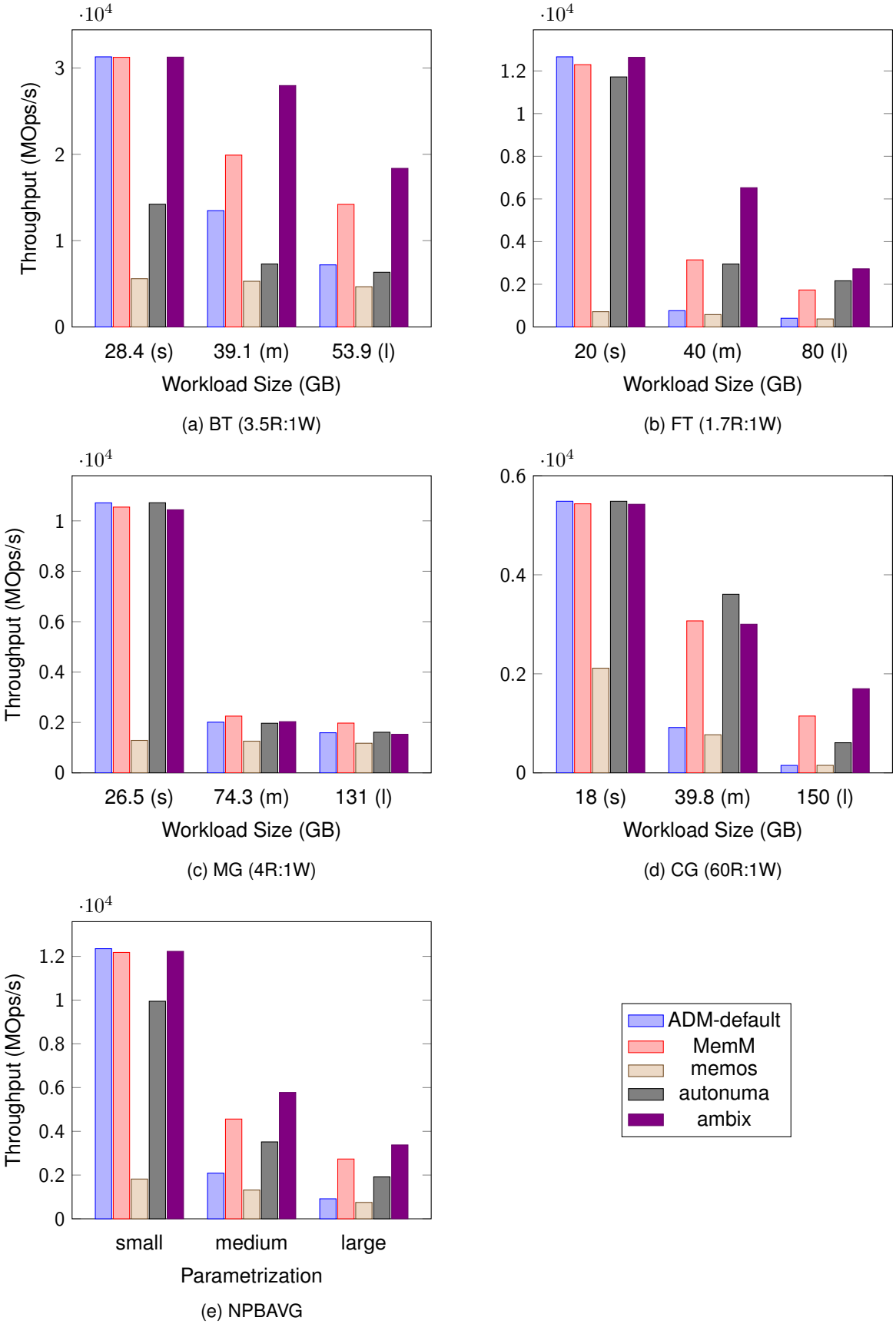


Figure 5.3: NPB plots. Varying workload size, Single socket, 32 threads

worse than the *ADM-default* configuration. Similarly, we also exclude *MemM* from the table.

Small Workloads

In the small data sets, *autonuma* has an identical performance to *ADM-default* in the FT, MG, and CG workloads, but presents a 60% lower throughput in the BT workload.

We attribute this larger drop to the combination of two factors. The AutoNUMA patch relies on a non-zero *swappinness* value, as it is designed to extend the *kswapd* daemon in order to demote pages to DCPMM. However, it does not make changes to how the page selection mechanism is implemented, and thus fails to consider asymmetry when choosing which pages to demote. This means that the algorithm may end up choosing write-intensive pages to migrate to DCPMM. When this happens: (i) the demoted page is written to DCPMM; (ii) it may quickly trigger the patch's fault-based promotion mechanism, which is more relaxed when the page fault is caused by a write access. This is further supported by the DRAM write % table, in which 65% of all write accesses are performed by DCPMM in the *autonuma* configuration.

In contrast, the migration mechanism of Ambix not only keeps a smaller 5% buffer in the faster tier, but also chooses to never demote a page it considers write-intensive even if it fails to find a suitable amount of cold ones. With this mechanism, Ambix is able to fully mitigate writes and virtually every access to the DCPMM tier in BT, as well as in all other small parametrizations.

On average, *MemM* performs better than *autonuma* and identically to *ambix*, in the small workloads (Figure 5.3e). Compared to *ADM-default*, *MemM* has an average 1.4% throughput penalty in all small workloads, reaching up to a 3% drop in the write-intensive FT workload.

Although the hardware-based caching algorithm starts by placing every page in DCPMM, the full working set is eventually cached in DRAM at first access, and therefore its performance only suffers from the initial placement policy. Therefore, since the initialization phase throughput is also measured in NPB's report, but dominated by the workload phase, we find the 1.4% overhead expected.

Similarly to the previous benchmarks, *memos* outputs the worst possible throughput in the small parametrizations, with an average 82% lower throughput, compared to *ADM-default*, being best in the very read-intensive CG benchmark, with still a large 62% drop.

Medium and Large Parametrizations

Compared to *ADM-default*, in the medium data sets, the *MemM*, *autonuma*, and *ambix* configurations have a speedup of 1.5x, 1.3x, and 2.7x on average, while in the large parametrizations the speedup increases to 2.8x, 1.8x, and 4.4x, respectively.

Despite the fact that Ambix performed no better than the AutoNUMA patch in the high locality pmbench workloads past 128GB, we observe that its placement mechanisms improve both throughput and DRAM hit rate more than the AutoNUMA patch in the larger parametrizations, despite only changing a single line of code in the kernel, and processing most of the placement decisions in a user-level process. In all medium and large workloads, *ambix* has an average speedup of 3.6x, compared to *autonuma*'s

1.6x. The speedup is further supported by *ambix*'s improved DRAM hit rate (76% vs. 55%), and write access hit rate (74% vs. 50%). Its benefit is most noticeable in the BT workloads, where *autonuma* fails to improve the *ADM-default* configuration, while *ambix* has an average 1.25x speedup. In the read-intensive CG workload, *autonuma* grants better performance than *ambix* in the medium parametrization, but falls off in the large one, with a 3x vs. 10x speedup compared to *ADM-default*.

While in the smaller parametrization, *MemM* is the third best non-static solution, it performs better than *autonuma* in the majority of medium and large workloads. However, *ambix* still surpasses it, having a 25% higher average throughput, in the medium and large parametrizations.

5.5.4 Summary

Throughout the evaluation section we reached the following set of conclusions:

- Ambix outperforms Intel's AutoNUMA patch in the NPB benchmarks, consistently improving DRAM total and write hit rate, as well as throughput. However, we identify some minor advantages to the latter in terms of scalability, as well as sequential access performance in the larger GAP BFS data set.
- Memos is unable to improve throughput, even in data sets that fully fit in DRAM, showing that its bandwidth-aware promotion mechanism is inefficient at saturating DRAM throughput.
- The MemM configuration improves the AutoNUMA patch but performs worse than Ambix in the NPB benchmarks. The configuration shows the worst scalability values in all tested pmbench workloads, often having worse results Memos, which maintains almost all pages in DCPMM.

Chapter 6

Final Considerations

In this chapter, we discuss the conclusions of this work, and reflect on the future work that could lead to further improvements to not only Ambix, but all HMA-aware placement solutions.

6.1 Conclusions

In this work, a dynamic page placement algorithm for DCPMM-equipped systems was proposed and evaluated against other relevant configurations, including two other proposed dynamic algorithms for our architecture and the DCPMM's hardware-based caching implementation. Our evaluation included benchmarks which test scalability, performance in sequential access patterns, and expected potential in HPC applications, all of which tested common scenarios where a DRAM-DCPMM system could be beneficial due to its increased memory capacity.

We described Ambix in detail, specifying critical implementation decisions in its design. These decisions were justified from two created benchmarks (IWB and PB), which tested the latency and energy impact of different page distributions and placement policies. Our discussion of these benchmarks not only helped justify Ambix's implementation, but also provides a relevant base for future work.

Despite the fact that DCPMM-equipped systems are still uncommon in servers and supercomputers, we estimate that the technology will become mainstream in the near future, due to the well studied memory scalability problem, which states memory capacity as one of the primary bottlenecks in current systems. This will broaden our work's scope to all scenarios that are currently DRAM-constrained but need to remain performant when running memory intensive workloads, in which scenario simply integrating DCPMM into the system is not sufficient. Ambix was able to improve throughput in the majority of workloads, when compared to DCPMM's MemM configuration, while leveraging existing mechanisms in commodity Linux kernels and being implemented with low footprint within the kernel.

Therefore, one of the major takeaways in our work is that a performant HMA system can be achieved without extensive kernel changes, and that DCPMM can replace some DRAM DIMMs, while still remaining performant.

Overall, we find our work's contributions to be useful to both the HPC and NVM-research fields, as

we find that our solution can remain relevant not only in future NVM technologies and memory management hardware improvements but also in emerging HPC workloads. We also devised our solution in a way that is easily extensible to consider future improvements in NVM technology but also a possible integration with parallel placement algorithms, which could be applied to multi-socket or a multiple machine scenarios.

6.2 Future Work

Although we have extensively tested Ambix in our system with benchmarks which test common HPC-related access patterns, we have yet to evaluate how our solution would perform with real HPC workloads when configured on dedicated hardware.

Before this can be accomplished, we find that we need to target some of Ambix's inefficiencies, namely its:

- **Migration throughput**, by transitioning from a syscall-based migration to a more efficient solution, such as the direct memory access (DMA) migration technique proposed in Nimble [95].
- **Netlink-based communication protocol**, which currently allows Ambix's user- and kernel-level to communicate should be replaced with a mechanism that allows more pages to be sent, as a consequence of the increased migration throughput.
- **Multi-socket agnosticism**. Ambix could be extended with the locality- and/or contention-based mechanisms discussed in prior NUMA-aware literature, considering new metrics such as access locality and load imbalance. Alternatively, we find that Ambix could also be integrated into an existing NUMA-aware solution, managing placement within each socket.

On a concrete note, we will continue working on Ambix, combining it with a static solution currently being developed in parallel with this work by the Barcelona Supercomputing Center, on the scope of the EPEEC project. We expect that this integration will improve both solutions, combining: (i) the static solution's in-depth knowledge about which objects cause the highest number LLC misses, using the information to steer initial placement; and (ii) Ambix's high-frequency migration component, which will enable the solution to react to access pattern changes, by temporarily or permanently deviating pages from the node where they were originally allocated.

Bibliography

[1] P. Balaprakash, D. Buntinas, A. Chan, A. Guha, R. Gupta, S. H. K. Narayanan, A. A. Chien, P. Hovland, and B. Norris. Exascale workload characterization and architecture implications. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 120–121. IEEE, 2013.

[2] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, et al. Scaling the power wall: a path to exascale. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2014.

[3] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[4] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Phase change memory architecture and the quest for scalability. *Communications of the ACM*, 53(7):99–106, 2010.

[5] A. Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25 – 38, 2016. ISSN 0038-1101. doi: https://doi.org/10.1016/j.sse.2016.07.006. URL `http://www.sciencedirect.com/science/article/pii/S0038110116300867`. Extended papers selected from ESSDERC 2015.

[6] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[7] S. Lai. Current status of the phase change memory and its future. In *IEEE International Electron Devices Meeting 2003*, pages 10–1. IEEE, 2003.

[8] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 24–33. ACM, 2009.

[9] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[10] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, et al. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, 28(2):223–262, 2010.

[11] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2013.

[12] K. Wang, J. Alzate, and P. K. Amiri. Low-power non-volatile spintronic memory: Stt-ram and beyond. *Journal of Physics D: Applied Physics*, 46(7):074003, 2013.

[13] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy reduction for stt-ram using early write termination. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 264–268. ACM, 2009.

[14] J. Müller, T. Böscke, S. Müller, E. Yurchuk, P. Polakowski, J. Paul, D. Martin, T. Schenk, K. Khullar, A. Kersch, et al. Ferroelectric hafnium oxide: A cmos-compatible and highly scalable approach to future ferroelectric memories. In *2013 IEEE International Electron Devices Meeting*, pages 10–8. IEEE, 2013.

[15] S. Dünkel, M. Trentzsch, R. Richter, P. Moll, C. Fuchs, O. Gehring, M. Majer, S. Wittek, B. Müller, T. Melde, et al. A fefet based super-low-power ultra-fast embedded nvm technology for 22nm fdsoi and beyond. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 19–7. IEEE, 2017.

[16] H. Akinaga and H. Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[17] I. B. Peng, M. B. Gokhale, and E. W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[18] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19, 2019.

[19] Intel. Intel Aurora Supercomputer. `https://www.intel.com.au/content/www/au/en/high-performance-computing/supercomputing/aurora-video.html`. Accessed:2021-04-11.

[20] Y. Huang. autonuma: Optimize memory placement in memory tiering system. `https://lwn.net/Articles/803663/`, 2020. [Online; accessed 19-November-2020].

[21] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks summary and

preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165. IEEE, 1991.

[22] NASA. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html, 2021. [Online; accessed 20-May-2021].

[23] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[24] S. Beamer, K. Asanović, and D. Patterson. GAP Benchmark Suite. http://gap.cs.berkeley.edu/benchmark.html, 2021. [Online; accessed 20-May-2021].

[25] S. Lee, H. Bahn, and S. H. Noh. Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures. *IEEE Transactions on Computers*, 63(9):2187–2200, Sep. 2014. doi: 10.1109/TC.2013.98.

[26] M. Lee, D. H. Kang, J. Kim, and Y. I. Eom. M-clock: Migration-optimized page replacement algorithm for hybrid dram and pcm memory architecture. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 2001–2006, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3196-8. doi: 10.1145/2695664.2695675. URL http://doi.acm.org/10.1145/2695664.2695675.

[27] H. Seok, Y. Park, and K. H. Park. Migration based page caching algorithm for a hybrid main memory of dram and pram. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 595–599, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0113-8. doi: 10.1145/1982185.1982312. URL http://doi.acm.org/10.1145/1982185.1982312.

[28] H. Seok, Y. Park, K.-W. Park, and K. H. Park. Efficient page caching algorithm with prediction and migration for a hybrid main memory. *SIGAPP Appl. Comput. Rev.*, 11(4):38–48, Dec. 2011. ISSN 1559-6915. doi: 10.1145/2107756.2107760. URL http://doi.acm.org/10.1145/2107756.2107760.

[29] Z. Sun, Z. Jia, X. Cai, Z. Zhang, and L. Ju. Aimr: An adaptive page management policy for hybrid memory architecture with nvm and dram. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 284–289. IEEE, 2015.

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

[31] Z. Zhang, Y. Fu, and G. Hu. Dualstack: A high efficient dynamic page scheduling scheme in hybrid main memory. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6. IEEE, 2017.

[32] S. Kim, S.-H. Hwang, and J. W. Kwak. Adaptive-classification clock: Page replacement policy based on read/write access pattern for hybrid dram and pcm main memory. *Microprocessors and Microsystems*, 57:65–75, 2018.

[33] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 521–534, 2017.

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

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

[36] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[37] L. Liu, S. Yang, L. Peng, and X. Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019.

[38] S. Yu, S. Park, and W. Baek. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

[39] R. Salkhordeh and A. Brinkmann. Online management of hybrid dram-nvmm memory for hpc. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 277–289. IEEE, 2019.

[40] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136, Sep. 2017. doi: 10.1109/CLUSTER.2017.50.

[41] K. Wu, Y. Huang, and D. Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[42] M. B. Olson, T. Zhou, M. R. Jantz, K. A. Doshi, M. G. Lopez, and O. Hernandez. Membrain: Automated application guidance for hybrid memory systems. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE, 2018.

[43] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones. Portable application guidance for complex memory systems. In *Proceedings of the International Symposium on Memory Systems*, pages 156–166, 2019.

[44] T. C. Effler, A. P. Howard, T. Zhou, M. R. Jantz, K. A. Doshi, and P. A. Kulkarni. On automated feedback-driven data placement in multi-tiered memory. In *International Conference on Architecture of Computing Systems*, pages 181–194. Springer, 2018.

[45] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[46] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang. Object-level memory allocation and migration in hybrid memory systems. *IEEE Transactions on Computers*, 69(9):1401–1413, 2020.

[47] L. Zhang, R. Karimi, I. Ahmad, and Y. Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–27, 2020.

[48] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104, Mar. 2011. ISSN 0362-1340. doi: 10.1145/2248487.1950379. URL `http://doi.acm.org/10.1145/2248487.1950379`.

[49] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 46(3):105–118, Mar. 2011. ISSN 0362-1340. doi: 10.1145/1961296.1950380. URL `http://doi.acm.org/10.1145/1961296.1950380`.

[50] R. Bharadwaj. *Mastering Linux Kernel Development: A kernel developer's reference manual*. Packt Publishing Ltd, 2017.

[51] M. Gorman. Page Table Management. `https://www.kernel.org/doc/gorman/html/understand/understand006.html`. [Online; accessed 19-December-2020].

[52] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[53] E. J. O'neil, P. E. O'neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[54] P. B. Galvin, G. Gagne, A. Silberschatz, et al. *Operating system concepts*, pages 367–369. John Wiley & Sons, 9th edition, 2012.

[55] F. J. Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[56] P. B. Galvin, G. Gagne, A. Silberschatz, et al. *Operating system concepts*, pages 436–440, 795–803. John Wiley & Sons, 10th edition, 2018.

[57] L. Torvalds. mm/vmscan.c. `https://elixir.bootlin.com/linux/latest/source/mm/vmscan.c`, 2021. [Online; accessed 21-November-2020].

[58] corbet. A CLOCK-Pro page replacement implementation. https://lwn.net/Articles/147879/, 2005. Accessed:2019-12-30.

[59] S. Jiang, F. Chen, and X. Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 35–35, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247395.

[60] T. Sterling, M. Anderson, and M. Brodowicz. Chapter 2 - hpc architecture 1: Systems and technologies. In *High Performance Computing*, pages 43 – 82. Morgan Kaufmann, Boston, 2018. ISBN 978-0-12-420158-3. doi: https://doi.org/10.1016/B978-0-12-420158-3.00002-2. URL http://www.sciencedirect.com/science/article/pii/B9780124201583000022.

[61] P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 80–91, 1992.

[62] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.

[63] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.

[64] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on {NUMA} systems: Asymmetry matters. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 277–289, 2015.

[65] V. Viswanathan, K. Kumar, T. Willhalm, P. Lu, B. Filipiak, and S. Sakthivelu. Intel memory latency checker. *Intel Corporation*, 2013.

[66] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. *ACM SIGOPS Operating Systems Review*, 30(5):279–289, 1996.

[67] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

[68] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. T. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[69] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov. Bandwidth-aware page placement in numa. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 546–556, Los Alamitos, CA, USA,

may 2020. IEEE Computer Society. doi: 10.1109/IPDPS47924.2020.00063. URL https://doi.ieeecomputersociety.org/10.1109/IPDPS47924.2020.00063.

[70] A. Kleen. A numa api for linux. *Novel Inc*, 2005.

[71] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[72] J. Corbet. Autonuma: the other approach to numa scheduling. *LWN. net*, 2012.

[73] S. Blagodurov, A. Fedorova, S. Zhuravlev, and A. Kamali. A case for numa-aware contention management on multicore systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 557–558. IEEE, 2010.

[74] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, pages 11–20, 2011.

[75] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing google's warehouse scale computers: The numa experience. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–197. IEEE, 2013.

[76] J. M. Bull and C. Johnson. Data distribution, migration and replication on a cc-numa architecture. In *Proceedings of the fourth European workshop on OpenMP*, 2002.

[77] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth. Challenges of memory management on modern numa systems. *Communications of the ACM*, 58 (12):59–66, 2015.

[78] Intel. An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing. https://software.intel.com/content/www/us/en/develop/blogs/an-intro-to-mcdram-high-bandwidth-memory-on-knights-landing.html, 2016. Accessed:2021-05-13.

[79] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[80] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[81] Intel. Intel 64 and ia-32 architectures software developer's manual. *System Programming Guide*, 3B, September 2016.

[82] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007.

[83] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45(5):758–767, 1997.

[84] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *ACM SIGARCH Computer Architecture News*, 42(2):20–23, 2014.

[85] S. Jiang and X. Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1): 31–42, 2002.

[86] T. Brown, T. Liao, and J. Chou. Analyzing the performance of intel optane dc persistent memory in app direct mode in lenovo thinksystem servers. *Lenovo Press*, 2019. URL `https://lenovopress.com/lp1083.pdf`.

[87] J. Chou, T. Brown, and T. Liao. Analyzing the performance of intel optane dc persistent memory in memory mode in lenovo thinksystem servers. *Lenovo Press*, 2019. URL `https://lenovopress.com/lp1084.pdf`.

[88] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[89] K. Wu, F. Ober, S. Hamlin, and D. Li. Early evaluation of intel optane non-volatile memory with hpc i/o workloads. *arXiv preprint arXiv:1708.02199*, 2017.

[90] OPCM. Processor Counter Monitor. `https://github.com/opcm/pcm`, 2016. [Online; accessed 25-November-2020].

[91] J. Yang and J. Seymour. Pmbench: A micro-benchmark for profiling paging performance on a system with low-latency ssds. In *Information Technology-New Generations*, pages 627–633. Springer, 2018.

[92] J. Yang and J. Seymour. pmbench. `https://bitbucket.org/jisooy/pmbench`, 2021. [Online; accessed 20-May-2021].

[93] Y. Huang and V. Verma. AutoNUMA: tiering-0.4. `https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git?h=tiering-0.4`, 2020. [Online; accessed 19-November-2020].

[94] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[95] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.