TÉCNICO
LISBOA

Figures/Airbus_A350.jpg

Smart-malloc: Rethinking Linux's Page Management to Support the new Intel Optane Persistent Memory

Miguel Soares Marques

Thesis to obtain the Master of Science Degree in

Information and Software Engineering

Supervisor(s): Dr. João Pedro Barreto
Dr. Rodrigo Miragaia Rodrigues

Examination Committee

Chairperson: Prof. Full Name
Supervisor: Prof. Full Name 1 (or 2)
Member of the Committee: Prof. Full Name 3

February 2021

Dedicated to someone special...

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: palavra-chave1, palavra-chave2,...

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: keyword1, keyword2,...

Contents

Acknowledgments . . . v
Resumo . . . vii
Abstract . . . ix
List of Tables . . . xiii
List of Figures . . . xv
Nomenclature . . . xvii
Glossary . . . 1

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Objectives . . . 3
1.3 Thesis Outline . . . 3

2 Background **5**
2.1 Memory Management . . . 6
2.1.1 Locality of Reference Principles . . . 7
2.1.2 Page Table . . . 7
2.1.3 Page Replacement Algorithms . . . 7
2.1.4 Virtual Memory Management in Linux . . . 11
2.2 NUMA-Aware Data Placement . . . 12
2.2.1 NUMA Architecture . . . 12
2.2.2 NUMA Evolution . . . 13
2.2.3 Linux NUMA Subsystem . . . 14
2.2.4 Contention-Aware Approaches . . . 16
2.3 Data Placement in HMAs . . . 18
2.3.1 Optane Configuration . . . 19
2.3.2 Static Placement . . . 20
2.3.3 Dynamic Placement . . . 21
2.3.4 NUMA-Aware Solutions Compatibility . . . 24

3 Implementation **25**
3.1 Theoretical Overview . . . 25

3.2 PnP in Practice 26
3.3 Goals 26
3.4 Goal Implementation 26

4 Results **27**
4.1 Problem Description 27
4.2 Baseline Solution 27
4.3 Enhanced Solution 27
4.3.1 Figures 27
4.3.2 Equations 29
4.3.3 Tables 29
4.3.4 Mixing 30

5 Conclusions **33**
5.1 Achievements 33
5.2 Future Work 33

Bibliography **35**

List of Tables

4.1 Table caption shown in TOC. 30
4.2 Memory usage comparison (in MB). 30
4.3 Another table caption. 30
4.4 Yet another table caption. 30
4.5 Very wide table. 30

List of Figures

2.1 NUMA Traffic imbalance 15
2.2 NUMA Interlink Imbalance 18

4.1 Caption for figure in TOC. 28
4.2 Some aircrafts. 28
4.3 Schematic of some algorithm. 28
4.4 Figure and table side-by-side. 31

Nomenclature

Greek symbols

α Angle of attack.

β Angle of side-slip.

κ Thermal conductivity coefficient.

μ Molecular viscosity coefficient.

ρ Density.

Roman symbols

C_D Coefficient of drag.

C_L Coefficient of lift.

C_M Coefficient of moment.

p Pressure.

$\mathbf{u}$ Velocity vector.

u, v, w Velocity Cartesian components.

Subscripts

∞ Free-stream condition.

i, j, k Computational indexes.

n Normal component.

x, y, z Cartesian components.

ref Reference condition.

Superscripts

* Adjoint.

T Transpose.

Chapter 1

Introduction

Memory scalability is an ever more relevant concern in the era of Exascale computing. As CPU speeds increase, traditional memory technologies have fallen behind, leaving an increasing gap between processor and memory performance. 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, to cooling, area constraints and even cost [1].

Non-volatile memory (NVM[1]) is an emerging class of memory that is able to mitigate these issues [2]. 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 [3]. Most notably, phase-change memory (PCM) [4–8], spin-transfer torque RAM (STT-RAM) [9–11], ferroelectric field-effect transistor (FeFET) [12, 13] and resistive random-access memory (RRAM) [14] have all been previously studied as possible candidates for replacing or complementing DRAM, in novel hybrid memory architectures (HMAs[2]).

Intel started research on a new persistent technology named 3D XPoint in 2012, a 2-layer implementation of PCM. This technology, under the brand name Optane, improves on PCM by performing

[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'll use these terms interchangeably throughout the dissertation, although we mainly use NVM when referring to persistent memory.

[2] We use the term HMA to refer to architectures that use a multi-tiered memory configuration, consisting of distinct memory types.

faster and enduring a higher number of writes, one of the main limiting factors of PCM. Intel Optane DC persistent memories range from 128GB, the current maximum DRAM stick size, to 512GB, while maintaining the same DIMM form factor. Optane's[3] 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.

1.1 Motivation

Optane, likewise previous NVM technologies, can be configured to extend main memory. In this configuration, applications are able to run larger data sets without needing to swap or perform I/O operations to storage due to main memory limitations. However, due to NVM's latency and bandwidth limitations, allocating data to the appropriate memory tier is most relevant, as placing intensive and especially write-intensive data in NVM can severely hinder an application's performance.

Intel provides a novel mode of operation that configures DRAM as a cache, and Optane as the sole memory tier. In this configuration, the most intensive data is cached in DRAM in a way seamless to the programmer. However, Optane also allows a more traditional configuration, where the system has two separate memory tiers, for DRAM and Optane. This 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 HMA, 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, access patterns and memory characteristics.

The improved characteristics of Optane compared to previous NVM technologies, as well as its novel configuration capabilities, spiked an interest in the data placement area. However, due to Optane's recentness, there is no published real-world implementation of a data placement algorithm that considers an Optane-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 being tested on a real machine, running Optane [15]. 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 Optane's characteristics in mind, which leverages the increased memory scalability offered by Optane to improve relevant large data set workloads, tested on a real NVM-equipped commodity system.

On the other hand, 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 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

[3]Despite the fact that the Intel Optane DC lineup includes products such as block-based 3D XPoint-accelerated PCIe SSDs, we simply use Optane from now on to refer to the byte-addressable Optane DC persistent memory.

computing node.

Nevertheless, previous NUMA-aware solutions, including Linux's implementation, are unsuitable for Optane 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 replace the existing Linux's page management mechanism with a solution that considers an heterogeneous memory configuration, consisting of both DRAM and Optane. 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 Optane.

Our solution leverages: (i) existing page walking mechanisms, (ii) the dirty and reference bits, managed by the memory management unit (MMU), and (iii) Intel's Performance Counter Monitor (PCM), which allows us to determine the bandwidth usage of each of the memories with hardware counters available in most modern Intel CPUs.

We show that we are able to determine the optimal page distribution dynamically, with low overhead. Our solution is tested with the latest relevant bandwidth and memory-intensive workloads in the high performance computing (HPC) field.

Our work will also extend the current NVM data placement-related research with the first published solution that provides insight on how a data placement algorithm performs on a real machine running Optane, which separates it from the available trace or simulation-driven evaluations available in past literature.

1.3 Thesis Outline

The remainder of this work is organised as follows:

- **Chapter 2** provides an introduction to the memory management field, data placement in NUMA and HMA architectures and algorithms and current.

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 capitalise 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 a cache to NVM. 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, the application has 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 HMA architecture is divided in two major areas. The first area studies how systems can integrate a slower but larger level of memory for data placement. This involves rethinking traditional memory management strategies to consider an heterogeneous memory configuration. The second area leverages NVM's non-volatility in order to implement faster durability techniques for data objects. Current literature in this area adapts traditional checkpointing algorithms

and transactional systems to consider the lower access latencies offered by NVM, when compared to storage.

Our work is focused on the first area. We will start by describing how Linux and other OSs manage 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 the unsuitability of previous work, along with an historical evolution of the memory management mechanisms provided by Linux, which we ultimately found to be still lacking in both the traditional NUMA and hybrid memory architectures, compared to the SotA in these areas.

2.1 Memory Management

Memory management is a critical part of an OS' design. Main memory is a scarce resource, and without memory management mechanisms processes would both need to compete for the available physical memory, as well as be aware of where they could write data in main memory, since they could inadvertently or maliciously write over other process' data.

Linux divides a process' data in same-size blocks, usually 4KB in size, called pages. It attributes each process its own virtual address space, where they can allocate pages, even if they exceed physical memory capacity by multiple times. Virtual addressing allows the process to perceive available memory as a large, contiguous structure, even though in reality its pages might physically reside in different memories or in storage. Linux implements demand paging. In demand paging, the process' pages are initially in storage. When the process tries to access a page, the CPU tries to find it 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 new page information.

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.

Since processes perceive available memory as a value much higher than the existing physical memory capacity, it can become full. 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. Higher values of *swappinness* correlate to more aggressive swapping, meaning that the kernel swaps pages more eagerly. In contrast, a *swappinness* of 0 makes the kernel only swap pages when needed, i.e., when there is no available space. 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 *swappiness* 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 has no sufficient space, then a page that the kernel expects to not be accessed soon is reclaimed to make space for the new one, which introduces a computational overhead compared to directly allocating the page when there is free space.

Placing the newly accessed page in memory is important because processes tend to access the same memory locations frequently over a short period of time, a phenomenon called *temporal locality*. As such, pages that have not been accessed recently are prioritised for reclamation.

In anticipatory paging systems, a related principle applies, called *spatial locality*. *Spatial locality* dictates that a page accessed is usually followed by accesses to pages within the vicinity. This is visible in many algorithms which perform common operations, such as array initialisations and iterations, where a page access is followed by page accesses within a close virtual address range. As such, anticipatory paging mechanisms additionally place pages which address is close to the referenced one, as they are predicted to be accessed in the near future.

A similar locality-based principle, called *branch locality*, is observed in branching operations such as *for* and *while* cycles, which normally access distant addresses repeatedly over a short span of time.

2.1.2 Page Table

In OS' that implement virtual memory such as Linux, a page table is used to keep track of virtual to physical addresses mappings in page table entries (PTEs). Each PTE also stores bits that provide information on page protection, and a present, dirty and reference bit.

The present bit indicates if a page is currently in physical memory or in the swap cache. If the present bit is unset, the current information in the PTE might not be valid, and as such information regarding the page is ignored by the OS.

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. This indicates that storage might contain an older value for the indicated page, and as such that the CPU must write the new value to storage before deciding to evict the page from memory.

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 current Linux kernels, to decide which pages to evict.

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 optimise them. All algorithms described in this section are thought out for a traditional architecture, consisting of a DRAM memory and swap cache.

Bélády's Algorithm

We mention Bélády's algorithm [16], also known as OPT, as it provides the theoretical best page replacement policy.

When a page needs to be swapped out, the algorithm selects the page that would be accessed the farthest from the current point in time. It cannot be implemented in general purpose OSs, as the algorithm would require 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 promotes 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.

Optimised 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 algorithm 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 counters. Furthermore, the *age* entry should be stored in multiple bits, to avoid clock overflow, which increases the memory requirements of the algorithm.

A modification to base LRU, named LRU-K [17] 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. 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. The algorithm introduces additional tracking data, which leads to a greater memory and computational requirement. Adding that to the fact that it uses a logical clock implementation 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 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, it 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 oldest page is always selected for replacement, even if it was accessed more frequently than newer pages.

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 [18] 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 [19] and the latest Linux kernel [19, 20].

CLOCK-Pro

Rik van Riel proposed a kernel patch to use an optimised version of CLOCK to manage OS-level page placement [21], named CLOCK-Pro [22]. 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[22] 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 frequently for two major reasons:

- Incrementing an access counter for each page access is expensive to the CPU, while using the reference bit has been proven to provide a good estimate of the optimal page for replacement.
- 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 as such it is not frequently implemented in current OSs.

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

It randomly selects a page with both bits set, if no pages that meet the previous criteria were found. The algorithm prioritises pages with the reference bit 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 [19, 20]. Even though it is considered to be LRU-based, the algorithm provides multiple improvements over the computational requirements of the LRU implementations, by using the CLOCK algorithm to reduce the overhead of the mechanism.

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.

When a process accesses a page for the first time, it is placed in memory, and the reference bit is set. Additionally, the page is also introduced into the active list.

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.

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. The demoted pages are selected with the CLOCK algorithm. Moreover, when a page that was demoted to the inactive list is referenced again, it is promoted to the active list.

If the current free memory falls below a threshold, the kernel launches a daemon, called *kswapd*, which reclaims as many pages from the inactive list as necessary until it achieves a new balance, starting from the older ones. 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 reclaim pages from the active list that have not been recently accessed. In this scenario, CLOCK is also used, but pages are directly reclaimed, instead of demoted to the inactive list.

2.2 NUMA-Aware Data Placement

In this section, we describe the memory management mechanisms provided by Linux when a system has multiple sockets and physically distributed memory (NUMA). We will also present the state of the art in data placement for these architectures, as they can be correlated to mechanisms proposed for NVM-aware solutions.

2.2.1 NUMA Architecture

Shared-memory multiprocessor (SMM) systems are composed of multiple processors which share access to all memory in the system [23]. In these systems, main memory can be organised in two configurations: (i) centralised, 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).

In NUMA architectures, although main memory is physically distributed, CPUs can still access all memory in the system via interconnection links, albeit at an higher latency.

In these systems, Linux implements a logical node view, where a node is characterised by a CPU and the memory banks closest to it, usually configured as a node per socket. Communication between the sockets is achieved via NUMA interlinks, which allow data operations between nodes.

As the name indicates, memory access latency can be faster or slower, 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 it instead accesses another node's memory, it is called a remote access. Remote accesses typically incur an higher access latency, as the request is performed to a memory bank logically far from the requesting core, which is served over the interconnection links that connect the local CPU to the remote memory.

Linux defines this latency difference as *distance*, which is attributed between node pairs. *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. For a distance d, between 10 and 254, the access latency 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 hard-coded by default, and are provided by a firmware-dependant 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) [24] 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 some 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. When multiple accesses are performed in parallel by multiple CPU cores to addresses that belong to the same memory, the average access latency deteriorates. This is due to data congestion on the bus that connects the memory to the memory controller that processes the data flow in both directions.

Memory controllers are chips that are placed between the CPU and physical memory in a logical view. They control data flow in both directions, and provide virtual to physical address translation in OSs that implement virtual memory. The chip is also responsible for DRAM-specific operations such as DRAM refreshes, which prevent data loss in the volatile memory by charging its capacitors.

The memory controller chip (MCC) is responsible for a subset of the system's memory. While a CPU can be associated to multiple MCCs, each memory DIMM can only be associated to one MCC, via a memory channel.

Modern motherboard designs often implement multiple memory channels, or buses. Memory channels increase the maximum throughput of data between a CPU and its memory. Multi-channel architectures reduce the bandwidth stress, by distributing data evenly across different channels. However, even in these architectures, data can still be processed by the CPU much faster than it can be requested from memory.

In order to solve this problem, a simple solution would be to increase the amount of memory channels, leading to further improvements on the total available memory bandwidth. However, a more scalable solution is to schedule a process and its data to idle or unstressed remote nodes.

2.2.2 NUMA Evolution

NUMA-aware data placement mechanisms for commodity OSs, such as Linux, usually rely on minimising 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 maximise the local access ratio of the running threads. In scenarios where multiple large footprint applications run in parallel, these systems 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 difference between local and remote accesses in older systems. These systems benefit of maximising the ratio of local accesses for all applications, as they can fulfil more data requests per unit of time, even if the utilisation 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 [25], modern NUMA systems reduce the penalty to less than 30% [26, 27].

The faster remote access latencies lead to a shift in focus in the optimal data placement strategy. Due to the reduced latency overhead, other metrics, which were trivial in comparison, 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 MCC load imbalance, interconnect link usage, and other bandwidth-related metrics, which have been shown to deteriorate the throughput of an application.

Contention occurs when multiple threads share data and try to write to it simultaneously. Previous literature defines contention in the memory channels and their chips as one of the major bottlenecks in performance [26, 28]. As such, solutions that are capable of managing contention 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**. Each node has its respective swapper thread, which manages the active and inactive lists and swaps out pages when needed. A global swapper thread runs less frequently, when the system runs out of space across all its nodes.

NUMA Memory Policy

By default, the global memory policy Linux follows for running applications is *node local*. The *node local* policy prioritises the allocation of a new page to the node where the computation runs, as it provides the lowest latency and highest bandwidth values in ideal conditions. The policy favours local accesses, which reduces the stress on the interconnection link. If there is no space available, Linux defaults to allocating pages on the least-distant node with available memory.

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 [29].

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. 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. Nevertheless, interleaving pages can outperform the *node local* policy in scenarios where allocating all data to the local node causes significant bus contention, in which case, the added throughput of the unstressed remote nodes can prove beneficial to the performance of the workload [26]. However, this only solves the contention problem partially, as the contention-unaware allocation can aggregate write-intensive pages in the same node.
- *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 either *node local* or *interleave* provide better performance in most if not all scenarios. Nevertheless, they can be useful for debug or benchmarking tools, as they allow testing the performance of different node configurations.

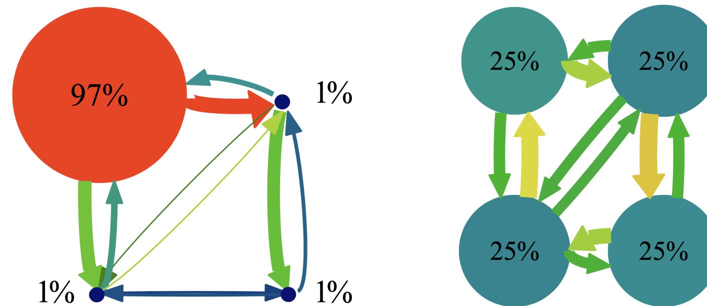


Figure 2.1: Traffic imbalance under *node local* (left) and *interleave* (right) policies for the PARSEC's [30] *streamcluster* benchmark [26].

Figure 2.1 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, visualised in percentage, are perfectly distributed across nodes. In this scenario, *interleave* outperforms the *node local* policy by a factor greater than 2 [26]. 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.

However, *node local* has been shown to perform better for other workloads, including the majority of the benchmarks in the PARSEC suite [26]. 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 [31]. It presents several optimisations 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 hardware view the page as resident but not accessible, therefore causing a 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 presents has 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 scheduled to migrate to that node. Although thread migration incurs significant CPU overhead, the algorithm 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 on a page, the accessed page is queued to migrate to that node. Periodically, a kernel routine migrates the queued pages. If another node accesses a queued page before the routine migrates it, then the page is cleared from the queue.

Enabling AutoNUMA provides significant performance benefits for multi-threaded workloads without the need for manual tuning. However, it performs worse when compared to contention-aware approaches for high-contention workloads. Nevertheless, we find it relevant for discussion, as an extension that integrates Optane was recently proposed [15].

2.2.4 Contention-Aware Approaches

In this section we describe in detail two approaches which extend locality-based principle with metrics that are related to bus contention.

Carrefour [26] and *AsymSched* [28] detail the importance of contention management in modern NUMA systems. *AsymSched* further improves contention-awareness by consider the NUMA interlinks' bandwidth asymmetry, which is common in modern systems.

Carrefour

Carrefour [26] 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 have a lower idle access latency. However, the algorithm's capability of detecting MCC imbalances allows it to prefer remote nodes in these scenarios for placing or migrating data. *Carrefour* also schedules threads in away that minimises 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 MCC imbalance, it distributes the pages across multiple nodes, in a way 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 MCC 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 MCC imbalance surpasses a defined threshold. If the MCC imbalance is low, then applications do not benefit from interleaving. However, interleaving can significantly improve performance in applications that would otherwise cause high MCC imbalance, as seen by the default *interleave* policy in Linux [32].

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 MCC imbalance.

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

AsymSched

AsymSched [28] 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.

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.

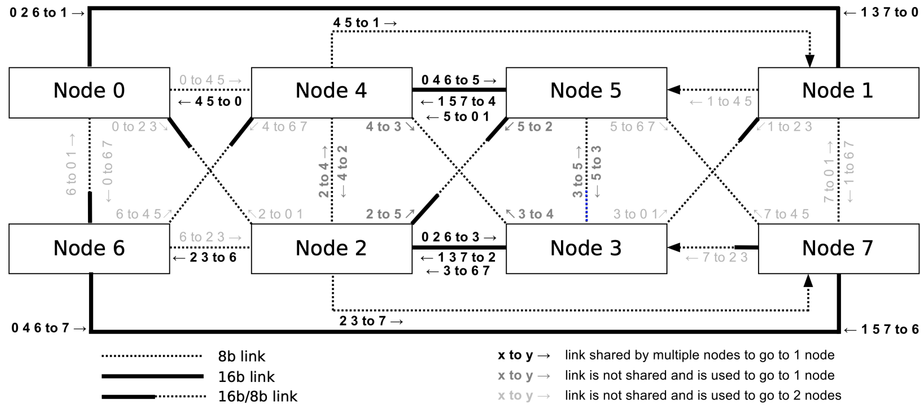


Figure 2.2: Asymmetric NUMA system [28].

The algorithm adapts to these asymmetric configurations 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 inter-links shared with other remote threads.

Likewise *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, defining where data should be placed is not trivial. DRAM and NVM provide different characteristics, where DRAM has a lower write access latency and NVM is denser, allowing more data to be placed in it.

Placing data in DRAM has obvious benefits, such as a reduced average memory access time (AMAT), and overall increased throughput, due to its higher bandwidth. This is especially true if data

is modified often, as NVM falls short on write latency. However, read-intensive and some less write-intensive data may be better suited to NVM, since DRAM space is scarce.

Placement algorithms need not only to decide the optimal memory tier for data based on access speed, but also to take into consideration NVM drawbacks not directly related to its latency, such as its higher active power draw when written to and its limited bandwidth. Thus, algorithms that consider HMAs, usually minimise as much as possible write operations to data that is placed in the NVM, by placing write-intensive data in DRAM [33–36].

In order to allocate data to the appropriate level, solutions may: (i) take advantage of the existing hardware performance counters, (ii) leverage code instrumentation, (iii) use access bits (reference/dirty) managed by the memory management unit (MMU) for each page, (iv) mark pages as read-only and intercept writes to them, (v) devise software-level structures that keep track of data accesses.

We divide data placement efforts in two areas: static placement, an approach that profiles data accesses and determines the optimal tier for data before executing a workload; and dynamic placement, an approach that decides data migration and eviction during execution.

While placement is most commonly decided at page-level [33–36], static object-level implementations exist [37]. Dynamic placement algorithms have been shown to incur minimal amounts of overhead in the overall execution of any workload, which further helped to justify our decision.

Page-level solutions benefit from consistently fine-grained allocation, that cannot be achieved by object-level approaches, when objects are large. For example, if the access frequency to a certain large data structure is uneven — i.e., one or more fields of the structure are accessed more frequently than others — placing the whole object in the faster tier may be inefficient. Moreover, managing placement at the page-level allows leveraging the MMU's access registers, which have been shown to be capable of providing accurate information on a page's access profile, with low overhead [33].

Before we start describing data placement algorithms for an NVM-equipped architecture, we will first describe how Optane can be configured, and its performance with different workload types in different physical configurations and modes. We also discuss how solutions designed for NUMA architectures can be extended to consider the novel HMA architecture, or combined into a holistic solution.

2.3.1 Optane Configuration

Optane can be configured in two modes: App Direct (ADM), and Memory (MM). In ADM, Optane can be seen as a natural extension of the address space, effectively increasing the system's available memory to the capacity of DRAM and Optane combined. This makes Optane especially relevant for large footprint workloads. These workloads currently run in multi-socket NUMA machines, or by leveraging the memory of multiple machines, due to memory scalability limitations. This brings a twofold benefit: it reduces the bandwidth stress on the limited NUMA interlink connections, as a larger subset of the workload can be placed in local memory; and improves data access latency by keeping data closer to the computation node, as remote accesses usually incur an higher latency compared to NVM. In this mode, Optane can also be configured as a directly-mapped storage medium, which can be used to store files and user

data, similarly to traditional storage. The latter configuration is referred to as "Storage over App Direct Mode". In MM, DRAM is configured as an extended cache, no different in functionality to the existing CPU caches and processes allocate pages directly to Optane, while the hardware manages which pages are cached in DRAM, seamless to the programmer. Furthermore, different combinations of the previous mentioned modes are possible.

In multi-channel systems, Optane and DRAM can populate the memory slots in different configurations, with varying performance results. Lenovo published two papers that analyse the performance of different DIMM configurations on a multi-socket and multi-channel server, each pertaining to a single Optane mode of operation: ADM [38], and MM [39].

Overall, the introduction of Optane configured in both app and memory mode greatly increases memory size at a varying latency and bandwidth penalty, as expected.

The App Direct mode study [38] provides a raw performance comparison between each of the memory types. In the study, we see an exponential increase in loaded latency for the Optane DIMMs, while DRAM DIMMs are able to remain performant under full load. This is relevant for data placement algorithms, as it is shown that Optane performs best in near idle conditions, which translates to it being most performant when used to store less intensive pages. Furthermore, Optane has a $\sim$2.5x higher bandwidth in read-only workloads compared to write-only workloads. We conclude that Optane can be used to store a subset of a workload's read-intensive pages while maintaining an acceptable performance level, as long as it is not under full load.

The Memory mode study [39] presents us with more interesting results. In this mode, DRAM caches the most intensive pages, which alleviates load on the Optane DIMMs. In the study, a 2-2-1 configuration consisting of 1 DRAM DIMM per channel, and an additional Optane DIMM configured in Memory mode in the first and second channels outperforms a 2-1-1 (3x DRAM ad 1x Optane) configuration in average access latency by as much as 40% in read-only workloads and 50% in "2 reads 1 write" (2R1W) workloads, while providing a full Optane DIMM of added capacity. Bandwidth results are similar, where the 2-2-1 and 2-2-2 configurations are able to sustain $\sim$50% bandwidth under full load scenarios, while 2-1-1 configuration drops to about 25%. Compared to a DRAM-only configuration (2-2-2 DRAM), bandwidth drops by only 20% in read-only workloads in idle conditions vs. hybrid memory mode configurations. However, after the DRAM cache is full, bandwidth drops drastically, while the DRAM-only configuration is able to maintain its bandwidth under full load.

2.3.2 Static Placement

Static placement solutions start by profiling an application and determine, before runtime, the optimal data allocation strategy.

Leveraging instruction-level instrumentation in order to monitor data allocation and access patterns is a common strategy used by static placement algorithms [40–43]. These algorithms are computationally heavy and introduce an unbearable overhead by today's standards. In order to mitigate this overhead, hardware manufacturers developed hardware performance counters, such as Precise-Event

Based Sampling (PEBS) [44]. Hardware performance counters are a feature widely available in the current generation processors.

Hybrid approaches that combine PEBS with data allocation information through instruction-level instrumentation [45–47] come as an improvement to the previously referred solutions. However, they are only able to provide the programmer with information on structures that cause a high number of LLC misses, among other metrics, leaving the work of deciding where to place structures to the developer.

Servat et al. [37] further improve on the mentioned hybrid approaches by also automatically managing data allocation. Their solution starts by collecting metrics of an application's referenced memory objects into a trace file. After the metrics are collected, an access cost is attributed to each of the memory objects identified based on the number of LLC misses. Then, data placement is decided based on each structure's access cost and size by solving a combinatorial optimisation problem. Finally, memory allocation calls are swapped in the program to reflect the decided placement strategy.

The final step of the framework automatically substitutes the allocation and deallocation memory calls of each object to the memory level decided in the previous step. The end result is a distribution of pages based on absolute relevance that contrasts with temporal relevance, offered by dynamic placement solutions. This distribution will lead to a lower average access latency during execution at the cost of a much greater overhead before the execution actually starts.

2.3.3 Dynamic Placement

Whilst the previous solutions decide where to place data by profiling the workload and identifying the most cost-effective data, approaches in this section make the same decision during execution — i.e., dynamically. However, deciding placement dynamically introduces new challenges, such as adapting to workload changes during runtime in a way that maximises throughput with minimal overhead. Management policies for HMAs include, but are not limited to defining the migration and eviction rules for data:

Migration Policies. Data can either be: (i) be stored within a certain memory level since placement, (ii) migrate freely across levels or (iii) be selected to migrate based on a defined probability.

Eviction Policies. Data eviction occurs when space needs to be reclaimed from memory. In the traditional architecture, the only possible option is to evict data from DRAM. However, approaches for HMAs can choose to migrate data from DRAM to NVM, deciding not to evict it from primary memory while still freeing up space from the faster tier.

Data recently placed in memory, and especially, data that is write-intensive should be allocated to DRAM, due to an higher probability of being heavily accessed in the near future. This is due to the *temporal locality* principle already described in Section 2.1. Dynamic data placement algorithms may keep a buffer of free space in DRAM for this effect, migrating pages eagerly when free space falls below a threshold. When new data is referenced, the system is able to find sufficient space in DRAM

without needing to default to Optane allocation or evict data from the volatile tier, which improves overall performance. Another approach is to allocate data to DRAM on write operations but directly to NVM on read operations. The latter approach is usually applied when considering NVMs that perform similarly to DRAM in read latency, such as PCM and Optane.

Placement solutions in HMAs usually suffer from a problem known as thrashing. Thrashing results from cyclic migrations when two tiers of memory are full and decide to migrate equally intensive data between each other instead of evicting it. In order to mitigate this problem, dynamic solutions may decide to migrate data lazily [33]. Lazy migration consists of giving data a second chance before deciding to migrate it to a full tier, therefore reducing the total number of migrations. Thrashing can also be reduced by applying a probabilistic migration strategy. A more extreme approach is to fully restrict migration since placement and always decide to evict data. The latter is equivalent to static placement algorithms and therefore is also not adaptable to changes in the access pattern of a workload.

We start by introducing relevant research on dynamic placement in HMAs, based on traditional replacement algorithms.

Hybrid CLOCK Variants

Lee et al. propose M-CLOCK [33], a page-level approach, 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 memory controller. 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 with little computational overhead. The algorithm places all pages initially in the volatile media. This leads to overcrowding and a high number of migrations when DRAM is full. This comes as a major drawback of the algorithm, against other solutions that can choose to place data directly on NVM.

A similar adaptation of CLOCK, CLOCK-DWF [34], surfaced in the same time frame of M-CLOCK. CLOCK-DWF tracks the total number of writes for each page in a software-level structure. The structure is used to select pages above a write threshold unfit for NVM and candidate for DRAM. Also, in the event of a read operation, the requested page is directly placed in NVM, unlike M-CLOCK.

Although CLOCK-DWF incurs a higher memory overhead when compared to M-CLOCK, as M-CLOCK utilises an already existing hardware-managed bit to signal a page as “write-intensive” and CLOCK-DWF updates a specifically created data structure, it does not outperform M-CLOCK, when evaluated with standard workloads [33].

Hybrid LRU Variants

Seok et al. developed a page-level system [35, 36] based on LRU. The objective is similar to the previous solutions: maximising NVM endurance, which is achieved through minimising 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 update equation is provided below:

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

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, where pages with higher weight (W_{cur}) are considered more likely to be written in the near future and therefore fit for volatile memory and vice-versa.

The end result is an algorithm similar to CLOCK-DWF in terms of overhead but able to change its strategy regarding the most adequate tier of memory for a page based on a moving average instead of absolute values. This solution has the potential of further reducing the number of NVM writes when compared to Lee et al.'s extension of CLOCK, but the additional stored information makes it a worse fit when NVM endurance is secondary to the programmer, due to the additional overhead of keeping per-page weight values at software-level vs. taking advantage of the already existing MMU managed bits.

AutoNUMA Patch

A patch for Linux's autoNUMA is currently being developed [15] in parallel with this work. 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 Optane memories. The patch aims to dynamically promote frequently accessed pages to DRAM, while demoting colder pages to Optane.

Promotion from Optane to DRAM is achieved by leveraging the existing page fault mechanisms in autoNUMA, to determine the access frequency per page. The patch adapts the page tables' scanning routine to track the time at which a page's present and protection bits were modified. On the first

subsequent access, a variable, called *hint page fault latency* is calculated based on the difference between the times of the page fault and the bit modification. This variable is used to determine the intensiveness of a page, where pages with a latency lower than a defined threshold are classified as hot, and therefore promoted to DRAM. The threshold value 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 doubled, 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 Optane 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.

In the patch, a page must be promoted to DRAM before it can be migrated to remote memory. This makes it potentially unresponsive on the face of frequent access pattern changes. For example if a node suddenly frequently accesses a page residing in remote Optane, then the algorithm has to first migrate it to the remote DRAM before finally migrating it to the local DRAM, which takes one more scanning interval than the base solution.

2.3.4 NUMA-Aware Solutions Compatibility

The autoNUMA patch already extends its locality-based NUMA-aware approach with mechanisms that adapt to systems equipped with Optane. We see an adaptability of similar contention-aware algorithms to the NVM scenario. If we define the NVM nodes as a memory extension to the local CPU and DRAM nodes, then contention-aware algorithms could consider inter-socket communication with the base co-location, interleaving and replication mechanisms, while applying an HMA data placement approach within the socket.

Chapter 3

Implementation

In this chapter, we describe our dynamic page placement algorithm in detail. We start by providing a theoretical overview of the mechanisms used in our solution, then discuss how they can be implemented in practice, after which we define our goals, and describe how we fulfilled them.

3.1 Theoretical Overview

HMA systems consisting of DRAM and NVM require an algorithm that selects where to place each process' data, in order to maximise the potential offered by NVM-integration. Our solution considers an Optane-equipped system, and leverages its multiple configuration possibilities.

We decide where to place data at page level, as Linux already presents performant mechanisms that allow the migration and classification of pages. The decision is performed dynamically, adapting to workload changes during runtime.

When a process first references a page, we place it in the DRAM node closest to the process, defaulting to Optane when DRAM has no available space. We also migrate pages from DRAM to Optane when DRAM's free space is near depletion. This enables new pages to almost always be placed in DRAM. As DRAM space is scarce, we want to keep pages that are accessed often in DRAM, while opting to leverage the larger density offered by Optane for data that is accessed less often. Furthermore, our solution prioritises keeping write-intensive pages in the volatile tier, due to the Optane's write asymmetry.

Promotion to DRAM is performed opportunistically, when there is sufficient space in the volatile tier. In this case, we prefer pages that were accessed recently, prioritising pages that were recently modified

3.2 PnP in Practice

3.3 Goals

3.4 Goal Implementation

Basic test cases to compare the implemented model against other numerical tools (verification) and experimental data (validation)...

Chapter 4

Results

Insert your chapter material here...

4.1 Problem Description

Description of the baseline problem...

4.2 Baseline Solution

Analysis of the baseline solution...

4.3 Enhanced Solution

Quest for the optimal solution...

4.3.1 Figures

Insert your section material and possibly a few figures...

Make sure all figures presented are referenced in the text!

Images

Make reference to Figures 4.1 and 4.2.

By default, the supported file types are *.png,.pdf,.jpg,.mps,.jpeg,.PNG,.PDF,.JPG,.JPEG*.

See `http://mactex-wiki.tug.org/wiki/index.php/Graphics_inclusion` for adding support to other extensions.

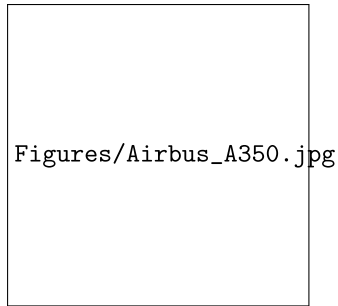
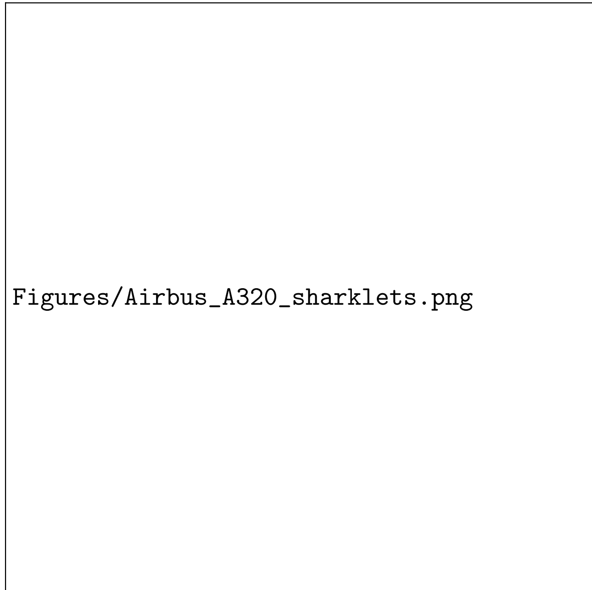
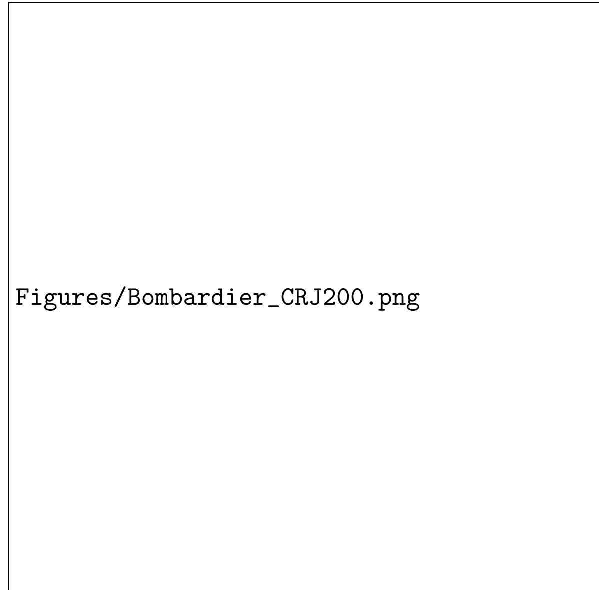


Figure 4.1: Caption for figure.



(a) Airbus A320



(b) Bombardier CRJ200

Figure 4.2: Some aircrafts.

Drawings

Insert your subsection material and for instance a few drawings...

The schematic illustrated in Fig. 4.3 can represent some sort of algorithm.

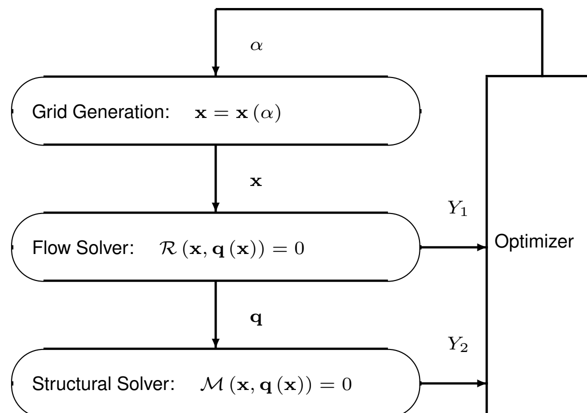


Figure 4.3: Schematic of some algorithm.

4.3.2 Equations

Equations can be inserted in different ways.

The simplest way is in a separate line like this

$$\frac{\mathrm{d}q_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(\mathbf{q}) = 0\,. \tag{4.1}$$

If the equation is to be embedded in the text. One can do it like this $\partial\mathcal{R}/\partial\mathbf{q} = 0$.

It may also be split in different lines like this

$$\begin{aligned} \text{Minimize} \quad & Y(\alpha, \mathbf{q}(\alpha)) \\ \text{w.r.t.} \quad & \alpha\,, \\ \text{subject to} \quad & \mathcal{R}(\alpha, \mathbf{q}(\alpha)) = 0 \\ & C(\alpha, \mathbf{q}(\alpha)) = 0\,. \end{aligned} \tag{4.2}$$

It is also possible to use subequations. Equations 4.3a, 4.3b and 4.3c form the Naver–Stokes equations 4.3.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}\left(\rho u_j\right) = 0\,, \tag{4.3a}$$

$$\frac{\partial}{\partial t}\left(\rho u_i\right) + \frac{\partial}{\partial x_j}\left(\rho u_i u_j + p\delta_{ij} - \tau_{ji}\right) = 0, \quad i = 1, 2, 3\,, \tag{4.3b}$$

$$\frac{\partial}{\partial t}\left(\rho E\right) + \frac{\partial}{\partial x_j}\left(\rho E u_j + p u_j - u_i \tau_{ij} + q_j\right) = 0\,. \tag{4.3c}$$

4.3.3 Tables

Insert your subsection material and for instance a few tables...

Make sure all tables presented are referenced in the text!

Follow some guidelines when making tables:

- Avoid vertical lines
- Avoid “boxing up” cells, usually 3 horizontal lines are enough: above, below, and after heading
- Avoid double horizontal lines
- Add enough space between rows

Make reference to Table 4.1.

Tables 4.2 and 4.3 are examples of tables with merging columns:

An example with merging rows can be seen in Tab.4.4.

If the table has too many columns, it can be scaled to fit the text widht, as in Tab.4.5.

Model	C_L	C_D	C_{My}
Euler	0.083	0.021	-0.110
Navier–Stokes	0.078	0.023	-0.101

Table 4.1: Table caption.

	Virtual memory [MB]	
	Euler	Navier–Stokes
Wing only	1,000	2,000
Aircraft	5,000	10,000
(ratio)	$5.0\times$	$5.0\times$

Table 4.2: Memory usage comparison (in MB).

	$w = 2$			$w = 4$		
	$t = 0$	$t = 1$	$t = 2$	$t = 0$	$t = 1$	$t = 2$
$dir = 1$						
c	0.07	0.16	0.29	0.36	0.71	3.18
c	-0.86	50.04	5.93	-9.07	29.09	46.21
c	14.27	-50.96	-14.27	12.22	-63.54	-381.09
$dir = 0$						
c	0.03	1.24	0.21	0.35	-0.27	2.14
c	-17.90	-37.11	8.85	-30.73	-9.59	-3.00
c	105.55	23.11	-94.73	100.24	41.27	-25.73

Table 4.3: Another table caption.

<table>
<tr><td rowspan="2">ABC</td><td colspan="4">header</td></tr>
<tr><td>1.1</td><td>2.2</td><td>3.3</td><td>4.4</td></tr>
<tr><td rowspan="2">IJK</td><td rowspan="2" colspan="2">group</td><td>0.5</td><td>0.6</td></tr>
<tr><td>0.7</td><td>1.2</td></tr>
</table>

Table 4.4: Yet another table caption.

Variable	a	b	c	d	e	f	g	h	i	j
Test 1	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
Test 2	20,000	40,000	60,000	80,000	100,000	120,000	140,000	160,000	180,000	200,000

Table 4.5: Very wide table.

4.3.4 Mixing

If necessary, a figure and a table can be put side-by-side as in Fig.4.4

Legend		
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 4.4: Figure and table side-by-side.

Chapter 5

Conclusions

Insert your chapter material here...

5.1 Achievements

The major achievements of the present work...

5.2 Future Work

A few ideas for future work...

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

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

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

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

[22] 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`.

[23] Chapter 2 - hpc architecture 1: Systems and technologies. In T. Sterling, M. Anderson, and M. Brodowicz, editors, *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`.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

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

[32] 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.

[33] 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`.

[34] 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.

[35] 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`.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] P. Cicotti and L. Carrington. Adamant: tools to capture, analyze, and manage data movement. *Procedia Computer Science*, 80:450–460, 2016.

[41] Intel. Intel Advisor. https://software.intel.com/en-us/advisor, . Accessed:2019-12-2.

[42] A. J. Peña and P. Balaji. Toward the efficient use of multiple explicitly managed memory subsystems. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 123–131, Sep. 2014. doi: 10.1109/CLUSTER.2014.6968756.

[43] A. Rane and J. Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 147–156. IEEE, 2012.

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

[45] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P.-T. Bremer, and B. Hamann. Dissecting on-node memory access performance: a semantic approach. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 166–176. IEEE, 2014.

[46] Intel. Intel VTune Amplifier. https://software.intel.com/en-us/vtune, . Accessed:2019-12-2.

[47] X. Liu and J. Mellor-Crummey. A data-centric profiler for parallel programs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.