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

Miguel Marques
Instituto Superior Técnico
Lisbon, Portugal

ABSTRACT

Intel Optane^{TM} DC Persistent Memory (DCPMM) is an emergent non-volatile memory (NVM) technology that is promising due to its byte-addressability, high density, and similar performance to DRAM. Hybrid DRAM and DCPMM architectures have the potential to improve applications by enabling them to place and directly access a larger working set in fast memory, and thus reduce the need of evicting data to slow block-based storage.

In order to reap the potential benefits of DCPMM integration, the underlying data placement algorithm must consider the disparity between access latency and bandwidth offered by both memory tiers in order to define an optimal strategy. However, due to the limited availability of commodity large-scale NVM, prior work on dynamic data placement in these architectures results from trace or simulation-driven experiments which are inherently inaccurate.

We propose Ambix, a dynamic page placement algorithm that is designed for and tested on a real system equipped with a DRAM-DCPMM memory configuration. We extensively discuss how different memory policies and distributions affect throughput and energy consumption in this 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, Data Placement

1 INTRODUCTION

The emergent workloads of tomorrow's Exascale systems 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]. However, scaling up memory capacity in order to meet these workloads' requirements incurs multiple challenges – from power limitations and cooling, to area constraints and cost [2].

Non-volatile memory (NVM) is an emerging class of memory that has the potential to mitigate these issues [3]. It is byte-addressable and has read/write latencies in the nanosecond range. Compared to DRAM, NVM is denser while being cheaper per GB.

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 [4, 5]. DCPM modules are up to 4x denser than DRAM, ranging from 128 to 512GB, and are compatible with DDR4 DIMM slots.

Integrating DCPMM comes as an interesting proposition for large-scale workloads, especially at Exascale-level, where processing power is often times underutilized due to memory- or I/O-related bottlenecks [5]. In workloads that require more memory than what is available per core, the additional capacity of the NVM tier can be leveraged in order to place a larger subset of the workload local to the running threads, and thus resort to fewer remote nodes.

In DCPMM-equipped systems, DCPMMs coexist with DRAM DIMMs. This constitutes a hybrid DRAM-NVM memory architecture. The DRAM-based nodes have better read and write throughput and latency, but suffer from limited capacity. In contrast, the DCPMM-based nodes offer worse performance than DRAM, especially for writes, but benefit from an increased density, granting ample memory resources. This duality raises the problem of where to place data objects. Allocating data to the appropriate memory tier, taking into account the performance differences between both tiers, becomes a decisive challenge to the effective scalability of Exascale applications [5].

The commercial availability of DCPMM 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.

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 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.

As a **third contribution**, we evaluate Ambix, as well as relevant page placement alternatives, with several benchmarks from the NAS Parallel Benchmark (NPB) [6] suite. 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.

2 BACKGROUND

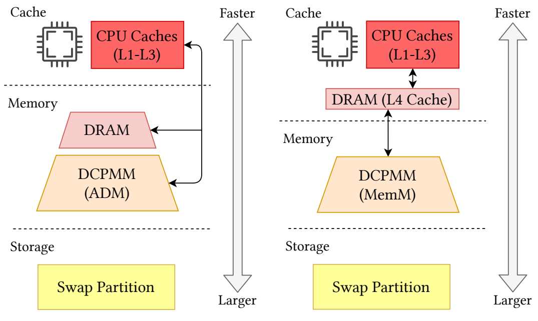
2.1 DCPMM internals

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 have *hybrid* memory architectures, where different DIMM configurations are possible, with varying DRAM-DCPMM capacity ratios, with the restriction that each iMC needs to be populated with at least one DRAM module. In multi-socket machines, the multiple hybrid DRAM-DCPMM memory systems (at each socket) are interconnected in a cache-coherent non-uniform memory access (ccNUMA) architecture.

2.2 DCPMM operation modes



(a) ADM Configuration **(b) MemM Configuration**

Figure 1: Architecture inside a socket

DCPMM can be configured in two different modes: *App Direct Mode* (ADM), and *Memory Mode* (MemM), which are illustrated in Figure 1.

In the MemM configuration, the DCPMM at each socket is transparently accessible to the operating system (OS) as a single memory node, whose capacity is the same as the total DCPMM capacity installed in the socket. The socket's DRAM is configured as an internal last-level cache, which interposes every access to the local DCPMM memory node.

When DCPMM is configured in ADM, both DRAM and DCPMM are directly exposed to the OS as two distinct memory nodes (at each socket). Each one can be directly accessed through load and store operations. Therefore, the OS has access to a larger aggregate main memory capacity, since the DRAM capacity is no longer hidden as a cache, and each access needs to be directed at one of the two types of memory.

2.3 Exposing DCPMM to applications

When MemM is used, the OS sees only one memory node per socket, as in a traditional DRAM-based NUMA system. Therefore, the standard NUMA-aware memory management mechanisms of modern OSes can still be used.

For instance, Linux prioritizes the allocation of new pages to the NUMA node (i.e., the socket's MemM node) that is local to the thread that *first-touched* the page. If that node is full, Linux falls back to allocating the new page to a remote NUMA node, trying to choose the one that minimizes the NUMA distance (according to the underlying NUMA interconnect topology). Additional allocation policies are provided through the *numactl* CLI, e.g., to interleave pages across NUMA nodes in a round-robin fashion . Once a page is allocated to a given NUMA node, it may be paged out to secondary storage if that node is occupied beyond a given threshold. The pages to evict from main memory are selected by a standard page replacement mechanism (e.g., LRU-based implementation in most Linux systems).

Linux also supports the Automatic NUMA balancer (AutoNUMA) [7], which is enabled by default. AutoNUMA dynamically allocates thread and pages across a NUMA system with the goal of maximizing locality, i.e., ensuring that threads and the pages they access are placed on the same socket.

When DCPMM is configured in ADM, the OS now views two distinct physical memory nodes at each socket. Starting with Linux v4.0, the plain NUMA model was extended to support hybrid memory subsystems in each socket. When used on a multi-socket machine with DCPMM configured in ADM, each socket comprises two logical NUMA memory nodes with differing characteristics. At each socket, the DRAM node is given priority when allocating pages accessed by local threads. When such preferential node is full, the DCPMM node is then chosen to allocate additional pages in that socket.

3 TAILORING PAGE PLACEMENT TO THE IDIOSYNCRASIES OF DCPMM

This section provides multiple insights on how pages should be distributed in DRAM-DCPMM systems. We leverage these conclusions in order to guide Ambix's design and implementation.

We answer two main questions:

- When a workload saturates DRAM bandwidth, is there any benefit in allocating a subset of pages in DCPMM? If so, what is the optimal distribution ratio for different workloads?
- Which page placement strategy provides the best possible throughput and lowest energy consumption, at different workload sizes?

In order to answer each question, we devise two benchmarks, which we describe next.

3.1 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 ~0.84x its effective size, in the benchmarks we run.

Both benchmarks generate multi-threaded workloads that allocate a test array, and then iterate sequentially over each of the array's pages for a fixed runtime, after which they output the number of accesses performed to the array per second (throughput). The test array is 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. Similarly, when a test array entry is modified, it is cached (load) and eventually written back (store), without accesses in between. Therefore, write-only workloads are *one read one write* (1R1W), and two accesses are counted when an array entry is modified.

Since we intend to study page placement between memory tiers, we isolate the benchmarks to a single NUMA socket. Besides measuring throughput, we also leverage *perf*, in order to collect memory energy consumption of the timed portion of the workload (without the allocation phase).

We assume that cold pages should never be prioritized in DRAM over intensive pages, and therefore do not factor them in the devised benchmarks. 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 Page Distribution Study

In order to study the throughput 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.

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

Pages Placed	Memory Access Demand (# Threads)									
in DRAM (%)	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
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 2: IWB Throughput Heat Map. Read-only Workload.

In Figures 2 and 3 we present two throughput heat maps from the RO and WO parametrizations of IWB, respectively.

Pages Placed	Memory Access Demand (# Threads)									
in DRAM (%)	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
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: IWB Throughput Heat Map. Write-only workload.

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 both 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 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. 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%).

In the write-only workload throughput heat map (Figure 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.

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 [4, 5].

3.3 Placement Strategy Study

The previous study assumed data sets with uniform access patterns, and tested 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

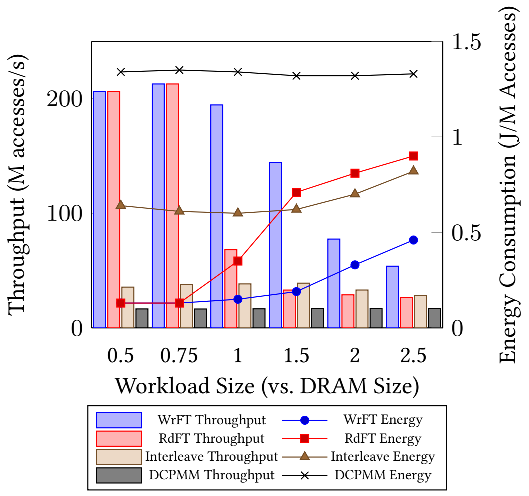


Figure 4: PB plot. 2R1W, 32 threads

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: (i) Write first-touch (WrFT): Allocates as many pages as possible in DRAM, initializing the write-intensive segment first; (ii) Read first-touch (RdFT): Similar to (i), but prioritizes read-intensive pages in DRAM (iii) Interleave: Follows the default interleave policy, which performs round-robin allocations between both DRAM and DCPMM nodes, leading to an even page distribution; (iv) 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 4 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).

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 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 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, and thus output the same throughput and energy efficiency results. At 1x, 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 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.

3.4 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%. We find that these results prove that a bandwidth-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. 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, 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 bandwidth-aware approach would seldom be beneficial in systems which integrate DCPMM. Therefore, we will design Ambix without a bandwidth-aware component.

4 AMBIX

Ambix manages page placement within a socket with a DRAM and ADM DCPMM tiers, illustrated in Figure 1a. Our solution requires minimal changes to the Linux kernel, and expands existing page placement mechanisms in order to accommodate the integration of DCPMM. Ambix leverages: (i) existing page walking mechanisms, (ii) the page table's dirty and reference bits, managed by the memory management unit (MMU), (iii) the *move_pages* syscall in order to migrate pages between tiers, and (iv) Processor Counter Monitor (PCMon) [8], which allows us to determine the bandwidth usage of each of the memories with hardware counters available in most modern Intel CPUs. Ambix combines the collected page and bandwidth metrics at runtime in order to perform page placement decisions that take advantage of each tier's characteristics, and ultimately improve throughput by several times compared to a default allocation policy.

Following the insights provided in Section 3, Ambix separates pages into three different categories: write-intensive, read-intensive and cold. The algorithm keeps as many write-intensive pages as possible in DRAM. If these do not fully occupy DRAM, Ambix prefers read-intensive pages over cold pages in the faster tier.
Ambix periodically determines the suitability of a current page distribution, following three main criteria:

- DRAM has enough free space to allow newly referenced pages to fit in the faster tier. These pages are expected to be accessed frequently after allocated, due to the *temporal locality* principle. Thus, Ambix maintains a defined buffer of free space in DRAM by demoting pages eagerly, before it is depleted.
- DCPMM's write throughput is nominal, indicating that the tier does not contain a significant amount of frequently modified pages.
- If DRAM is at capacity but the DCPMM's write threshold is substantial, no pages can be exchanged between both tiers, such that the threshold is reduced.

If Ambix finds the current distribution to be suboptimal based on these criteria, it devises a new placement decision that corrects the current distribution, migrating a subset of pages in a given orientation.

In order to select which pages to migrate, Ambix leverages unmodified page table walk (pagewalk) and PTE bit manipulation mechanisms, implemented in the Linux kernel. These mechanisms have a relatively stable implementation, which benefits Ambix by making it compatible with a wide range of kernel versions. Furthermore, configuring Ambix on a new kernel version requires only a single line of code, which exports the pagewalk routine, making it available to our solution.

In demotion scenarios, we apply concepts from the traditional CLOCK algorithm, modified to separate intensive pages into read- and write-dominated. Page promotion, on the other hand, applies a novel `delay` mechanism, which allows Ambix to identify recently accessed and modified pages in the DCPMM tier with low overhead.

Ambix also implements an exchange-based migration technique, using only pre-existing system calls, wherein an equal number of pages are switched between both tiers, thus preserving their current allocation.

In order to monitor multiple sockets, different instances of Ambix must be launched, each targeting a single socket. For simplicity of presentation, the following sections assume a single-socket scenario, where only one instance of Ambix runs.

4.1 Architecture

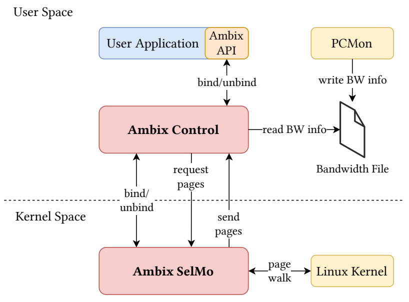


Figure 5: Ambix Architecture Overview

Figure 5 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`) [8], 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.

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 1: `PageFind` Modes and Goal.

4.2 Implementation

`SelMo` is the first component launched in Ambix, and waits a bind request from `Control`. When `Control` binds itself, it starts to monitor memory usage metrics and accept bind requests from external processes. 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.

Ambix avoids direct communication between a user application and `SelMo`. Instead, it performs a sanity check in user-space, before forwarding the request to kernel space.

4.2.1 Control. `Control` periodically monitors current memory usage and throughput values per NUMA node. The node IDs specific to a socket's DRAM and DCPMM tiers are statically defined in a header file, imported by both of Ambix's components. In order to get the current memory allocation of each tier, the component leverages the `libnuma` library [9], which provides per-node statistics on total node size and utilization. It is also able to detect the presence of write-intensive pages in DCPMM, without communicating with `SelMo`, by reading the information file generated by `PCMon`.

Depending on the collected metrics, `Control` devises a new placement decision and sends a `PageFind` request to `SelMo`. The request contains the number of pages to find and specifies the selection criteria, or mode.

In Table 1 we summarize the multiple modes, describing the tier from which they select pages, and their respective goals.

By design, DRAM has a defined maximum usage threshold below its actual size. Above the threshold, Ambix 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 defined write throughput threshold. If DCPMM's current throughput is above the defined threshold, `Control` requests intensive pages and promotes them to DRAM. If DRAM is above its usage threshold, an equal number of pages must be demoted, such that the free space buffer is preserved. Therefore, `SWITCH PageFind` is sent to the module. Otherwise, it tries to maximize the faster tier's utilization, by promoting as many intensive pages as possible such that the usage threshold is not surpassed, by requesting a `PROMOTE_INT PageFind`.

Inversely, if the DCPMM's throughput is below its threshold, two decisions can be performed. If DRAM has enough available space, `Control` allows cold pages to be eagerly promoted, via a `PROMOTE PageFind`. Otherwise, if DRAM is near depletion, a `DEMOTE PageFind` is requested, such that cold pages are demoted.

Before sending any `PageFind` request that will promote pages, `Control` requests `SelMo` to clear the R/D bits of all PTEs pointing to pages in the DCPMM tier, via the `DCPMM_CLEAR PageFind`, after which it waits for a configurable `delay`.

`Delay` affects the access frequency at which a page is considered intensive. 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.

If, after a `PageFind` 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 and made available by the kernel.

4.2.2 SelMo. `SelMo` iterates over each bound process' page table, in order to select which pages `Control` should migrate in a given direction, based on the `PageFind` request's goal.

The module stores the bound processes' information in an array of `task_struct` structures, which are defined and maintained by the Linux kernel. The structure contains the PID of a process and a pointer that allows access to its page table, among other information. When a process ends, its `task_struct` is freed. Before any request `SelMo` updates the array, removing all entries from processes that are no longer running from it.

The module leverages the kernel-implemented `walk_page_range()` routine, which iterates over a defined virtual address range.

`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 `PageFind` 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 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 last group contains pages that the algorithm decides to retain in their current memory tier, since they fail to meet the selection criteria.

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.

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 modified 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 R/D bits. Instead, the algorithm deems a page in the DCPMM intensive if only referenced during the `delay` window, and write-intensive if modified.

Pages in the priority or backup groups are placed into 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.

When: (i) the number of priority 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. 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.

Then, a reply-back phase begins, which prepares a final page array to be sent back to `Control`, initially containing all pages in the priority group. 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 priority and backup pages.

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;
```

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 the faster tier.

Page classification is achieved by using the `pte_young()` and `pte_dirty()` functions, defined in the Linux kernel, which indicate if a PTE has its R, and D bits set, respectively.

In order to find the NUMA node in which the PTE's 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 ID where the page frame resides.

Every observed page that does not meet the priority array criteria has its PTE's R/D bits cleared by the callback. 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 a faster tier. The callback selects any page from DCPMM, with an emphasis on read- and 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. Similarly to the `DEMOTE` callback, all pages that do not meet the priority criteria have their referenced and modified bits cleared.

In scenarios where DRAM space is scarce, `Control` sends a `PROMOTE_INT PageFind` instead, which is associated with a callback where only write or read-intensive pages are selected for promotion.

The `SWITCH PageFind` differs from the previous variants, as the module is requested to find N pages to swap between both tiers. In this mode, we perform two page table iterations, starting with the slower tier, associated with the `PROMOTE_INT` callback. The module then looks for the same number of pages found in the faster tier. In this case, `SelMo` uses 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.

When the module receives an `DCPMM_CLEAR PageFind` request from `Control`, it performs a *pagewalk* with a callback that clears the R/D bits from the PTEs associated to the pages allocated in DCPMM. In this mode, no pages are selected or sent back to `Control`. The callback precedes a `PageFind` operation in DCPMM, and is used to tune the frequency at which a page is considered write or read-intensive.

5 EVALUATION

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 [10] and Memos [11], 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.

5.2 Experimental Setup

5.2.1 Hardware Configuration. The hardware configuration is identical to the one described in Section 3

5.2.2 OS Configuration. 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.

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 to make the `walk_page_range()` routine callable from kernel-level modules.

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

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.

The used variables are defined as follows:

- `Control`'s periodicity, i.e., the frequency at which the it 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, after which `Control` reacts to the presence of write-intensive pages in DCPMM is set to 50MB/s.

5.3 Baseline

This section presents further information about our chosen baselines. 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.

We choose the *tiering-0.4* version [12], 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[2].

Additionally, we incorporate Memos into the comparison, as it presents a bandwidth-aware approach, which starts by initially placing every page in NVM. 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 [3].

5.3.2 Default Configurations. We benchmark dynamic placement solutions against the *ADM-default* and *MemM* configurations. The former will set the baseline for improvement, as it provides no dynamic placement decisions, we expect it to perform best in workloads with low footprint, and enable us to compare the classification and migration/caching overhead of the dynamic solutions and *MemM*.

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

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, 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 evaluate multiple workloads from the NAS Parallel Benchmark (NPB) suite [6]. NPB provides benchmarks which mimic common access patterns in computational fluid dynamics applications, and was designed to evaluate the performance of parallel supercomputers.

We choose the BT (3.5R:1W),FT (1.7R:1W), MG (4R:1W), and CG (>60R:1W) benchmarks from the OpenMP[13] version of NPB v3.4.1, which present a good balance between computational cost and memory bandwidth requirements, and have different read/write intensity.

We 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 (~0.8x DRAM size), medium (~1.5x) and large (~3.5x) parametrizations.

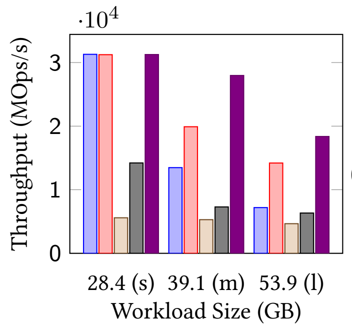
5.5 Evaluation

Figure 6 presents four graphs which contain the average throughput of the BT (fig. 6a), FT (fig. 6b), MG (fig. 6c), and CG (fig. 6d) 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 6e). In all graphs, the y axis represents throughput in million operations per second (M Ops/s).

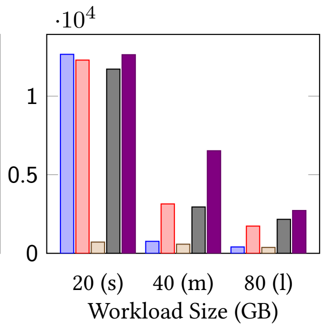
5.5.1 Small Workloads – Overhead Study. In the small data sets, *autonuma* has an identical performance to *ADM-default* in the FT, MG, and CG workloads, but presents a 60% lower throughput in the BT workload, when compared to both.

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.

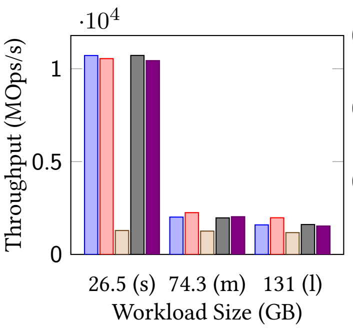
In contrast, the migration mechanism of Ambix not only keeps a smaller 5% buffer in the faster tier, but also chooses to never demote a page it considers write-intensive even if it fails to find a suitable amount of cold ones. With this mechanism, Ambix is able to fully



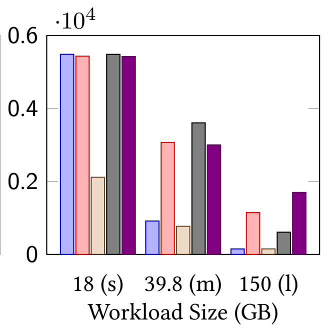
(a) BT (3.5R:1W)



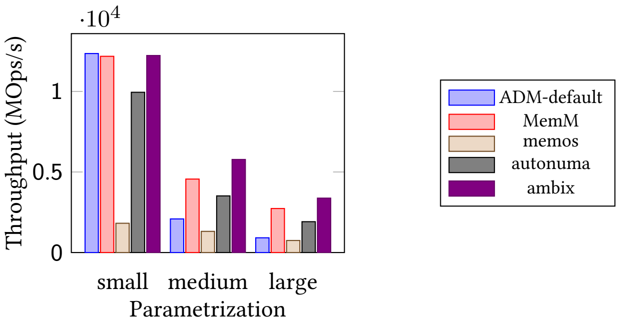
(b) FT (1.7R:1W)



(c) MG (4R:1W)



(d) CG (60R:1W)



(e) NPBAVG

Figure 6: NPB Plots. Varying Workload Size, Single socket, 32 threads

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 6e). 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.

5.5.2 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. 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 workloads, *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.

6 RELATED WORK

Shortly after the first research papers started showcasing promising breakthroughs with NVM technologies [14] – nearly a decade ago –, many proposals for data placement on the upcoming hybrid DRAM-NVM systems have been published. While some rely on profiling and compile-time instrumentation [15–17], others propose a transparent support by extending OS kernel's page management mechanisms to dynamically migrate an application's pages to the most appropriate tier [10, 11, 18–21]. Since commodity NVM was not available at the time, the design of all such proposals is based on speculative assumptions – not only about the performance of NVM, but also about how NVM would be integrated into the hardware architecture and supported by the OS. Furthermore, these proposals were evaluated through inaccurate software-based emulation.

7 CONCLUSION

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.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

[12] 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].

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.