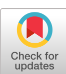


PDF Download
3736174.pdf
29 January 2026
Total Citations: 0
Total Downloads: 784

Latest updates: https://dl.acm.org/doi/10.1145/3736174

Published: 02 July 2025
Online AM: 16 May 2025
Accepted: 30 April 2025
Revised: 27 March 2025
Received: 01 October 2024

Citation in BibTeX format

RESEARCH-ARTICLE

Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems

MANOLIS KATSARAGAKIS, National Technical University of Athens (NTUA), Athens, Attica, Greece

CHRISTOS BALOUKAS, National Technical University of Athens (NTUA), Athens, Attica, Greece

LAZAROS PAPADOPOULOS, Democritus University of Thrace, Komotini, Eastern Macedonia and Thrace, Greece

FRANCKY CATTHOOR, Interuniversity Microelectronics Centre, Leuven, Vlaams-Brabant, Belgium

DIMITRIOS J SOUDRIS, National Technical University of Athens (NTUA), Athens, Attica, Greece

Open Access Support provided by:

National Technical University of Athens (NTUA)

Interuniversity Microelectronics Centre

Democritus University of Thrace

Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems

MANOLIS KATSARAGAKIS, Electrical and Computer Engineering, National Technical University of Athens, Zografou, Greece and KU Leuven, Leuven, Belgium
CHRISTOS BALOUKAS, Electrical and Computer Engineering, National Technical University of Athens, Zografou, Greece
LAZAROS PAPADOPOULOS, Electrical and Computer Engineering, Democritus University of Thrace, Thrace, Greece
FRANCKY CATTHOOR, IMEC, Leuven, Netherlands
DIMITRIOS SOUDRIS, National Technical University of Athens - Zografou Campus, Zografou, Greece

The need for increased memory capacity, which also needs to be affordable and sustainable, leads to the adoption of heterogeneous memory hierarchies, combining DRAM and NVM technologies. This work proposes a memory management methodology that relies on multi-objective optimization in terms of performance, energy consumption and impact on NVM's lifetime, for applications deployed on heterogeneous (i.e., DRAM/NVM) memory systems. We propose a scalable and lightweight data structure exploration flow for supporting data type refinement based on access pattern analysis, enhanced with a weighted-based data placement decision support for multi-objective exploration and optimization. The evaluation of the methodology was performed both on emulated and real DRAM/NVM hardware for different applications and data placement algorithms. The experimental results show up to 58.7% lower execution time and 48.3% less energy consumption compared with the results obtained by the initial versions of the applications. Moreover, we observed 72.6% less NVM write operations, which can significantly extend the lifetime of the NVM memory. Finally, thorough evaluation shows that the methodology is flexible and scalable, as it can integrate different data placement algorithms and NVM technologies and requires reasonable exploration time.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Software system structures**; • **Information systems → Data structures;**

Additional Key Words and Phrases: Memory management, hetegerogeneous memory systems, dynamic data type refinement, DRAM, NVM, dynamic data types

This work has received funding from the EU Horizon Europe programme PRIVATEER under Grant Agreement No. 101096110.

Authors' Contact Information: Manolis Katsaragakis, Electrical and Computer Engineering, National Technical University of Athens, Zografou, Greece and KU Leuven, Leuven, Belgium; e-mail: mkatsaragakis@microlab.ntua.gr; Christos Baloukas, Electrical and Computer Engineering, National Technical University of Athens, Zografou, Greece; e-mail: christosbaloukas@microlab.ntua.gr; Lazaros Papadopoulos, Electrical and Computer Engineering, Democritus University of Thrace, Thrace, Greece; e-mail: lpapadop@ee.duth.gr; Francky Catthoor, IMEC, Leuven, Netherlands; e-mail: Francky.Catthoor@imec.be; Dimitrios Soudris, National Technical University of Athens–Zografou Campus, Zografou, Attica, Greece; e-mail: dsoudris@microlab.ntua.g.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1544-3973/2025/06-ART80
https://doi.org/10.1145/3736174

ACM Reference Format:
Manolis Katsaragakis, Christos Baloukas, Lazaros Papadopoulos, Francky Catthoor, and Dimitrios Soudris. 2025. Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems. *ACM Trans. Arch. Code Optim.* 22, 2, Article 80 (June 2025), 27 pages. https://doi.org/10.1145/3736174

1 Introduction

Over the last few years, the rapid growth of applications generating huge amounts of data that need to be stored and processed is increasing at a high pace. Several application domains across the computing continuum are enabled by **Machine Learning** (**ML**) algorithms [32] and rely on databases for performing transactions based on complex queries [25], or process streams of data generated in real-time at the edge of IoT networks [54]. Such application paradigms often rely on in-memory and in-place data processing over complex memory hierarchies to avoid expensive data transfers. Thus, developers attempt to meet high performance requirements and address energy consumption constraints, by putting severe pressure on the main memory subsystem.

Traditional HPC systems consist of DRAM-based main memories. However, DRAM technologies have hit their physical limits over the last years [24, 66], as they exhibit limitations in terms of scalability and sustainability [55, 67]. Integrating multiple DRAM DIMMs to enable higher performance and satisfy increased capacity requirements is becoming inefficient, due to increased energy consumption attributed to leakage and refresh power of the DRAM technology. Therefore, the maintainability cost of state-of-the-art supercomputing systems makes it highly unattractive to invest in memory capacity increase, entirely based on DRAM [5]. Existing literature indicates that a typical DRAM main memory subsystem consumes up to 50% of the energy of an HPC system [17].

Aiming to tackle these limitations, emerging memory technologies are investigated, including **Non-Volatile Memories** (**NVM**), also known as **Persistent Memories** (**PM**). NVM technologies, such as the 3D-XPoint [70], which is a subclass of Phase-Change Memory [69], Spin-Transfer Torque RAM [44] and Resistive RAM [65] are products of an active research area, involving both academic and industrial communities. The Intel Optane DCPM (Data Center Persistent Memory) is the one and only commercial NVM up to now. Various state-of-the-art HPC systems integrate up to petabytes (PB) of Optane DCPM, such as the DAOS [31] and the Alibaba Cloud [48], aiming to expand the memory capacity in a sustainable and cost-effective way. Moreover, upcoming HPC systems are expected to further benefit from emerging memory technologies enabled by advanced interconnection protocols. For example, the **Compute Express Link** (**CXL**) protocol will allow increased capacities based on new products such as the Samsung Memory Expander [41].

NVM technologies offer higher density compared with DRAM, resulting in increased memory capacities at reduced costs [27]. Furthermore, their integration significantly impacts energy consumption and sustainability [58]. However, despite the undeniable advantages of rapidly advancing NVM technologies, their constraints present considerable challenges for application developers. NVMs exhibit increased access latency, low bandwidth, and reduced endurance compared with DRAM technologies, necessitating specialized programming techniques and effective tools for optimization. Additionally, NVMs demonstrate asymmetry in the latency and energy of read/write operations [51]. Directly replacing DRAM with NVMs, such as the Intel Optane DCPM, can notably diminish performance due to effects like write throttling and concurrency contention [37]. Consequently, leveraging suitable programming tools, such as memory management libraries and performance profiling frameworks, becomes essential for efficiently harnessing the capabilities of NVM technology. NVM DIMMs are commonly paired with DRAM modules to

compose heterogeneous DRAM/NVM systems, addressing these performance limitations and enhancing overall system efficiency.

Toward the efficient exploitation of the DRAM/NVM heterogeneous memory systems, several data placement algorithms have been proposed over the years [20, 22, 28, 30, 40, 45, 49, 50, 53, 56, 59, 63, 68]. These algorithms typically recommend or perform an efficient placement and/or runtime migration of data objects, memory pages, or application data structures over heterogeneous memory systems, considering the performance and/or the energy consumption as optimization objectives. Although the data placement algorithms are often sophisticated and manage to efficiently exploit complex memory hierarchies for applications under execution, their results are often limited by the fact that the original applications were normally designed for DRAM and not for heterogeneous memories.

From the application's design perspective, developers do not normally consider any data optimization for data organization at application-level, which could enable further improvements in the data placement results. Although several works propose data structure design targeting NVM or heterogeneous memories, applications still rely on DRAM-oriented data structures, due to their simplicity and, in contrast with NVM technologies, the extensive library support [51]. Also, even though application-level data optimization methodologies have been proposed several years ago [11, 15], their interplay and their impact on the data placement over heterogeneous memories has not been investigated, yet. Moreover, state-of-the-art solutions rely on extensive and resource-hungry profiling mechanisms, leading to huge requirements for storage and processing. Existing solutions that consider both data structure exploration and data placement typically rely either on instrumentation tools [11, 35] or on system-level profiling mechanisms [42, 47]. Even though profiling-based solutions provide extensive data for further processing in the decision-making phase, these are usually time-consuming and impose significant overhead both during the examined application's execution. Last, their scalability is limited, due to the rapidly increasing storage requirements, as the number of parameters is growing.

This work is a contribution to bridging the gap between the application-level data optimizations and the data placement algorithms and investigating their interaction. More specifically, in this work, we propose a **memory management methodology that provides dynamic data type refinement and organization. Our proposed solution relies on access pattern analysis, enhanced with a decision support flow for data placement that provides multi-objective design space exploration for co-optimizing performance, energy and NVM lifetime**. We aim at optimizing the data organization of applications in terms of data accesses and memory capacity requirements, by exploring and recommending alternative data organization options at the application level [11]. We form and solve a constraint-aware multi-objective optimization problem through user-defined thresholds and we conduct a thorough exploration of varying dynamic data types over discrete access patterns, thus generating a knowledge base for data structure behavior and performance. The methodology is evaluated using several NVM technologies: both real (Optane DCPM) and emulated (PCM, RRAM, STT-MRAM). It is supported by an open-source tool flow[1] that automates many steps of the methodology. It targets developers, who aim at deploying applications on computing systems that integrate Optane DCPM (or other types of NVM DIMMs), or they aim at investigating through emulation/simulation how their applications would behave on heterogeneous memory systems in terms of performance/energy/impact on the NVM lifetime.

The rest of this article is organized as follows. Section 2 presents an overview of the related work and Section 3 illustrates the fundamental background and the motivation. In Section 4, the access pattern analysis, the proposed methodology and the implementation of our flow are described in

[1] https://github.com/mkatsa/DDTR-DRAM-NVM

detail. Section 5 shows the experimental evaluation and the discussion of the main observations. Finally, Section 6 concludes this work.

2 Related Work

Data structure optimization: Various works have been conducted on the optimization of data structures at the application-level, based on design-space exploration [11] or leveraging AI techniques [35]. With regard to data structures optimized for DRAM/NVM systems, authors of Reference [51] propose a framework for implementing persistent data structures for NVM, while in Reference [46] the integration of tree and hash-based data structures for persistent memories is proposed. Similarly, authors of Reference [19] revisit popular persistent indexing structures on NVM and evaluate them in terms of performance results.

General purpose placement over heterogeneous DRAM/NVM Systems: These works can be classified based on the placement granularity level [40, 49, 56, 59]: (i) data structures, (ii) memory objects and (iii) memory pages. Each approach has different advantages and drawbacks. The authors of [56] present an online profile-guided data tiering solution involving placement at page granularity for heterogeneous memory systems targeting improved performance for HPC applications. Other indicative approaches include a static code instrumentation tool to automatically perform memory object placement for performance and energy optimization [49]. At data structure placement granularity, write-aware data structure placement for cached and uncached NVMs is proposed in Reference [59]. AI-based data placement at page granularity is proposed in Reference [40].

Application-domain specific placement over heterogeneous DRAM/NVM Systems: Recent research has also focused on application-domain specific placement strategies, aiming to outperform domain-agnostic placement algorithms. Within the realm of database-related workloads, studies such as. Reference [45] have investigated placement strategies optimized for database operations. Similarly, in the context of big-data applications, authors of References [20, 68] have explored placement techniques tailored to the complexities of processing large datasets. Graph-based applications, such as [53] and [22], have also seen advancements in domain-specific placement approaches, accommodating the intricacies of graph processing tasks. Moreover, in the domain of **Deep Neural Network** (**DNN**)-based applications, placement strategies have been tailored to optimize the performance of neural network inference and training tasks, such References [28] and [63]. These application-domain specific placement strategies underscore the importance of customizing placement approaches to the unique requirements of diverse application domains, ultimately leading to improved system performance and resource utilization.

It can be noticed that none of the related works investigates the interaction between application-level data optimizations and data placement on heterogeneous memory systems. Moreover, the NVM lifetime has not been considered yet as an axis of optimization, apart from performance and energy consumption, while the existing solutions rely on expensive profiling mechanisms. Representative profiling for the main memory access level requires workloads that have a huge length in order to derive an accurate view of the hardware/software later on.

The present work extends existing literature by: (i) integrating a third level of optimization objective, going beyond performance and energy consumption, for investigating the impact of high-level data organization optimizations on the NVM's lifetime, (ii) providing a lightweight and scalable solution for dynamic data structure refinement, through effective access pattern analysis, (iii) integrating a fast, scalable, and lightweight decision support flow which minimizes the overhead and reduces exploration time through forming and solving a constraint-aware multi-objective problem for optimizing performance, energy, and NVM lifetime and (iv) providing

an in-depth analysis of evaluation results, impact on NVM's lifetime, distribution of data over the heterogeneous memory, scalability, and exploration overhead evaluation of the proposed framework.

3 Background and Motivation

This section provides an overview of the NVM technologies, focusing on the Optane DCPM and discusses the motivation for this work.

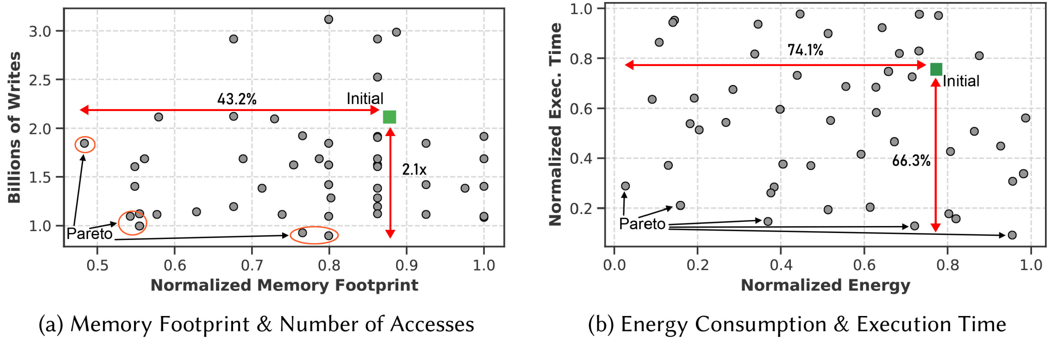
3.1 NVM Technologies and Intel Optane DCPM

Recent advances in NVMs enabled the adoption of new memory products in state-of-the-art computing systems, by integrating heterogeneous memory hierarchies. Typically, NVMs provide complex specifications, such as the asymmetry of read and write access latency and energy consumption [51]. NVM hardware endurance, in terms of the number of write cycles that can be applied to the memory segment of the target memory until it becomes unreliable, is another parameter that should be taken into consideration. The Intel Optane DCPM, which is the only commercial NVM up to now, is a PCM-based memory technology and it is based on the 3D-XPoint architecture. Intel Optane can operate in two distinct modes: (i) *Memory Mode* and (ii) *App-Direct Mode*, which should be selected based on the specific use case and workload requirements. The former configures Optane memory as volatile memory and extends the memory capacity beyond the physical DRAM installed in the system, while DRAM is utilized as L4 cache. In the latter configuration, Optane DCPM operates as persistent storage on the same layer as DRAM and can be directly accessed by the application. In this work, we utilize Intel Optane DCPM in *App-Direct* mode, which allows for direct control of which objects are placed in either DRAM or Optane memory.

3.2 Motivation

Can dynamic data type organization improvement enhance placement policies? Even though prior research has examined solely data type refinement [11, 15, 19, 46] or solely data placement on heterogeneous memory systems [40, 49, 56, 59], the effective co-design of both has not been considered yet. Thus, we first aim at investigating the potential influence of application-level data organization improvement over data placement in heterogeneous DRAM/NVM systems. We conduct a motivational experiment exploring the design space of data structure implementations and replacing original implementations with those yielding superior results in terms of data accesses and memory footprint. More specifically, a set of Pareto optimal data structure implementations in terms of data accesses and memory footprint is derived, where each point represents a distinct combination of data structure implementations. A variant of the dynamic data type refinement methodology tailored for DRAM-only systems is documented in References [11, 39]. As motivational example, we utilize the SMS-EMOA application, derived by Shark-ML library [32], which is illustrated in Figure 1. Figure 1(a) depicts the output of the methodology for the evaluated application, where each point in the Pareto space signifies a version of the application with identical functionality but different data structure implementations. It is evident that developers can exchange required memory capacity for reduced data accesses (and vice versa) by modifying the data structure implementations of the optimized application. More specifically, memory footprint can be potentially reduced up to 43.2% and the number of write accesses can be reduced up to 2.1×. These factors form a set of high-level indicators, which show that performance, energy and NVM lifetime can be optimized.

Additionally, we deployed the distinct versions of the application on a heterogeneous DRAM/NVM system using a state-of-the-art data placement algorithm [49]. As depicted in Figure 1(b), we observe (i) that the dynamic data type refinement step facilitated improved performance and



(a) Memory Footprint & Number of Accesses (b) Energy Consumption & Execution Time

Fig. 1. Investigating the impact of the application-level optimization for improved data organization on the results of data placement, in terms of execution time and energy consumption for the SMS-EMOA [32] application. (a) The output of dynamic data type refinement: application versions, each one having a different data organization (b) The execution time and energy consumption after placement and execution on a DRAM/Optane DCPM system for each version.

energy efficiency compared with the initial application version (up to 66.3% lower execution time and up to 74.1% lower energy consumption) and (ii) that tradeoffs between execution time and energy consumption can be achieved by selecting different application versions. Furthermore, we noted that the data organization of the application significantly impacts the number of physical read and write accesses (i.e., as measured in the NVDIMMs). Through judicious selection of data structure implementations, the number of NVM write operations can be reduced by up to 2.23× orders of magnitude compared with the initial application version, which constitutes a high-level indicator for optimizing the NVM lifetime [36]. Thus, it is indeed evident that dynamic data type organization improvement can enhance placement policies and further exploration is required to co-optimize the NVM-lifetime through application-level refinement, alongside with the performance and the energy consumption.

Overhead and limitations of profiling mechanisms: Profiling overhead, often incurred during efforts to enhance memory management, can result in substantial performance overhead and prolonged exploration times. Recent research heavily relies on extensive profiling mechanisms, which constitute the most time-consuming and resource-intensive phase. For example, research presented in Reference [16] indicates that full memory and copy profiling can impose up to 53% performance overhead, while in Reference [39], the overall exploration time spans up to 3 hours, with advanced profiling constituting the most time-consuming step for applications consisting of up to 5 dynamic data structures. Therefore, the scalability to high number of data structures cannot be performed. Various tools have been developed for comprehensive and precise memory access analysis [57], relying on instrumentation mechanisms such as Valgrind [10] and Extrae [2]. However, these tools entail significant overhead for general and precise profiling, even when only a fraction of the derived results are needed. Moreover, the overhead escalates notably with the size and complexity of the target workload, thereby constraining the scalability of existing solutions. Our research objective aims to substantially reduce exploration time by offering a lightweight exploration approach while delivering near-optimal solutions. Our approach relies on a dynamic data type access pattern analysis solution, where each data type is characterized as suitable for a set of access patterns, based on its memory accesses, which form a high-level indicator of an application's performance and energy consumption [37, 38]. Though this approach, we avoid resource and time consuming extensive profiling solutions. The access pattern analysis mechanism is detailed in Section 4.1.

Table 1. The Design Space of Data Structure Implementations in the Data Structures Library Used for the Data Organization Optimization

Data Structure	Short Description
VEC	An array that can grow or shrink dynamically(Vector)
SLL	Single Linked List
SLLR	SLL with a pointer to the last accessed element
DLL	Double Linked List
DLLR	DLL with a pointer to the last accessed element
MAP-RB	An associative container(map in STL) that stores pairs of key-value in a Red-Black tree
MAP-AVL	A map that stores key-value pairs in an AVL tree

4 Overview of the Proposed Methodology

4.1 Dynamic Data Type Access Pattern Analysis and Exploration

As a pre-processing step of our proposed methodology, we conduct an extensive analysis for the impact of different dynamic data structures, aiming to derive key indicators for the impact of data structures over persistent memory systems. Analyzing memory access patterns is crucial for optimizing data retrieval efficiency and minimizing performance bottlenecks [29], ultimately enhancing overall performance. Prior research has not conducted a thorough mapping of dynamic data types and access patterns over heterogeneous memory systems. Our objective is to derive a mapping of dynamic data types and access patterns, that operate efficiently over heterogeneous memory systems, thus allowing to select the optimal data structure implementation for the corresponding access pattern [11, 35, 39, 46, 51]. Through the effective analysis of dynamic data structures and access patterns, we avoid the time-consuming and resource-hungry profiling process on the exploration methodology presented in Section 4.2. The dynamic data structures and library support are derived from our previous research conducted in Reference [39] and are briefly summarized in Table 1. The library of dynamic data types is publicly available under open-source licence. We perform the analysis over Vector (VEC), Single Linked List (SLL), SLL with Roving pointer (SLLR), Double Linked List (DLL) and DLL with Roving pointer (DLLR). Our approach covers both object storage and key/value container data types, evaluating various implementation variations without changing the container's type and purpose. Object storage container types like STL's vectors and lists directly store individual elements and access them sequentially or via indices, acting as simplified containers without the complexities and overheads associated with key management. Thus our Dynamic Data Type Refinement step evaluates several variations in their implementation that still keep this simplified approach. On the other hand, for the key/value container types, we evaluate different implementations like Red Black Trees and AVL trees, that are suitable for accessing data by key. The access patterns are selected based on the following criteria: (i) the frequency met throughout the application, (ii) the impact of the access patterns when deployed on persistent memories [14, 18, 29, 33]. The examined patterns are summarized and briefly described in Table 2, along with their corresponding regular expressions to illustrate how they are identified through pattern matching. More specifically, we analyze the following access patterns at **data structure-level**, based on the sequence that the objects of the data structure are processed:

— ***Sequential***: Sequential access patterns involve accessing memory locations in a linear fashion. This pattern optimizes memory retrieval efficiency by exploiting spatial locality, enhancing data throughput and reducing latency in sequential data processing tasks. By accessing adjacent memory locations, sequential patterns leverage the underlying memory hierarchy more effectively, facilitating faster retrieval times [18].

Table 2. Short Description and Overview of Memory Access Patterns with Corresponding Regular Expressions

Access Pattern	Short Description	Regular Expression
Sequential	Data are accessed in linear order, typically from the beginning to the end of a data structure.	`(r|w)+`
Reverse Sequential	Data are accessed in linear order, typically from the end to the beginning of the structure.	`(r|w)+` (in reverse)
Random	Data are accessed randomly; the next accessed data is unknown.	`(rw|wr|rr|ww)*`
Strided (N)	Data are accessed in a pattern where each access is separated by a fixed distance *N* within the data structure.	`(r|w)(.N-1(r|w))*` e.g., for N=1: `r.w.r.w.`
Gather	Random read accesses and sequential write access operations.	`(r.*)+w+`
Anti-Gather	Sequential read accesses and random write access operations.	`r+(w.*)+`

— ***Random***: Random access patterns involve accessing memory locations in a non-sequential manner, however, without having a predictable behavior. This pattern often arises in scenarios such as hash table lookups, tree traversal, or accessing elements of a matrix in a non-linear order. Unlike sequential access, random access may lead to lower data throughput due to the lack of spatial locality, thus resulting in higher memory access latency and decreased overall system performance. Moreover, NVM technologies are known to have significant performance degradation due to the randomness of access pattern of the application deployed [71], thus the selection of an efficient data structure implementation is critical.

— ***Strided with step N***: A strided access pattern with a step of *n* involves accessing memory locations in a pattern where each access is separated by a fixed distance *N* within the data structure [52]. Strided access patterns are common in algorithms such as matrix operations, where accessing elements with a fixed step allows for parallelization and optimization. However, excessive striding can lead to increased cache misses and reduced memory access locality [71], potentially impacting performance in memory-bound tasks.

Our approach extends prior research conducted on access pattern analysis for dynamic data structures, such as [15], which rely on analytical models and focus on sequential and random access patterns. In contrast to prior research, we also incorporate `Reverse Sequential` and `Strided` access patterns into our experiments, which have not been previously considered. In our approach, we conduct experiments aimed at accurately deriving the number of data accesses for each individual data structure across different access patterns. Figure 2 illustrates the application-level data accesses of operations performed over discrete combinations of dynamic data type implementations and access patterns. Specifically, we focus on the STL `insert` (top), `get` (middle), and `delete` (bottom) operations [12], which are among the most dominant operations performed on dynamic data structures [15]. The size of the corresponding data structure and its functionality are maintained constant across the different access patterns. To clarify the interpretation of Figure 2 further: the Y-axis represents the number of r/w operations for a given data structure and access patterns, which are tracked using our library's integrated logging mechanisms. These accesses correspond to read and write accesses made to every data structure variable during a get, insert, or erase operation under various examined access patterns (e.g., random gets, sequential inserts, mixed operations, etc.). The accesses are logged for all the data structure elements, thus providing an in-depth monitoring at the application-level. While not true memory accesses these data accesses act as an input stimulus or workload profile. This data provides a preliminary approximation of the memory traffic generated by different data structure variants. By utilizing this information, we perform an initial filtering step, identifying candidates with favorable access patterns. The major

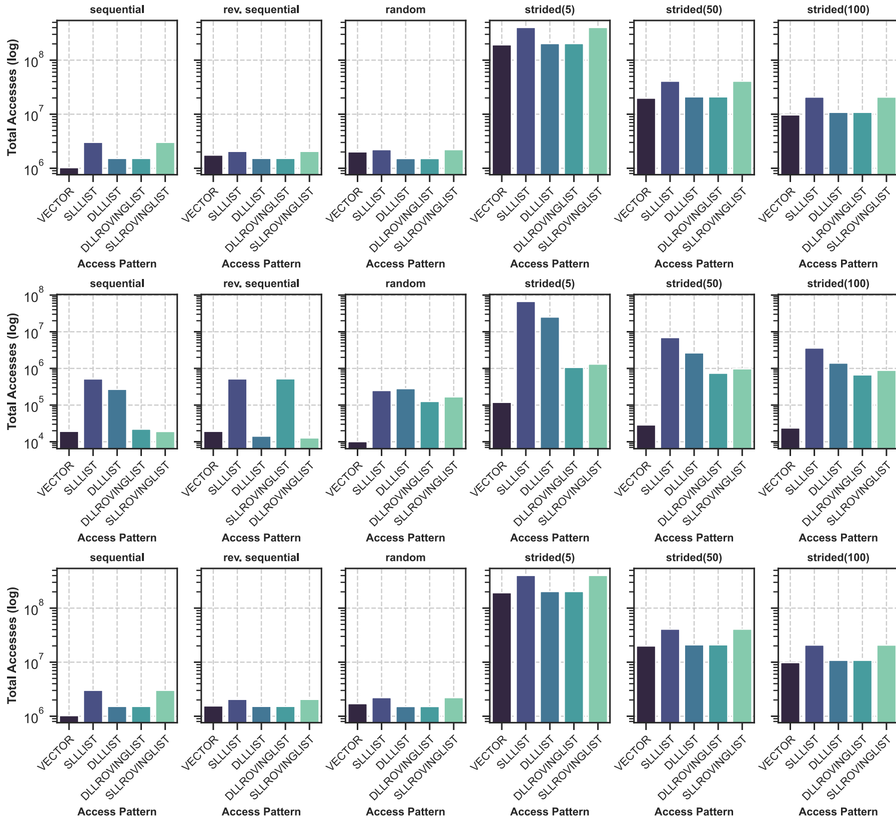


Fig. 2. Number of total application-level accesses of the dynamic data structures for the examined access patterns for **insert** (top), **get** (middle) and **delete** (bottom) operations.

observations of our experiments are the following: (i) for the `insert` (top) and `delete` (bottom) operations, `Vector` outperforms the other implementations for `sequential` and `strided` accesses patterns, while `SLL/DLL` minimize the accesses when the pattern is `reverse sequential` or `random`; (ii) for the get (middle) operations we observe that for the `reverse sequential` access patterns the DLL(R) minimize the number of accesses, while for the `sequential` the SLLR is the optimal implementation. Even though the illustrated outcomes provide a strong guideline for effectively selecting the optimal implementation for minimizing the number of accesses, while significantly pruning the design space for dynamic data type refinement, there exist cases, where different implementations provide similar performance, thus further exploration is required. Last, as shown in Section 3.2, the examined data structures trade memory footprint for memory accesses, therefore, in our proposed methodology, we consider both memory accesses and memory footprint. Therefore the co-exploration of the corresponding metrics needs to be performed.

Finally, apart from the access pattern that the structures are processed, we also classify the data structures based on the way that read and write operations are performed. The distinction relies on the sequential/random behavior of the read and write operations performed on the corresponding data structure. Write randomness is critical for NVMs, as high level of randomness tailored on NVMs can lead to significant performance degradation [71]. Thus, characterizing the pattern of write accesses serves as a valuable high-level indicator, aiding in understanding data volatility and informing memory allocation strategies tailored to the specific characteristics of different data

types. We classify the data structures at **memory-level**, based on the sequentiality/randomness that the read/write operations are performed:

- ***Gather***: The gather access pattern typically involves random reads and sequential write operations [34]. This pattern is often encountered in scenarios such as data sorting algorithms, where elements are read from various locations in memory and then written to a new location in a sorted sequence. Efficient handling of this pattern involves optimizing read operations for random access while ensuring that write operations are performed sequentially to maximize memory throughput and minimize latency.
- ***Anti-Gather***: Similar to the *Gather* access pattern, the anti-gather involves sequential reads and random writes. Efficient handling of the anti-gather pattern involves optimizing read operations for sequential access while managing random write operations.

Unlike linear data structures like vectors or linked lists, where access patterns are more easily defined and analyzed, tree structures are inherently hierarchical and their access patterns can vary widely depending on factors such as the specific operations being performed (e.g., insertion, deletion, search), the shape and size of the tree, and the nature of the data being stored, thus the analysis of tree-based structures is performed for the next steps of the methodology. The outcome of the dynamic data type and access pattern analysis is propagated as input to our proposed methodology, presented in Section 4.2.

4.2 Proposed Methodology

The proposed methodology is illustrated in Figure 3 and has been designed to meet four objectives: (i) demonstrate improved results for data placement algorithms, enabled by application-level code optimizations, (ii) boost data structure profiling by limiting the expensive and resource-hungry system-level monitoring, (iii) evaluate placement solutions based on multiple objectives (performance, energy consumption, NVM lifetime) and (iv) provide usability, scalability, and extensibility features (e.g., reasonable exploration time even for relatively large design spaces and support for a variety of application domains, data placement algorithms and real/emulated NVM technologies). Moreover, our library is enhanced with the knowledge of the mapping of the optimal dynamic data type implementation to the corresponding pattern. The extended library is presented in detail in Section 4.2.1.

Our methodology consists of three distinct steps:

1. ***Architecture Independent Source Code Transformations***, which leverages design space exploration for performing source-to-source data-flow transformations at the application level that improve data organization by mapping the high-level optimization objectives to application-specific indicators. This step strongly relies on detailed source-code analysis, aiming to characterize the examined dynamic data types based on their access pattern. The *Architecture Independent Source Code Transformations* is detailed in Section 4.2.1.
2. ***Architecture Dependent Placement***, which aims to support data placement of the corresponding data structure implementations. For the different combinations of dynamic data organization and placement and through run-time monitoring, we monitor system-level metrics, which focus on the per-DIMM analysis of performance, energy, and memory accesses, both for DRAM and NVM DIMMs. Through these metrics, the overall performance, energy consumption and NVM lifetime are extracted, thus composing a multi-objective optimization problem. The *Architecture Dependent Placement* is analyzed in Section 4.2.2.
3. ***Decision Support***, which is the last step of our flow aims to extend existing placement policies and perform objective-weighted decision-making, aiming to support placement decisions based on different user-defined optimization thresholds. More specifically, based

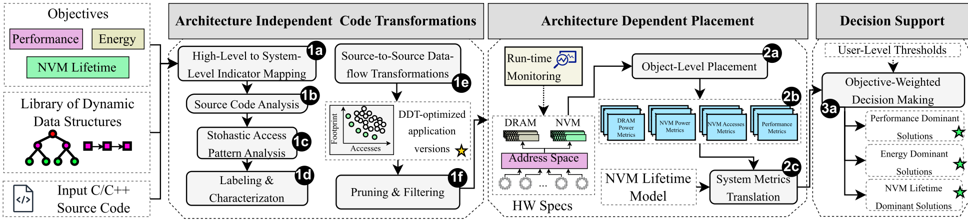


Fig. 3. Overview of the proposed methodology.

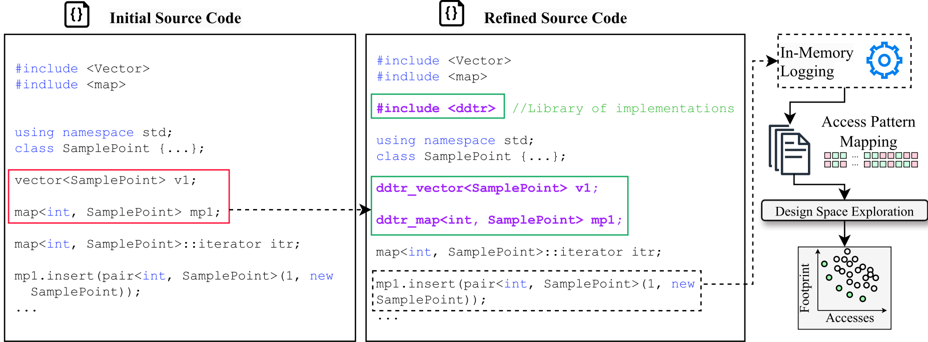


Fig. 4. Code snippet of the integration of dynamic data type library and exploration.

on varying thresholds for performance, energy, and NVM lifetime, the exploration is performed and the Pareto optimal solutions are derived. The *Decision Support* flow is analyzed in Section 4.2.3.

4.2.1 Architecture Independent Source Code Transformations. The first step of our proposed flow is the *Architecture Independent Source Code Transformations*, aiming to provide effective source code-level dynamic data type exploration and refinement. The overall objective of our methodology is the co-optimization of performance, energy consumption, and the NVM lifetime. These high-level optimization objectives are mapped to dynamic data type indicator metrics, that strongly affect the performance, energy and NVM wear (1a). From the perspective of dynamic data type organization, these metrics are mostly affected by the number of read and write accesses, as the latter strongly affect the overall NVM's behavior [13] and the memory footprint requirements, which can be the root cause of performance bottlenecks. More specifically, we utilize the overall number of read and write operations and the memory footprint requirements as key indicators for selecting the optimal implementations. Every defined dynamic data structure container is independently analyzed and transformed based on its unique access patterns and usage context. Our methodology retains the original STL-based function interfaces, ensuring seamless integration with existing operations, by treating each container as a separate entity and avoiding the unification of multiple container into a single implementation.

From a technical perspective, the interface of the library relies on STL, thus it can be directly integrated to applications source code which use STL-based data structures. In Figure 4 a simplified illustrative code snippet of the integration of the library is shown. The original data structures of the application source code are replaced with the ones of the library. Therefore, all operations (e.g., add, search, remove) are performed by the integrated library data structures. The library supports in-memory logging for dynamic data type monitoring, while, it further extends existing approaches [35, 39], by integrating a novel mechanism for supporting stochastic access pattern

prediction per dynamic data type, i.e., providing report about the access pattern of a dynamic data type with a specific level of confidence.

Next, the Source Code Analysis (1b) is performed. The dynamic data type refinement library is enhanced with an efficient access pattern logging mechanism. More specifically, we monitor the accesses performed per dynamic data type object, thus generating a sequence of read and write operations. This process is performed once, thus we minimize the overhead of analysis of discrete application versions and dynamic data type configurations, in contrast to previous approaches, such as Reference [39]. More specifically, each individual dynamic data type object is assigned a unique object identifier (*id*). Through this approach, we form a zero-shot [61] profiling, where the application is executed once, and the optimal data type implementations are derived from the stochastic access pattern analysis (1c). The design space is significantly pruned and we only provide exploration for the dynamic data types that either their access pattern is non-explainable/unpredictable, or that the outcome of the analysis does not have a clear optimal implementation.

Each access pattern is formed by two discrete sequences: (i) the sequence that the dynamic data type objects are accessed, based on their unique *id*s and (ii) the sequence of read (r) and write (w) operations performed in the corresponding dynamic data type/object. The former provides the knowledge of the way that the target data structure is accessed, while the latter enables the potential predictability of the linearity/randomness of read and write operations, which is dominant for detecting access patterns such as `Gather` and `Anti-Gather`. The sequence of object *id*s is sequentially traversed and the corresponding correlation of the consecutive elements is derived, thus mapping the access pattern to `Sequential` or `Reverse Sequential`. Moreover, the access patterns are targeted as strings consisting of r/w symbols and are mapped to the examined access patterns (Section 4.1) through solving a regular expression matching problem [21]. The access patterns (base access patterns) presented in Section 4.1 are formed as regular expressions [26]. Next, the generated access patterns of each data type from the input application are formulated as combination of the regular expressions of the base access patterns. The target access pattern is characterized based on the most dominant regular expression utilized for the construction. The access patterns are finally characterized based on the randomness of r/w operations, thus being mapped to either `Gather` or `Anti-Gather` patterns. Last, if no correlation can be derived based either on the *id* sequence or on the r/w sequence, then the access pattern is characterized as `Random` and further processing and exploration are required. This process enables the identification of distinct access patterns followed by automated analysis using bash scripting for efficient regular expression matching.

Once the stochastic access pattern analysis is performed, the different data structures are classified into *known*/*unknown* (1d). The former are the dynamic data types, who have already been mapped to an access pattern and their optimal implementations are known, while the latter correspond to those that have not been mapped and require further exploration. Subsequently, the *known* dynamic data structures are labeled with a unique *id* and are excluded from the exploration phase. The rest are characterized as *unknown* and are propagated to the next steps for further exploration.

Next, the source-to-source transformations step is performed (1e), where the application is transformed and executed once for each different combination of the data structure implementations shown in Table 1. During each execution, the number of data accesses and the memory footprint are monitored, based on profiling tools integrated in the library. Apparently, the number of combinations rises exponentially to the number of available data structure implementations. To prune the design space of combinations and reduce the exploration time, the data structures that are independent in the sense that are not affected by data accesses on other data structures are optimized separately. Thus, the exploration time becomes linear and the duration of the first

ALGORITHM 1: Design Space Pruning Mechanism

```
1: for pointA in Pareto Set do
2:   for pointB in Pareto Set do
3:     // Compute Euclidean Distance (ED)
4:     ED = (pointA.footprint - pointB.footprint)^2 + (pointA.accesses - pointB.accesses)^2
5:     if ED is below Threshold then
6:       remove(pointB) // Eliminate close neighbors
7:     end if
8:   end for
9: end for
```

step is significantly reduced. Moreover, all monitoring information (i.e., data accesses, memory footprint per data structure) is kept in memory instead of disk storage to minimize the overhead.

The output of the source-to-source data-flow transformation step is the Pareto curve of data accesses vs. memory footprint. Each point corresponds to an application version with the same functionality but different combination of data structure implementations. Aiming to further reduce the size of the design space, we implemented a pruning algorithm that removes Pareto points that are too close to each other, thus flagged as neighbor points (1f). Neighbor points are detected based on the Euclidean distance among all points in the design space. For those that their distance is less than a user-specified threshold, we keep a single point, as the neighbor points are expected to provide very similar results in terms of performance, energy consumption and impact on the NVM lifetime. While sophisticated strategies could offer more refined operating point distillation [62], the simplicity and low overhead of Euclidean distance make it a practical choice for our iterative optimization process. The retained design points are further refined during the design space exploration and decision-making phases to ensure that sub-optimal solutions are systematically excluded. The pruning heuristic algorithm is shown in Algorithm 1. The final set of Pareto optimal solutions is provided as input to the *Architecture Dependent Placement* step.

4.2.2 Architecture Dependent Placement. tThe data placement algorithms considered by the proposed methodology are the ones that rely on analytical models for decision-making [30, 49]. Typically, the models receive as input (i) profiling information per memory page, object or data structure and (ii) memory specifications, such as read/write latency, read/write energy consumption and memory capacity (2a). This information can be either obtained through manufacturer's reports or found in the existing literature [30]. The memory systems can be either real hardware platforms, such as DRAM/Optane DCPM or emulated for emerging NVM technologies.

Each application version is deployed on the heterogeneous memory system, based on the selected data placement algorithm, and is executed. To monitor the execution time, the energy consumption and the number of NVM writes, real-time monitoring is required. The energy consumption can be measured either based on relevant memory models (for the emulated systems) or by sensors installed on the DIMMs for real hardware (such as in Optane DPCM). Once the data placement is performed, we derive the following system-level metrics: (i) DRAM power metrics, (ii) NVM power metrics, (iii) NVM read/write accesses and (iv) the target application's execution time (2b). For monitoring the system-level metrics we utilized the Intel's **Performance Counter Monitor** (**PCM**) [8] and VTune Profiler [4]. The number of physical read/write accesses for Intel Optane DCPM are measured using the `ipmctl` tool. For emulated heterogeneous memory systems, emulators such as Reference [3] report the number of accesses on the corresponding memory technology.

Moreover, concerning the impact on the NVM's lifetime, we apply the approach described in Reference [60]. More specifically, we map the write accesses for each application's version to the

lifetime-related specs of the target NVM technology (2c). In particular, we rely on the number of write accesses throughout the execution time and by taking into consideration the lifetime-related specs of Optane DCPM (Petabytes Written—PWB), we obtain a high-level estimation of the NVM's remaining lifetime. The information derived by this step are propagated as input to the *Decision Support*, where the final decision making is performed, as presented in Section 4.2.3.

4.2.3 Decision Support. The *Decision Support* step aims to solve the multi-objective problem of deriving the Pareto Optimal solutions of the existing design space under performance, energy and NVM lifetime constraints to the end-users(3a). The Pareto optimal solutions represent the trade-offs between these axes, enabling us to find the best-performing solutions under user-defined constraints for each axis. As input, we provide the derived design space *DS* (Equation (2)) from the *Architecture Dependent Placement*, which consists of the different combinations of dynamic data type implementations *ddt* and placement alternatives *pl*. The Design Space *DS* (Equation (2)) corresponds to all the possible the mappings of each data type ddt_i to an implementation, and a mapping for each dynamic data type instance pl_j to either NVM or DRAM. For the ith dynamic data type implementations ddt_i and the jth data placement pl_j, a single solution s_i^j is derived, which form the overall *DS*. Each solution of the *DS* is characterized by a 3-dimensional vector $\{t,e,l\}$, consisting of its execution time t, energy consumption e and estimated NVM lifetime l, as derived by the run-time monitoring performed in *Architecture Dependent Placement* step.

Our optimization objective is presented in Equation (1) and targets the minimization of execution time t and energy consumption e and the maximization of the NVM's lifetime l. The aforementioned objectives form a multi-objective design space where tradeoffs need to be derived. In the *DS* we derive the optimal solution for each optimization objective, i.e., the optimal execution time t_{opt}, the optimal energy consumption e_{opt} and the optimal expected NVM lifetime l_{opt}, which are achieved through different combination of dynamic data type implementations and placements. The optimal solutions form a baseline. We form the corresponding constraints through defining the execution time, energy and NVM lifetime percentage degradation α, β and γ, respectively, as shown in Equation (3). The corresponding degradation values range from 0 to 1 (Equation (4)). Increasingly smaller values assigned to the constants α, β, and γ constrict the solution space by imposing stricter limitations on acceptable levels of execution time, energy consumption, and NVM lifetime degradation, respectively, thus narrowing the range of feasible solutions. Therefore, the target constants are derived through extensive experimentation.

The parameters α, β, and γ collectively formulate a weighted-objective problem, affording distinct degradation levels to each axis, thereby enabling prioritization across execution time, energy consumption, and NVM lifetime. The optimal solutions (t_{opt}, e_{opt}, and l_{opt}) become accessible to the user once the design space is composed, just prior to the *Decision Support* step. In real-world scenarios, the values of α, β, and γ can be tailored based on user-defined priorities, ensuring the solution aligns with specific performance, energy and resource requirements. Solving this optimization problem allows us to identify Pareto optimal solutions that balance execution time, energy consumption, and NVM lifetime, providing valuable insights for decision-making in heterogeneous memory systems. Thus, developers can identify tradeoffs between select the Pareto optimal solutions that meet their design constraints.

$$\text{objective:} \quad \left\{ \underset{s_i^j}{\text{Min}}\, t, \underset{s_i^j}{\text{Min}}\, e, \underset{s_i^j}{\text{Max}}\, l \right\} \quad \forall s_i^j = \{ddt_i, pl_j\} \in DS \tag{1}$$

$$\text{where} \quad DS = \left\{ s_i^j \mid s_i^j = (ddt_i, pl_j) \right\} \tag{2}$$

$$\text{s.t.} \quad t_{opt} \leq (1-\alpha) \times t, \quad e_{opt} \leq (1-\beta) \times e, \quad l \leq (1-\gamma) \times l_{opt} \tag{3}$$

$$\text{where} \quad \{\alpha, \beta, \gamma\} \in (0, 1] \tag{4}$$

ALGORITHM 2: Multi-Objective Optimization and Pareto Optimal Solutions

Data: Design Space $DS = \{s_i^j | s_i^j = \{ddt_i, pl_j\}\}$, Optimal values $t_{opt}, e_{opt}, l_{opt}$, Constraints $\alpha, \beta, \gamma \in (0, 1]$
Result: Pareto optimal solutions P

```
1   /* Step 1: Evaluate Design Space */
2   for s_i^j ∈ DS do
3       Compute t(s_i^j), e(s_i^j), l(s_i^j) ;                  // Performance metrics
4       Form v(s_i^j) = {t(s_i^j), e(s_i^j), l(s_i^j)} ;        // Vector representation
5   /* Step 2: Apply Constraints */
6   for s_i^j ∈ DS do
7       if t(s_i^j) ≤ (1 − α) · t_opt and e(s_i^j) ≤ (1 − β) · e_opt and l(s_i^j) ≥ (1 − γ) · l_opt then
8           P ← P ∪ {s_i^j} ;                                   // Add valid solution
9   ParetoFront ← ∅ ;                                           // Initialize Pareto front
10  /* Step 3: Derive Pareto Front */
11  for s ∈ P do
12      if there does not exist s′ ∈ P such that t(s′) ≤ t(s), e(s′) ≤ e(s), l(s′) ≥ l(s) and t(s′) ≠ t(s) or
          e(s′) ≠ e(s) or l(s′) ≠ l(s) then
13          ParetoFront ← ParetoFront ∪ {s} ;                   // Add non-dominated solution
14  return ParetoFront
```

The detailed decision support is presented in Algorithm 2. Our solution begins with initializing an empty set to store valid solutions. Each solution in the design space is evaluated by calculating its execution time, energy consumption, and NVM lifetime, represented as a three-dimensional vector (lines 3–5). Solutions that meet the constraints, defined as bounds on acceptable degradation in execution time, energy consumption, and NVM lifetime, are added to the valid set (lines 7–9). From this filtered set, the Pareto front is derived by identifying solutions that are not dominated by others, i.e., they offer optimal tradeoffs across all objectives (lines 12–15). The resulting Pareto front represents the best-balanced solutions, enabling informed decision making by highlighting tradeoffs and guiding the selection process under the given constraints.

5 Experimental Evaluation

5.1 Experimental Setup

The proposed methodology was evaluated based on the following criteria: (i) The impact of the high-level data organization optimization methodologies, such as dynamic data structures exploration, on the results of data placement, in terms of performance, energy consumption and impact on NVM lifetime, (ii) The extent to which the methodology can efficiently integrate various data placement algorithms with different optimization objectives, and various memory technologies (both emulated or real-hardware) and (iii) The scalability of the methodology in terms of exploration time, when applied to applications with a relatively large number of data structures.

The experiments were conducted on a high-end server integrating a 2x20 core Intel Xeon Gold 5218R CPU@2.10GHz with 4x32GB DDR4 DIMMs and 6x256GB Optane DC NVDIMMs. Intel **Memory Latency Checker** (**MLC**) [6] was utilized to monitor Optane DCPM latency. Optane DCPM was configured in App-Direct mode with EXT4-DAX filesystem. Version 1.11 of the **Persistent Memory Development Kit** (**PMDK**) [9] and gcc-9.4 were employed for the deployment of the applications on the DRAM/Optane DCPM system. Moreover, we relied on HP Quartz emulator [3] to emulate the PCM, RRAM and STT-MRAM technologies. Even

Table 3. Overview of the Applications Used for the Evaluation of the Methodology

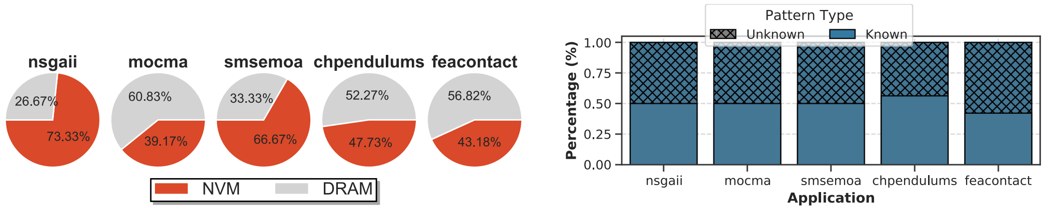
Application Library	Application Name	Short Description	Data Structures
Shark-ML Library	NSGA-II	Fast and elitist multi-objective genetic algorithm	2 VECTORS & 1 MAP
	Steady-State MOCMA	Steady-State Multi-Objective Selection in Covariance Matrix Adaptation	4 VECTORS & 1 MAP
	SMS-EMOA	Multi-objective selection based on dominated hypervolume	2 VECTORS & 1 MAP
Chrono Physics Engine	CH-PENDULUMS	Simulation of movement of various pendulum objects	19 VECTORS
	FEA-Contact	Finite element analysis(FEA) of pendulums interaction	16 VECTORS & 1 MAP

though Intel Optane relies on PCM technology, conducting experiments on both PCM and Intel Optane is essential we can deploy discrete characteristics in terms of read/write access latency and bandwidth for the same technology, and application-specific suitability within the broader spectrum of non-volatile memory solutions. The latency and power specifications of each memory technology are reported in relevant literature [30], while the lifetime-related values for the Optane DCPM were obtained from the corresponding manufacturer's reports [7].

The methodology is evaluated based on applications from two domains: ML and physics-based simulations. Five representative applications were selected from the Shark ML Library [32] and the Chrono Physics engine [1]. Table 3 summarizes and briefly describes each application. Concerning the data placement, we implement two algorithms that operate at memory object granularity and focus on the optimization of performance and energy consumption (References [30] and [49]). The former relies on a performance and energy consumption model and the memory objects are placed so that the energy is minimized and latency is raised by no more than a fixed percentage over a DRAM-only system [30]. The later is based on a performance/energy model to direct the placement, after characterizing the access patterns of the objects at different phases of the execution [49]. The algorithm supports the runtime migration of objects. However, in the context of this work, we apply the initial object placement phase only. Additionally, we compare against the aforementioned memory management solutions and against [43], a DRAM/NVM memory management solution, which employs phase-based data placement and is implemented from scratch.

5.2 Evaluation

Placement Distribution and Access Pattern Mapping: We applied the methodology to all applications using the data placement algorithm described in Reference [49]. Before evaluating the impact of our approach in terms of performance, energy and NVM accesses, we first demonstrate the average distribution of the data structures over the heterogeneous memory system (both DRAM/Optane DCPM and emulated) for each application, respectively as illustrated in Figure 5(a). For instance, for the NSGA-II, it can be noticed that 26% of the data structures on average are placed to DRAM and 73% on the NVM, while for FEA Contact, the placement is more balanced. These results indicate that the applications exploit both memory layers effectively. By doing so, they achieve a balance that considers tradeoffs between various factors such as performance, energy consumption, and the impact on the NVM's lifetime. The ability to identify and manage these tradeoffs is crucial for optimizing the overall efficiency and longevity of the memory system in heterogeneous environments. Moreover, in Figure 5(b) we illustrate the percentage of the successfully classified access patterns to known and unknown for the different dynamic data structures for each application, respectively. We observe that on average the 49.6%



(a) Average distribution of the data structures of the Pareto optimal solutions over heterogeneous memory systems for each application.

(b) Percentage of successfully classified access patterns (Known) and non-classified (unknown) of the data structures for the examined applications.

Fig. 5. Average DRAM/NVM placement distribution (left) and percentage of classified and non-classified access patterns (right) for the examined application workloads.

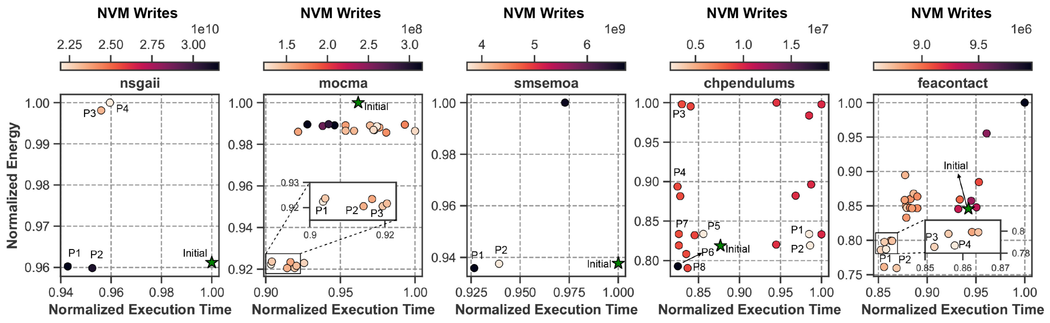


Fig. 6. Design space and performance/memory size/number of NVM writes Pareto optimal solutions for each application deployed on a DRAM/Optane DCPM system.

of the examined data structures is mapped to the access patterns presented in Section 4.1, thus effectively pruning the design space by half.

Multi-objective Analysis and Pareto Optimal Solutions for Intel Optane DCPM: Figure 6 shows the output of the methodology for each application. In particular, we illustrate the three-dimensional design space (performance/energy/NVM write accesses) and the Pareto optimal solutions for each application deployed on DRAM/Optane DCPM heterogeneous system. The initial version of the application (i.e., with the native data structure implementations) is highlighted in each figure and it is used as a baseline. Each point in the design space corresponds to a version of the application with the same functionality, but different data organization, and therefore different results in terms of performance, energy consumption and number of NVM write accesses.

A first observation is that **significant gains in terms of performance, energy consumption and reduction in the number of NVM writes are obtained for all applications, compared with the corresponding initial solution**. In particular, the execution time improves within a range of 4.1% (NSGA-II) to 9.7% (MOCMA) and the energy consumption from 0.15% (NSGA-II) to 10.1% (FEA-CONTACT). With regard to NVM writes, the reduction ranges from 0.8% (NSGA-II) up to 72.6% (CH-PENDULUMS). Finally, an interesting observation is that none of the initial versions of the applications belong to the Pareto optimal solutions in any of the experiments.

Table 4 shows details about some of the Pareto optimal solutions. In particular, the best solutions in terms of performance, energy consumption and number of NVM write accesses and corresponding data structure implementation of each one are shown for all applications in the table. Considering the initial solution as a baseline, we notice that **there exist application**

Table 4. Pareto Optimal Solutions for Each Application Deployed on a DRAM/Optane DCPM System

	Application	Version	Data Structures (DSs) & Placement	Normalized Execution Time	Normalized Energy Consumption	NVM Write Accesses
Shark-ML Library	NSGA-II	Initial	VEC(NVM), VEC(NVM), RB(NVM)	1.0	0.998	31840655
		Best Time (P1)	SLLR(NVM), VEC(NVM), AVL(DRAM)	**0.942(-5.8%)**	0.960(-3.8%)	31560946(-0.8%)
		Best Energy (P2)	DLLR(NVM), VEC(NVM), AVL(NVM)	0.952(-4.8%)	**0.959(-3.8%)**	30840925(-3.1%)
		Best Accesses (P4)	VEC(NVM), DLLR(NVM), AVL(DRAM)	0.959(-4.1%)	1.0(+0.2%)	**21960658(-31.02%)**
		Key Outcomes: Transforming 1st DS to SLLR/DLLR mostly affects time/energy, 2nd DS to DDLR affects accesses. Placement of 3rd DS (AVL) affects time/energy.				
	MOCMA	Initial	VEC(NVM), VEC(NVM), VEC(NVM), VEC(NVM), RB(NVM)	1.0	0.996	1649304
		Best Time & Accesses (P1)	SLLR(DRAM), SLL(NVM), SLLR(DRAM), SLL(DRAM), RB(DRAM)	**0.903(-9.7%)**	0.922(-7.3%)	**1224036(-25.7%)**
		Best Energy (P2)	SLLR(NVM), SLL(NVM), SLL(DRAM), VEC(DRAM), RB(DRAM)	0.914(-8.6%)	**0.920(-7.6%)**	1512045(-8.3%)
		Key Outcomes: Modifying all DSs to SLL(R) strongly affects time and accesses. Maintain 4th DS as VEC and place on DRAM boosts energy reduction.				
	SMS-EMOA	Initial	VEC(DRAM), VEC(DRAM), RB(NVM)	1.0	0.937	7074123
		Best Time & Energy (P1)	SLLR(DRAM), VEC(DRAM), AVL(NVM)	**0.926(-7.4%)**	**0.935(-0.15%)**	6725001(-4.9%)
		Best Accesses (P2)	SLL(NVM), VEC(DRAM), AVL(NVM)	0.939(-6.1%)	0.936(-0.1%)	**3720111(-47.4%)**
		Key Outcomes: 1st DS to SLL(R) and 3rd DS to AVL affect time/energy. Placement of 1st DS affects NVM accesses.				
Chrono Library	CH Pendulums	Initial	All VEC	0.876	0.818	8662956
		Best Time (P4)	2nd,3rd DS SLL(DRAM), 6th DDDST SLL(DRAM), 10th DS VEC(NVM)	**0.824(-5.9%)**	0.895(+9.4%)	8585696(-0.9%)
		Best Energy (P7)	2nd,3rd DS SLL(DRAM), 6th DS VEC(NVM), 10th DS VEC(NVM)	0.836(-4.5%)	**0.790(-3.4%)**	8748676(+0.9%)
		Best Accesses (P1)	10th DS to SLL(NVM)	0.984(+12.3%)	0.833(+1.8%)	**2372476(-72.6%)**
		Key Outcomes: Transforming 10th DS to SLL strongly reduces the number of NVM write accesses. 6th DS placement reduces energy consumption, 2nd and 3rd DSs affect the mainly the execution time.				
	FEA Contact	Initial	All VEC & RB(DRAM)	0.931	0.845	9412784
		Best Energy (P2)	1st DS VEC(DRAM), 11th DS SLL(NVM), AVL(NVM)	0.868(-6.7%)	**0.759(-10.1%)**	8700268(-7.5%)
		Best Time (P1)	1st DS SLL(NVM), 11th DS SLL(DRAM), AVL(NVM)	**0.852(-8.4%)**	0.785(-7.1%)	8663364(-7.9%)
		Best Accesses (P4)	1st DS SLL(DRAM), 11th DS SLLR(NVM), AVL(NVM)	0.857(-7.9%)	0.786(-6.9%)	**8573788(-9.7%)**
		Key Outcomes: Transformation of 1st DS to SLL and 2nd DS to SLL(R) boosts execution time/energy reduction. Effective placement boosts time or energy or accesses reduction.				

versions with alternative data organizations that provide better results in terms of all optimization objectives. This highlights the significance of improving the data structures of applications, before the deployment on a heterogeneous memory system. A representative example is the NSGA-II application: by changing the tree-based data structure from MAP-RB to MAP-AVL and placing it either on DRAM (P2) or on NVM (P4) and the first data structure from VEC to either SLLR (P1) or DLLR (P2) improves performance by 5.8% and reduces energy consumption by 3.8%, respectively. Even though these Pareto points already reduce the number of NVM write accesses, aiming for further reduction, solution P4 is the optimal in terms of the number of NVM write operations. Indeed, when the second data structure is implemented as DLLR instead of VEC, the number of write accesses on the NVDIMMs is reduced by 31% compared with the initial solution. CH-PENDULUMS is another representative case, in which changes in data structure implementations, result in a significant reduction in the number of NVM write accesses. The transformation of a data structure implementation from VEC to SLL placed in NVM, trades performance and energy consumption for number of NVM write accesses (72.6% reduction when selecting the solution P1).

Multi-objective Analysis and Pareto Optimal Solutions for PCM, RRAM and STT-MRAM: Figure 7 shows the output of the methodology when applied to the same applications, using the same data placement algorithm, but on NVM technologies emulated by HP Quartz:

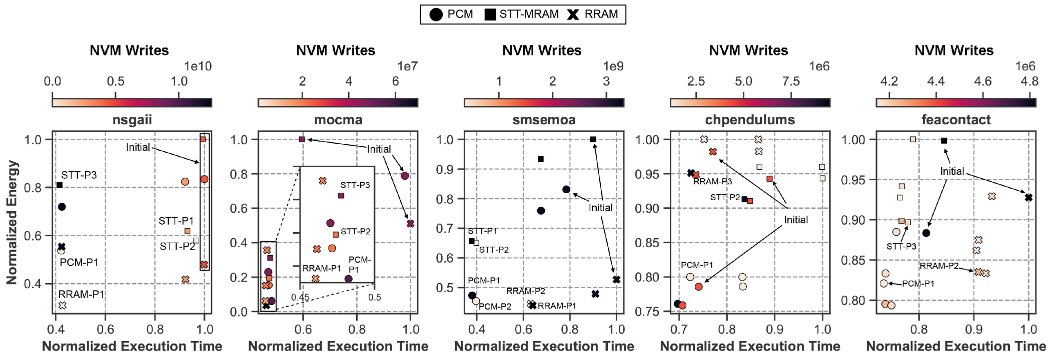


Fig. 7. Design space and performance/memory size/number of NVM writes Pareto optimal solutions for each application deployed on a DRAM/PCM, DRAM/RRAM and DRAM/STT-MRAM system.

PCM, RRAM and STT-MRAM. As in Figure 6, the X axis indicates the normalized execution time, while the Y axis shows the normalized energy consumption on the corresponding NVM. Each point represents a version of the application with alternative data organization deployed either on DRAM/PCM, DRAM/RRAM or DRAM/STT-MRAM system. Similar results with the DRAM/Optane DCPM system are obtained for these configurations as well: The execution time reduction ranges from 6.3% (CH-PENDULUMS STT-P2) to 58.7% (MOCMA RRAM-P1) compared with the initial solution. With regard to the energy consumption, the average reduction is 48.3% across all applications and memory technologies. Furthermore, the number of NVM write accesses is reduced by up to 73.4% (NSGAII RRAM-P1) compared with the initial version. These results demonstrate the applicability and effectiveness of the methodology on various emerging NVM technologies.

Table 5 provides more insights into the impact of the data structure optimization on the outcome of the data placement. We can observe that **there is no specific memory technology that provides optimal results in terms of performance or energy consumption for all applications**. For example, in NSGAII, the solution that yields highest performance is the application with the data structure implementation P3 deployed on a DRAM/STT system. However, for MOCMA, the corresponding optimal solution is the P1 deployed on DRAM/RRAM.

Another significant observation is that the same Pareto Optimal combination of data structure implementations may optimize different aspects depending on the underlying memory. For example, the solution STT-P3 for the NSGA-II, which corresponds to the first data structure implemented as DLLR, the second as VEC and the third as AVL, provides optimal results in terms of performance on a DRAM/STT system. Table 4 shows that the same combination of data structures for NSGA-II deployed on a DRAM/Optane DCPM system provides optimal results in terms of energy consumption. However, transforming the first data structure into SLLR slightly improves the performance on the DRAM/Optane DCPM. However, in other cases, the same combination of data structures provides optimized results in terms of the same metrics in various memory technologies. For example, in MOCMA, the solutions PCM-P1, RRAM-P1 (Table 5) and P1 on DRAM/Optane DCPM (Table 4) correspond to the same combination of data structures, with the same placement configuration and provide optimal results in terms of performance in all the examined memory technologies.

Evaluation over alternative placement policy: To demonstrate the effectiveness of the proposed methodology using alternative data placement algorithms, Figure 8 shows the output of the methodology for the MOCMA and CH-PENDULUMS on DRAM/Optane DCPM, when using the data placement algorithm of Reference [30]. Although this algorithm uses different performance

Table 5. Subset of Pareto Optimal Solutions for Each Application Deployed on a DRAM/PCM, DRAM/RRAM and DRAM/STT-MRAM System

Application	Solution	Data Structures (DSs) & Placement	Optimization Range
NSGA-II	STT, PCM, RRAM-P1	VEC(NVM), DLLR(NVM), RB(DRAM)	Performance:12.1%-58.7% Energy:18.2%-37.4% NVM Writes:13.2%-47.3%
	STT-P3	DLLR(NVM), VEC(DRAM), AVL(NVM)	
MOCMA	PCM, RRAM-P1	SLLR(DRAM), SLL(NVM), SLLR(DRAM), SLL(DRAM), RB(DRAM)	Performance:29.3%-34.4% Energy:62.3%-79.8% NVM Writes:6.3%-52.4%
	STT-P2	SLLR(NVM), VEC(NVM), SLLR(DRAM), DLL(DRAM), AVL(DRAM)	
SMS-EMOA	STT, PCM, RRAM-P1	SLLR(DRAM), VEC(NVM), RB(NVM)	Performance:13.2%-36.7% Energy:7.2%-34.3% NVM Writes:3.6%-69.8%
	STT, PCM, RRAM-P2	SLLR(DRAM), SLLR(DRAM), AVL(NVM)	
CH Pendulums	PCM-P1	2nd,3rd DS SLL(DRAM), 6th DS SLL(DRAM), 10th DS VEC(NVM)	Performance:5.1%-6.8% Energy:3.9%-5.1% NVM Writes:28.3%-50.8%
	STT-P2	2nd,3rd DS SLL(DRAM), 6th DS VEC(NVM), 10th DS VEC(NVM)	
	RRAM-P3	10th DS to SLL(NVM)	
FEA Contact	PCM-P1	1st DS VEC(DRAM), 11th DS SLL(NVM), AVL(NVM)	Performance:6.4%-9.7% Energy:7.1%-11.4% NVM Writes:12.5%-18.3%
	RRAM-P2	1st DS SLL(NVM), 11th DS SLL(DRAM), AVL(NVM)	
	STT-P3	1st DS SLL(DRAM), 11th DS SLLR(NVM), AVL,(NVM)	

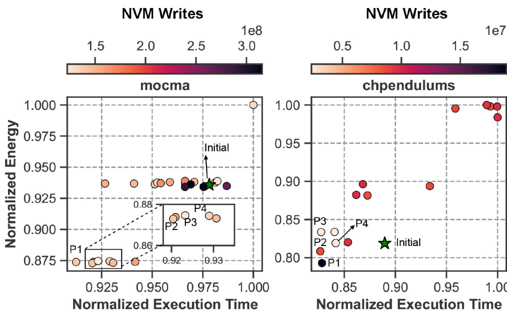


Fig. 8. Design space and performance/memory size/number of NVM writes Pareto optimal solutions for the MOCMA and CH-PENDULUMS applications for an alternative data placement algorithm.

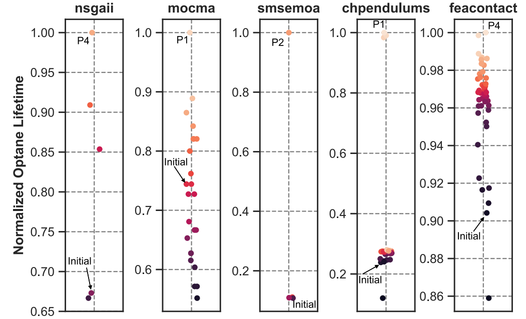


Fig. 9. Impact of data organization optimization and placement on the lifetime of the Optane DCPM DIMMs.

and energy models, being more energy-oriented, the results are relatively similar in terms of all metrics for both applications. This indicates that the placement decisions taken by the two algorithms do not differ significantly. Similar results have been observed for the rest of the applications.

Comparison with DRAM/NVM memory management solutions: Even though there exist prior research that propose data structure exploration [35], these are designed for optimizing data structures for DRAM-only architectures and do not provide support for heterogeneous architecture, thus they are not sufficient for heterogeneous memory systems. Thus, we focus on comparing our proposed solution with the DRAM/NVM memory management solutions that aim at providing performance and/or energy consumption optimization for DRAM/NVM memory systems. More

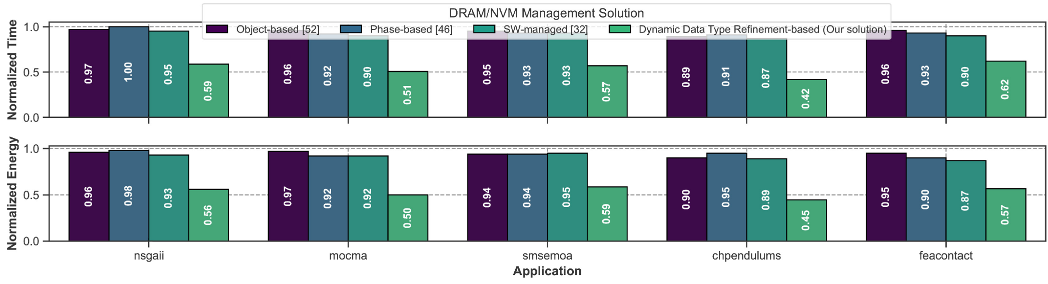


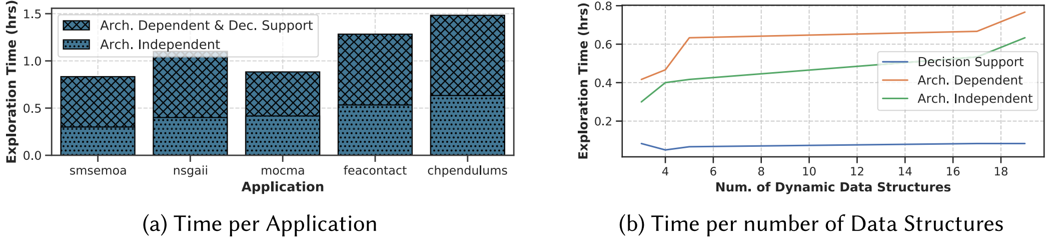
Fig. 10. Average Normalized execution time (top) and energy consumption (bottom) comparison of DRAM/NVM memory management solutions and the Dynamic Data Type Refinement-based placement over DRAM/Optane DCPM and emulated memory systems. We compare against the following DRAM/NVM memory management placement policies: Object-level [49], Phase-based [43] and SW-managed [30].

specifically, we compare with Reference [49] (Object-level), [30] (SW-managed) and [43] (Phase-based), which have been implemented from scratch. We compare the examined DRAM/NVM memory management policies with the initial version of data types, against the average of the Dynamic Data Type Refinement-based solution (i.e., our approach), which contains the refined data structures derived by our solution. Figure 10 illustrates the normalized execution time (top) and energy consumption (bottom) , in which we compare the examined DRAM/NVM memory management policies with the average placement execution time and energy consumption after the integration of our proposed dynamic data type refinement (DDTR) solution for the examined applications. We observe that the integration of our proposed solution outperforms existing approaches by achieving 48.9% less execution time and 47.3% less energy consumption on average. Our proposed solution benefits compared with existing solutions as application-level data structure re-organization boosts performance and reduces energy.

Impact on the NVM Lifetime: We investigate the impact of the data organization and placement policies on the underlying NVM's lifetime. We followed a similar approach to Reference [60], where the physical write accesses are projected to the underlying NVM technology, to evaluate the expected lifetime of the Optane DCPM. Figure 9 illustrates the projected NVM lifetime for the discrete application versions. As stated earlier, the NVM lifetime-related values for the Optane DCPM are obtained from the manufacturer's reported specifications [7], while the physical memory accesses are measured using the `ipmctl` tool. Moreover, in order to quantify the memory lifetime, we assume continuous operation over the Optane DIMMs with the write access rates shown in Figure 6. As shown also in Table 4, the application versions that maximize the Optane's lifetime are NSGAII P4, MOCMA P1, SMS-EMOA P2, CH-PENDULUMS P1, FEA-CONTACT P4. By selecting the aforementioned solutions, the lifetime of the Optane DCPM DIMMs is extended by 21.64% on average. Similar observations can be derived for the emulated memory technologies.

Methodology Overhead Analysis: Figure 12 shows the average memory footprint reduction for the Pareto optimal solutions for each application, respectively, compared with the initial solution. It can be noticed that the Pareto optimal solutions require 6.4% less memory size, on average. This is a positive side effect of the optimization of the data organization integrated as a step before the data placement on heterogeneous memories. In general, the reduced memory footprint may lower the stress on the memory layers and improve the sustainability of computing systems.

The complexity of the design space rises exponentially to the number of data structures under optimization. Therefore, we monitored the exploration time of the framework for the end-to-end optimization of each application. Figure 11 shows the exploration time required by each step of the methodology, for each application, in a stacked barplot. We observe that the total exploration time



(a) Time per Application

(b) Time per number of Data Structures

Fig. 11. Overall exploration time and scalability of the proposed methodology steps.

ranges from ≈48min. (SMS-EMOA) to ≈89 min. (CH-PENDULUMS). The exploration time depends on the execution time of the application under optimization and the number of data structures that need to be refined. However, even though the number of data structures increases (up to 19 data structures in CH-PENDULUMS) and the number of combinations that are evaluated rise exponentially, the exploration time does not increase exponentially. This is due to the fact that the data structures which are independent in the sense that are not affected by accesses of other structures are optimized separately, as explained in Section 4.2. The Dynamic Data Type Refinement acts as a pre-processing step for optimization, needs to be performed once and requires significantly less exploration time compared with prior research [35], which demands days of training. Furthermore, as the output of our proposed solution is an optimized source code tailored for the final deployment, the decision-making on the data placement phase requires zero run-time overhead compared with the other SoA memory management solutions [30, 43, 49].

Aiming to investigate more in-depth the overhead of our approach, we calculate the correlation between the number of data structures and the exploration time per step, illustrated in Figure 11(b). We observe that the exploration time rises linearly to the number of data structures under optimization. To quantify the linearity, we calculate the Pearson correlation [23] of the execution time and the number of data structures (closer to 1 indicates more linear correlation). For the discrete steps of our flow, i.e., *Architecture Independent Code Transformation*, *Architecture Dependent Placement* and *Decision Support*, the Pearson correlation is equal to 0.95, 0.85 and 0.98, respectively. Therefore, we provide near-linear exploration time to the number of data structures over an exponentially increasing design space. Thus, the exploration is scalable to a higher number of data structures.

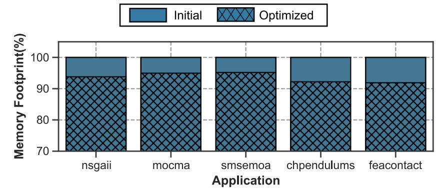


Fig. 12. Impact of the methodology on the application memory footprint: The Pareto optimal application versions have lower memory requirements compared with the initial versions.

Finally, regarding resource/memory overhead, our solution performs offline source code analysis and dynamic data type transformations for individual C files, generating in-memory logs specifically for STL dynamic data structures used in the examined application. This logging mechanism captures three primary metrics per data structure: read accesses, write accesses, and memory footprint, with a minimal overhead of 192 bytes per data structure. In the worst-case scenario, tracking 19 highly utilized data structures incurred an overhead of approximately 3.5KB, which is negligible given the available memory resources of modern HPC systems. For multi-core systems with parallel threads, the logging overhead remains manageable, peaking at 280KB in our experiments. Moreover, as logs are dynamically updated in real-time during execution and reset upon refinement completion, issues such as stale or obsolete logs are avoided, even with workload switching across cores.

5.3 Discussion, Limitations, and Future Work

In this section, we discuss factors that influence our overall methodology, outlining existing limitations and highlighting future directions for extending this research. Our proposed solution effectively boosts existing placement policies in terms of performance, energy and NVM lifetime, while ensuring scalability and minimal overhead. The proposed solution focuses on data type refinement and placement on heterogeneous memory systems, without, however, providing support for data migration. For applications that require frequent data migration, effective mechanisms need to be integrated, such as dedicated allocators and OS adaptations [49]. Furthermore, even though our library supports nested dynamic data structures (e.g., list of vectors), this research focuses on evaluating the characteristics of individual dynamic data structures, aiming to assess their key properties and behavior when deployed on DRAM or NVM [64]. Nesting different container types is not anticipated to significantly alter the high-level primitives outlined in Section 4.1, as nested containers can be decomposed into primitive data structures. However, access patterns produced by nested containers can potentially lead to dependencies, which affect the behavior of read and write operations performed on the underlying infrastructure. Thus, specified access pattern analysis should be conducted. Last, although nested container types are not explicitly addressed in this work, their inherent decomposability ensures that the proposed methods remain broadly applicable. By focusing on the fundamental primitives of individual dynamic data structures, we provide insights that can extend to complex nested configurations without loss of generality. Future work may explore these extensions in detail to confirm their scalability and performance in nested scenarios.

For future work, we aim at extending our research toward the following directions: (i) our proposed solution relies on optimizing existing data placement policies through dynamic data type refinement, thus we aim at integrating novel data placement policies for dynamic data structures; (ii) we aim at extending the functionality of our library by introducing support for nested container types; (ii) the impact of our proposed methodology under varying levels of discrete interference caused by multiple workloads running concurrently on the same node, potentially resulting in performance and memory bottlenecks will be additionally examined and (iii) developing support for dynamic data type migration at runtime, incorporating an analysis of the costs associated with migrating different dynamic data structure types between DRAM and NVM, and vice versa. Moreover, future work could explore the incorporation of more advanced operating point distillation methods, such as Reference [62], to enhance the precision and efficiency of design space pruning. Our solution can be adapted to different application domains, e.g., big data analytics, graph processing, and database, by refining our solution to accommodate domain-specific structures. In particular, big data and graph workloads, which rely on static table-based structures, would benefit from additional support for immutable data types, while for databases, the extension should focus on hash indexes, trees, and B+-trees. Last, our methodology offers a versatile foundation for wear-aware data management, adaptable to future NVM and CXL-based memory architectures, ensuring efficient data placement and sustained performance across evolving hardware landscapes.

6 Conclusion

This work proposes a methodology for improving the data placement results of applications deployed on DRAM/NVM heterogeneous systems, by integrating a data re-organization step at the application-level. Our solution relies on access pattern analysis, enhanced with a decision support flow for data placement that provides multi-objective design space exploration for co-optimizing performance, energy and NVM lifetime. The methodology is scalable and can integrate various data placement algorithms and NVM technologies, showing up to 58.7% lower execution time,

48.3% less energy consumption and 72.6% less NVM write operations compared with the results obtained by the initial versions of the applications. Developers can use it as a tool for co-design between applications with increased memory requirements and heterogeneous memory systems.

References

[1] 2024. Chrono Physics Engine. Retrieved from https://projectchrono.org/
[2] 2024. EXTRAE. Retrieved from https://tools.bsc.es/sites/default/files/documentation/extrae-3.2.1-user-guide.pdf
[3] 2024. HP Quartz Emulator. Retrieved from https://github.com/HewlettPackard/quartz
[4] 2024. Intel Vtune Profiler. Retrieved from https://software.intel.com/en-us/vtune
[5] 2024. Micron's Perspective on Impact of CXL on DRAM Bit Growth Rate. Retrieved from https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-impact-dram-bit-growth-white-paper.pdf
[6] 2024. MLC. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html
[7] 2024. Optane DC Specs. Retrieved from https://www.mouser.com/datasheet/2/612/optane-dc-persistent-memory-brief-1710301.pdf
[8] 2024. Performance Counter Monitor. Retrieved from https://github.com/opcm/pcm
[9] 2024. PMDK. Retrieved from https://github.com/pmem/pmdk
[10] 2024. Valgrind. Retrieved from https://valgrind.org/info/about.html
[11] David Atienza Alonso, Stylianos Mamagkakis, Christophe Poucet, Miguel Peón-Quirós, Alexandros Bartzas, Francky Catthoor, and Dimitrios Soudris. 2015. *Dynamic Memory Management for Embedded Systems*. Springer.
[12] Matthew H. Austern. 1998. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc.
[13] Amro Awad, Sergey Blagodurov, and Yan Solihin. 2016. Write-aware management of nvm-based memory extensions. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.
[14] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.
[15] Christos Baloukas, Jose L. Risco-Martin, David Atienza, Christophe Poucet, Lazaros Papadopoulos, Stylianos Mamagkakis, Dimitrios Soudris, J. Ignacio Hidalgo, Francky Catthoor, and Juan Lanchares. 2009. Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems. *Journal of Systems and Software* 82, 4 (2009), 590–602.
[16] Emery D. Berger. 2020. Scalene: Scripting-language aware profiling for python. arXiv:2006.03879. Retrieved from https://arxiv.org/abs/2006.03879
[17] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. 2017. Emerging NVM: A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 2 (2017), 1–32.
[18] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-Assisted Design for Architecture (AIDArc'19), Held in Conjunction with ISCA*.
[19] Heng Bu, Ming-Kai Dong, Ji-Fei Yi, Bin-Yu Zang, and Hai-Bo Chen. 2021. Revisiting persistent indexing structures on Intel Optane DC persistent memory. *Journal of Computer Science and Technology* 36 (2021), 140–157.
[20] Chen Lei, Zhao Jiacheng, Wang Chenxi, Cao Ting, Zigman John, Volos Haris, Mutlu Onur, Lv Fang, Feng Xiaobing, Xu Guoqing Harry, and Cui Huimin. 2022. Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories. *ACM Transactions on Computer Systems (TOCS)* 39, 1-4 (2022), 1–38.
[21] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 487–502.
[22] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 293–304.
[23] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise Reduction in Speech Processing* (2009), 1–4.
[24] Amin Farmahini-Farahani, Sudhanva Gurumurthi, Gabriel Loh, and Michael Ignatowski. 2018. Challenges of high-capacity dram stacks and potential directions. In *Proceedings of the Workshop on Memory Centric High Performance Computing*. 4–13.
[25] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.
[26] Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions*. "O'Reilly Media, Inc.".

[27] Yuhua Guo, Weijun Xiao, Qing Liu, and Xubin He. 2018. A cost-effective and energy-efficient architecture for die-stacked dram/nvm memory systems. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC'18)*. IEEE, 1–2.

[28] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, and Woongki Baek. 2019. Hotness-and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems. *IEEE Trans. Comput.* 69, 3 (2019), 377–391.

[29] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 1919–1928.

[30] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2015. Software-managed energy-efficient hybrid DRAM/NVM main memory. In *ACM CF*.

[31] Michael Hennecke. 2020. Daos: A scale-out high performance storage stack for storage class memory. *Supercomputing Frontiers* 40 (2020).

[32] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. 2008. Shark. *Journal of Machine Learning Research* 9, 6 (2008), 993–996.

[33] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2010), 105–118.

[34] Jim Jeffers, James Reinders, and Avinash Sodani. 2016. Chapter 10 - vectorization advisor. In *Intel Xeon Phi Processor High Performance Programming (Second Edition)* (second edition ed.), Jim Jeffers, James Reinders, and Avinash Sodani (Eds.). Morgan Kaufmann, Boston, 213–250. DOI: https://doi.org/10.1016/B978-0-12-809194-4.00010-7

[35] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: Effective selection of data structures. *ACM SIGPLAN Notices* 46, 6 (2011), 86–97.

[36] Saeed Kargar and Faisal Nawab. 2021. Extending the lifetime of NVM: Challenges and opportunities. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3194–3197.

[37] Manolis Katsaragakis, Christos Baloukas, Lazaros Papadopoulos, Verena Kantere, Francky Catthoor, and Dimitrios Soudris. 2022. Energy consumption evaluation of optane DC persistent memory for indexing data structures. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC'22)*. IEEE, 75–84.

[38] Manolis Katsaragakis, Dimosthenis Masouros, Lazaros Papadopoulos, Francky Catthoor, and Dimitrios Soudris. 2023. On the implications of heterogeneous memory tiering on spark in-memory analytics. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'23)*. IEEE, 945–952.

[39] Manolis Katsaragakis, Lazaros Papadopoulos, Christos Baloukas, and Dimitrios Soudris. 2022. Memory management methodology for application data structure refinement and placement on heterogeneous DRAM/NVM systems. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE'22)*. IEEE, 748–753.

[40] Manolis Katsaragakis, Konstantinos Stavrakakis, Dimosthenis Masouros, Lazaros Papadopoulos, and Dimitrios Soudris. 2023. Adjacent LSTM-based page scheduling for hybrid DRAM/NVM memory systems. In *14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'23)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[41] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-defined memory tiering for heterogeneous computing systems with CXL memory expander. *IEEE Micro* 43, 2 (2023), 20–29.

[42] Taeuk Kim, Safdar Jamil, Joongeon Park, and Youngjae Kim. 2020. Optimizing heap memory object placement in the hybrid memory system with energy constraints. *IEEE Access* 36, 8 (2021), 140–157.

[43] Jannis Klinkenberg, Clément Foyer, Pierre Clouzet, Brice Goglin, Emmanuel Jeannot, Christian Terboven, and Anara Kozhokanova. 2024. Phase-based data placement optimization in heterogeneous memory. In *CLUSTER 2024-International Conference on Cluster Computing*. IEEE.

[44] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, 256–267.

[45] Robert Lasch, Robert Schulze, Thomas Legler, and Kai-Uwe Sattler. 2021. Workload-driven placement of column-store data structures on DRAM and NVM. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN'21)*. 1–8.

[46] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[47] Bolun Li, Qidong Zhao, Shuyin Jiao, and Xu Liu. 2023. DroidPerf: Profiling memory objects on android devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. 1–15.

[48] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[49] Haikun Liu, Renshan Liu, Xiaofei Liao, Hai Jin, Bingsheng He, and Yu Zhang. 2020. Object-level memory allocation and migration in hybrid memory systems. *IEEE Trans. Comput.* 69, 9 (2020), 1401–1413.

[50] Wei Liu, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2021. Hngraph: Parallel graph processing in hybrid memory based numa systems. In *2021 IEEE International Conference on Cluster Computing (CLUSTER'21)*. IEEE, 388–397.

[51] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 757–773.

[52] Mohammad Alaul Haque Monil, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. Understanding the impact of memory access patterns in intel processors. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC'20)*. IEEE, 52–61.

[53] Diego Moura, Vinicius Petrucci, and Daniel Mosse. 2021. Learning to rank graph-based application objects on heterogeneous memories. In *The International Symposium on Memory Systems*. 1–14.

[54] Adhish Nanda, Swati Gupta, and Meenu Vijrania. 2019. A comprehensive survey of OLAP: Recent trends. In *2019 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA'19)*. IEEE, 425–430.

[55] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 41–54.

[56] M. Ben Olson, Brandon Kammerdiener, Michael R. Jantz, Kshitij A. Doshi, and Terry Jones. 2022. Online application guidance for heterogeneous memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 3 (2022), 1–27.

[57] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*. 1–7.

[58] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*. 288–303.

[59] Ivy Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. 2020. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS'20)*. IEEE, 916–925.

[60] Lillian Pentecost, Alexander Hankin, Marco Donato, Mark Hempstead, Gu-Yeon Wei, and David Brooks. 2022. NVM-Explorer: A framework for cross-stack comparisons of embedded non-volatile memories. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 938–956.

[61] Eliseu Pereira, João Reis, Rosaldo JF Rossetti, and Gil Gonçalves. 2024. A zero-shot learning approach for task allocation optimization in cyber-physical systems. *IEEE Transactions on Industrial Cyber-Physical Systems* 2 (2024), 90–97.

[62] Behnaz Pourmohseni, Michael Glaß, and Jürgen Teich. 2017. Automatic operating point distillation for hybrid mapping methodologies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1135–1140.

[63] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, 598–611.

[64] Thomas Rupprecht, Xi Chen, David H. White, Jan H. Boockmann, Gerald Lüttgen, and Herbert Bos. 2017. DSIbin: Identifying dynamic data structures in C/C++ binaries. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, 331–341.

[65] Shyh-Shyuan Sheu, Kuo-Hsing Cheng, Meng-Fan Chang, Pei-Chia Chiang, Wen-Pin Lin, Heng-Yuan Lee, Pang-Shiu Chen, Yu-Sheng Chen, Tai-Yuan Wu, Frederick T. Chen, et al. 2010. Fast-write resistive RAM (RRAM) for embedded applications. *IEEE Design & Test of Computers* 28, 1 (2010), 64–71.

[66] Shigeru Shiratake. 2020. Scaling and performance challenges of future DRAM. In *2020 IEEE International Memory Workshop (IMW'20)*. IEEE, 1–3.

[67] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. 2016. An effective dram cache architecture for scale-out servers. *Technical Report* (2016).

[68] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–362.

[69] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[70] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2020. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 5, 1 (2020), 1–28.

[71] Jialiang Zhang, Nicholas Beckwith, and Jing Jane Li. 2021. Gordon: Benchmarking optane dc persistent memory modules on fpgas. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'21)*. IEEE, 97–105.

Received 1 October 2024; revised 27 March 2025; accepted 30 April 2025