

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and Information Engineering

KU LEUVEN

KU Leuven
Electrical Engineering
Doctor of Engineering Science

Optimized Dynamic Memory Management Techniques on Heterogeneous Memory Systems

Ph.D. Thesis
of
Emmanouil Katsaragakis

Supervisors: Dimitrios Soudris (NTUA)
Francky Catthoor (KUL)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

Athens, June 2024

Copyright ©Emanouil Katsaragakis, 2024.
All rights reserved.



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής και
Συστημάτων Πληροφορικής

KU LEUVEN

Καθολικό Πανεπιστήμιο του Λέουβεν
Σχολή Ηλεκτρολόγων Μηχανικών
DOCT Επιστήμης Μηχανικης

Βελτιστοποιημένες Τεχνικές Δυναμικής Διαχείρισης Μνήμης για Ετερογενή Συστήματα Μνήμης

Διδακτορική Διατριβή
του
Εμμανουήλ Κατσαραγάκη

Επιβλέποντες: Δημήτριος Σούντρης (ΕΜΠ)
Francky Catthoor (KUL)

Αθήνα, Ιούνιος 2024



National Technical University of Athens
School of Electrical & Computer Engineering
Division of Communication, Electronic and
Information Engineering

KU LEUVEN

KU Leuven
Electrical Engineering
Doctor of Engineering Science

Optimized Dynamic Memory Management Techniques on Heterogeneous Memory Systems

Ph.D. Thesis
of
Emmanouil Katsaragakis

Supervisory Committee: Dimitrios Soudris, Francky Catthoor, Mario Konijnenburg, Panayiotis Tsanakas, Kiamal Pekmestzi

Approved by the advisory committee on June, 2024.

.....	.....	.....
Dimitrios Soudris	Francky Catthoor	Kiamal Pekmestzi
Professor NTUA	Professor KUL	Emeritus Professor NTUA
.....	.....	.....
Panayiotis Tsanakas	Mario Konijnenburg	Rudy Lauwereins
Professor NTUA	Senior Researcher IMEC	Professor KUL
.....	.....	.....
Sotirios Xydis	Ioanna Roussaki	George Theodoridis
Assistant Professor NTUA	Professor NTUA	Associate Professor UP

Athens, June 2024

.....
Emmanouil Katsaragakis
Doctor of Electrical and Computer Engineering NTUA
Doctor of Engineering Science KUL

Copyright ©Emmanouil Katsaragakis, 2024.
All rights reserved.

This work is partially supported by the European Commission H2020 EXA2PRO project.
It is forbidden to copy, store, and distribute this work, in whole or in part, for commercial purposes. Reproduction, storage, and distribution are permitted for non-profit, educational or research purposes, provided that the source is referenced and this message is retained. Questions concerning the use of this work for profit should be addressed to the writer. Content that is reused from publications that the author has (co-)authored (excerpts, figures, tables, etc.) is under copyright with the respective paper publishers (IEEE, ACM, etc) and is cited accordingly in the current text. Content that reused from third-party publications appears with the appropriate copyright note. Reuse of such content by any interested party requires the publishers' prior consent, according to the applicable copyright policies. Content that has not been published before is copyrighted jointly as follows:

2024 - Electrical & Computer Engineering NTUA
2024 - Faculty of Engineering Science KUL

Extended Abstract

Memory management in recent systems refers to the process of controlling, coordinating, and organizing computing platforms main memory resources. The primary and major goal of memory management is to provide efficient and effective utilization of the computing platforms memory, enabling programs and data to be stored, accessed, and manipulated through a systematic process. However, the advances in processing, memory and interconnection technologies over the last decade keep increasing the disparity of performance among the diverse processing cores, e.g. Compute Processing Units (CPUs), Graphics Processing Units (GPUs), RISC-V and ARM cores and Tensor Processing Units (TPUs) and the memory technologies and interconnection protocols, known as the "*Memory Wall*" and the "*Interconnection Wall*". Furthermore, in addition the ever-increasing amount and dynamicity of data produced, alongside with the integration of emerging memory technologies compose a complex heterogeneous paradigm. In this context, dynamic memory management plays a significant role in modern computing systems, allowing efficient allocation and deallocation of memory resources at runtime. In the state-of-the-art, dynamic memory management encompasses a variety of techniques and algorithms aimed at optimizing memory usage and improving overall system performance, sustainability and hardware wear. Aiming to satisfy the aforementioned criteria, dynamic memory management solutions require an effective co-design of multiple components of the hierarchy of the computing continuum, i.e. varying from application level to hardware-based solutions.

In this dissertation we examine the applicability of dynamic memory management techniques over distinct levels across the hardware-software continuum, aiming to provide performance, energy, and memory resources Pareto-tradeoff optimizations. More specifically, the novel contributions of this thesis focus on the following optimization levels: i) application-level optimizations, ii) OS support, libraries and placement optimizations, and iii) hardware-level mapping optimizations. Given the inseparable and strong relationship between hardware and software memory components, we aim to enhance the interoperability of different optimization layers and provide an end-to-end workflow and methodology. In summary, the major contributions of this PhD thesis are the following: (i) we present an extensive analysis, classification and we provide guidelines for data placement design for heterogeneous memory systems, (ii) we propose a novel solution

for applying source-to-source transformations for Python applications aiming to support memory footprint optimization, leveraging a set of static (offline) and dynamic (online) analysis optimizations, (iii) we design a dynamic data structure refinement, exploration and placement solution over heterogeneous DRAM/NVM Memory systems aiming to optimize and derive trade-offs among performance, energy consumption, and hardware wear and (iv) we propose near-hardware page placement solution which relies on LSTMs and the boosting of dynamic memory management through effective cache configuration.

Keywords: Dynamic Memory Management, Heterogeneous Memory Systems, DRAM, NVM, High Performance Computing

Εκτεταμένη Περίληψη

Η διαχείριση μνήμης σε σύγχρονα υπολογιστικά συστήματα αναφέρεται στη διαδικασία ελέγχου, συντονισμού και οργάνωσης των κυρίων πόρων μνήμης υπολογιστικών πλατφορμών. Ο βασικός και κύριος στόχος της διαχείρισης μνήμης είναι η αποδοτική και αποτελεσματική χρήση της μνήμης των υπολογιστικών πλατφορμών, επιτρέποντας σε προγράμματα και δεδομένα να αποθηκεύονται, να έχουν πρόσβαση και να υποβάλλονται σε επεξεργασία μέσω ενός συστηματικού επεξεργαστικού. Ωστόσο, οι προόδοι στις τεχνολογίες επεξεργασίας, μνήμης και διασύνδεσης της τελευταίας δεκαετίας συνεχίζουν να αυξάνουν τη διαφορά στην απόδοση μεταξύ των διαφορετικών πυρήνων επεξεργασίας, όπως οι μονάδες επεξεργασίας υπολογισμού (CPUs), μονάδες επεξεργασίας γραφικών (GPUs), πυρήνες RISC-V και ARM και μονάδες επεξεργασίας τανυστικών μονάδων (TPUs) και οι τεχνολογίες μνήμης και πρωτόκολλα διασύνδεσης, γνωστά ως "Memory Wall" και "Interconnection Wall". Επιπλέον, η αυξανόμενη ποσότητα και δυναμικότητα των παραγόμενων δεδομένων, σε συνδυασμό με την ολοκλήρωση νέων τεχνολογιών μνήμης, δημιουργούν ένα πολύπλοκο παράδειγμα παραλληλισμού. Σε αυτό το πλαίσιο, η δυναμική διαχείριση μνήμης διαδραματίζει σημαντικό ρόλο στα μοντέρνα υπολογιστικά συστήματα, επιτρέποντας την αποτελεσματική κατανομή και αποδέσμευση πόρων μνήμης κατά τη διάρκεια εκτέλεσης. Η δυναμική διαχείριση μνήμης περιλαμβάνει μια ποικιλία τεχνικών και αλγορίθμων που στοχεύουν στη βελτιστοποίηση της χρήσης της μνήμης και στη βελτίωση της συνολικής απόδοσης του συστήματος, της βιωσιμότητας και της φθοράς του υλικού. Με στόχο την ικανοποίηση των παραπάνω κριτηρίων, οι λύσεις δυναμικής διαχείρισης μνήμης απαιτούν αποτελεσματική συνδυαστική σχεδίαση πολλαπλών στοιχείων της ιεραρχίας του εύρους της υπολογιστικής συνέχειας, δηλαδή από το επίπεδο εφαρμογής έως τις λύσεις που βασίζονται στο υλικό.

Σε αυτή τη διατριβή εξετάζουμε την εφαρμοσιμότητα τεχνικών δυναμικής διαχείρισης μνήμης σε διαφορετικά επίπεδα στο εύρος του υλικού-λογισμικού, με στόχο τη βελτιστοποίηση της απόδοσης, της ενέργειας και των πόρων μνήμης. Πιο συγκεκριμένα, οι νέες συνεισφορές αυτής της διατριβής επικεντρώνονται στα εξής επίπεδα βελτιστοποίησης: i) βελτιστοποιήσεις στο επίπεδο εφαρμογής, ii) υποστήριξη λειτουργικού συστήματος, βιβλιοθήκες και βελτιστοποιήσεις τοποθέτησης, και iii) βελτιστοποιήσεις στο επίπεδο σχεδιαστικών αποφάσεων του υλικού. Δεδομένης της αναπόσπαστης και ισχυρής σχέσης μεταξύ των υλικών και λογισμικών στοιχείων μνήμης, στοχεύουμε στη βελτίωση της συμβατότητας των διαφορετικών επιπέδων βελτιστοποίησης και στην παροχή μιας συνολικής διαδικασίας εργασίας

και μεθοδολογίας. Συνοψίζοντας, οι κύριες συνεισφορές αυτής της διατριβής διατυπώνονται ως εξής: (i) παρουσιάζουμε μια εκτενή ανάλυση, ταξινόμηση και παρέχουμε οδηγίες για το σχεδιασμό τοποθέτησης δεδομένων για ετερογενή συστήματα μνήμης (ii) προτείνουμε μια νέα λύση για την εφαρμογή μετασχηματισμών κώδικα για εφαρμογές Python με στόχο την υποστήριξη βελτιστοποίησης της επίδοσης μνήμης, εκμεταλλευόμενη ένα σύνολο από βελτιστοποιήσεις στατικών και δυναμικών τεχνικών ανάλυσης, (iii) σχεδιάζουμε μια λύση διαμόρφωσης, εξερεύνησης και τοποθέτησης δυναμικών δομών δεδομένων σε ετερογενή συστήματα μνήμης DRAM/NVM με στόχο τη βελτιστοποίηση και την εύρεση ισορροπιών μεταξύ απόδοσης, κατανάλωσης ενέργειας και φθοράς υλικού και (iv) προτείνουμε λύσεις για τοποθέτηση σελιδών που βασίζεται σε LSTMs σε ετερογενή συστήματα, καθώς επίσης και για την ενίσχυση της δυναμικής διαχείρισης μνήμης μέσω αποτελεσματικής διαμόρφωσης cache.

Λέξεις Κλειδιά: Δυναμική Διαχείριση Μνήμης, Ποικίλες Συστοιχίες Μνήμης, Συστήματα DRAM, NVM, Υπολογισμοί Υψηλής Απόδοσης

Uitgebreide Samenvatting

Geheugenbeheer in recente systemen verwijst naar het proces van controleren, coördineren en organiseren van de belangrijkste geheugenbronnen van computerplatforms. Het primaire en belangrijkste doel van geheugenbeheer is het bieden van efficiënt en effectief gebruik van het computerplatformgeheugen, waardoor programma's en gegevens via een systematisch proces kunnen worden opgeslagen, toegankelijk gemaakt en gemanipuleerd. De vooruitgang op het gebied van verwerking, geheugen en interconnectietechnologieën heeft de afgelopen tien jaar de ongelijkheid in prestaties tussen verschillende verwerkingskernen zoals Compute Processing Units (CPUs), grafische Processing Units (GPUs), RISC-V- en ARM-kernen en Tensor Processing Units (TPUs) vergroot. Daar bovenop, zijn de uitdagingen in de realizaties van de geheugenorganisatie en de communicatientwerken sterk vergroot. Deze laatste zijn bekend als samen met de geheugentechnologieën en interconnectieprotocollen, bekend als de "Memory Wall" en de "Interconnectiemuur". Bovendien, vormen steeds toenemende volume en de dynamiek van geproduceerde gegevens, samen met de integratie van opkomende geheugentechnologieën, een complex en heterogeen ontwerpruimte. In deze context speelt dynamisch geheugenbeheer een belangrijke rol in moderne computersystemen, waardoor efficiënte toewijzing en deallocatie van geheugenbronnen tijdens runtime mogelijk is. In de literatuur omvat dynamisch geheugenbeheer een verscheidenheid aan technieken en algoritmen die gericht zijn op het optimaliseren van geheugengebruik en het verbeteren van de algehele systeemprestaties, duurzaamheid en hardware slijtage. Om aan de genoemde criteria te voldoen, vereisen dynamische geheugenbeheeroplossingen een effectief gezamenlijk ontwerp van meerdere componenten van het computercontinuüm, variërend van toepassingsniveau tot hardware-architectuur.

In dit proefschrift onderzoeken we de toepasbaarheid van dynamische geheugenbeheertechnieken over verschillende niveaus in het hardware-software continuüm, met als doel optimalisatie van prestaties, energie en geheugenbronnen te bieden. Meer specifiek richten de bijdragen van dit proefschrift zich op de volgende optimalisatieniveaus: i) optimalisaties op toepassingsniveau, ii) OS-ondersteuning, bibliotheken en plaatsingsoptimalisaties, en iii) hardwareoptimalisaties voor mappingniveaus. Gezien de onafscheidelijke en sterke relatie tussen hardware- en softwaregeheugencomponenten, streven we ernaar de interoperabiliteit tussen verschillende optimalisatielagen te verbeteren en bieden we een globale

workflow en methodologie.

Samenvattend zijn de belangrijkste bijdragen van dit proefschrift als volgt: (i)we presenteren een uitgebreide analyse, classificatie en we bieden richtlijnen voor het ontwerp van dataplaatsing voor heterogene geheugensystemen, (ii) we stellen een nieuwe oplossing voor om bron-naar-bron-transformaties toe te passen op Python-applicaties met als doel de geheugenvoetafdruk te optimaliseren, door middel van een reeks statische (offline) en dynamische (online) analyse-optimalisaties, (iii) ontwerpen we een oplossing voor dynamische datastructuurverfijning voor het zoeken, verkennen en plaatsen via heterogene DRAM/NVM-geheugensystemen gericht op het optimaliseren en afleiden van trade-offs tussen prestaties, energieverbruik en hardware-slijtage, en (iv) stellen we een oplossing dicht bij de hardware oplossing voor paginaplaatsing voor die vertrouwt op LSTMs en dynamisch geheugenbeheer stimuleert door effectieve cache-configuratie.

Trefwoorden: dynamisch geheugenbeheer, heterogene geheugensystemen, DRAM, NVM, High Performance Computing

Acknowledgements

To be filled in the final version of the manuscript

Contents

Extended Abstract vii

Extended Abstract in Greek ix

Extended Abstract in Dutch xi

Acknowledgements xiii

1 Introduction 1
- 1.1 The Growing Role of Memory Management in the Digital Era 1
- 1.2 Summary of the State-of-the-Art and Challenges over Dynamic Memory Management 5
- 1.3 Ph.D. Thesis Scope & Contributions 9
  - 1.3.1 Extensive Analysis, Classification and Guidelines of Data Placement over Heterogeneous Memory Systems 11
  - 1.3.2 Application-Level Memory Management and Source-to-Source Transformations for Python Workloads 11
  - 1.3.3 Data Structure Exploration & Placement over Heterogeneous DRAM/NVM Memory Systems 12
  - 1.3.4 Hardware-Level Page Placement and Configuration 13
- 1.4 PhD Thesis Structure and Organization 14

2 Background and Fundamentals of Dynamic Memory Management 17

2.1 Background on Dynamic Memory Management 17
2.1.1 Importance & Mechanisms of Static and Dynamic Memory Management 17
2.1.2 Rise of Emerging Memory Technologies &Heterogeneous Memory Architectures 19
2.1.3 Industrial Memory Technologies 21

3 Review of State-of-the-Art Challenges and Limitations 25
3.1 Application-Level Dynamic Memory Management 25
3.1.1 Related Work 25
3.1.2 Key differences and extensions from the State of the Art 27
3.2 Data Structure, Indexes Design and Placement for Heterogeneous Memory Systems 27
3.2.1 Persistent Memory Data Structure/Indexes Optimization and Design 27
3.2.2 Evaluation and exploration of persistent memory indexes 28
3.2.3 Data Placement over Heterogeneous Memory Systems 28
3.2.4 Key differences and extensions from the State of the Art 28
3.3 Near-Hardware Solutions 29
3.3.1 Related Work 29
3.3.2 Key differences and extensions from the State of the Art 31

4 Classification and Analysis of Data Placement over Heterogeneous Memory Systems 33
4.1 Introduction 34
4.2 General Purpose and Application-specific Placement 36
4.2.1 General purpose placement 36
4.2.2 Databases and Big Data 37
4.2.3 Deep Learning 38
4.2.4 Graph-based Applications 39
4.2.5 HPC Applications 40
4.3 Analysis, Discussion and Qualitative Comparison 41
4.4 Design Space of Data Placement Approaches 45
4.5 Guidelines and Future Directions 47
4.5.1 Bridging gaps and future research 48
4.5.2 How can placement policies adapt to upcoming memory technologies? 49

5 Application-Level Memory Management and Source-to-Source Transformations for Python Workloads 51
5.1 Introduction 52
5.2 Background and motivation 54
5.3 Memory Footprint Optimization Methodology 57

5.3.1 Overview . . . 57
5.3.2 Memory Footprint Optimizations based on Static Analysis . . . 61
5.3.3 Memory Footprint Optimizations based on Dynamic Analysis . . . 62
5.4 Experimental Results and Evaluation . . . 66
5.4.1 Experimental Setup . . . 66
5.4.2 Pyperformance suite & CNN Application . . . 68
5.4.3 IoT biomedical application and use-case demonstration . . . 70
5.4.4 Discussion . . . 74
5.5 Conclusion . . . 77

6 Data Structure Exploration & Placement over Heterogeneous DRAM/NVM Memory Systems 81
6.1 Evaluation of Optane DC Persistent Memory for Indexing Data Structures 82
6.1.1 Introduction . . . 82
6.1.2 Evaluation Methodology . . . 84
6.1.3 Evaluation . . . 87
6.1.4 Conclusion . . . 98
6.2 Persistent B+-Tree Indexes over Heterogeneous DRAM/NVM Systems . . 99
6.2.1 Introduction . . . 99
6.2.2 Proposed Methodology . . . 101
6.2.3 B+-tree Indexes Library . . . 101
6.2.4 Proposed Methodology . . . 103
6.2.5 Experimental Setup and Evaluation . . . 107
6.2.6 Experimental Evaluation . . . 109
6.2.7 Use-Case Integration of YCSB to LevelDB Database . . . 122
6.2.8 Conclusion . . . 122
6.3 Dynamic Data Type Refinement and Placement over DRAM/NVM . . . 123
6.3.1 Introduction . . . 123
6.3.2 Background &Motivation . . . 125
6.3.3 Proposed Methodology . . . 129
6.3.4 Optimization of Data Organization . . . 131
6.3.5 Memory Object Analysis &Profiling . . . 133
6.3.6 Placement on Heterogeneous Memory Systems &Evaluation . . . 134
6.3.7 Experimental Evaluation . . . 135
6.3.8 Conclusion . . . 143
6.4 On the Implications of Heterogeneous Memory Tiering on Spark In-Memory Analytics . . . 144
6.4.1 Introduction . . . 144
6.4.2 Hardware and Software Testbed . . . 145
6.4.3 Characterization &Analysis . . . 148
6.4.4 Conclusion . . . 156

7 Hardware-Level Memory Management and Configuration 157
7.1 Adjacent LSTM-based Page Scheduling for Hybrid DRAM/NVM Memory Systems . . . 158
7.1.1 Introduction . . . 158
7.1.2 Proposed Methodology . . . 159
7.1.3 Critical Page Selector . . . 160
7.1.4 Experimental Setup . . . 164
7.1.5 Discussion . . . 168
7.1.6 Conclusion . . . 169
7.2 Optimizing Cache Configurations via Dynamic Memory Management in Edge Computing Systems . . . 169
7.2.1 Introduction . . . 169
7.2.2 Exploration Methodology &Decision Support . . . 171
7.2.3 Experimental Evaluation . . . 173
7.2.4 Conclusion . . . 177

8 Conclusions 179
8.1 Thesis Summary . . . 179
8.2 Memory Sub-systems of the Future . . . 181
8.3 Future Extensions . . . 183
8.3.1 Sustainable and Green Computing . . . 183
8.3.2 Novel Hybrid Memory Architectures and Technologies . . . 184
8.3.3 Memory Disaggregation and in-memory Computing . . . 185

Appendices 187

A DMRM: Distributed Market-Based Resource Management on Edge Computing Systems 189
A.1 Introduction . . . 190
A.2 Related Work . . . 191
A.3 System and Problem Formulation . . . 192
A.4 Distributed Market-Based IoT Management . . . 195
A.4.1 Background and Main Proposed concepts . . . 195
A.4.2 Decision making functions of DMRM on the IoT node . . . 196
A.4.3 Decision making functions of DMRM on the Gateway . . . 198
A.5 Experimental Evaluation . . . 199
A.5.1 Experimental Setup . . . 199
A.5.2 Comparative Study . . . 202

B SGRM: Stackelberg Game-Based Resource Management on Edge Computing Systems 205

B.1 Introduction . . . 206
B.2 Related Work . . . 207
B.3 System and Problem Formulation . . . 208
B.4 SGRM: A Stackelberg Game-Based Resource Management Framework . . 210
B.4.1 Background and Main Proposed Concepts . . . 210
B.4.2 Task Evaluation Module . . . 212
B.4.3 IoT Node Module . . . 213
B.4.4 Gateway Module . . . 214
B.4.5 Implementation . . . 215
B.5 Experimental Evaluation . . . 215
B.5.1 Experimental Setup . . . 215
B.5.2 Experimental Results . . . 216
B.6 Conclusion . . . 219

C RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems 221
C.1 Introduction . . . 222
C.2 Related Work . . . 224
C.3 `RoaD-RuNNer`: A collaborative DNN partitioning &offloading framework . 225
C.3.1 Offline Phase . . . 226
C.3.2 Online Phase . . . 227
C.4 Experimental Evaluation . . . 229
C.4.1 Experimental Setup . . . 229
C.4.2 Evaluation . . . 231
C.5 Conclusion . . . 234

D Decoupled Access-Execute enabled DVFS for tinyML deployments on STM32 microcontrollers 237
D.1 Introduction . . . 238
D.2 Clocking Scheme of STM32 Microcontrollers . . . 240
D.2.1 Considerations of `SYSCLK` Frequency Scaling . . . 241
D.3 Decoupled Access Execute enabled DVFS on STM32 MCUs . . . 242
D.3.1 Step 1: Memory Access &CPU Execution Decoupling . . . 243
D.3.2 Step 2: DAE and Clocking Co-exploration . . . 245
D.3.3 Step 3: QoS-aware Energy Optimization . . . 246
D.4 Experimental Setup and Evaluation . . . 247
D.5 Conclusion . . . 249

Abbreviations &Acronyms 251

Publications 255

List of Figures

1.1 Scaling of the bandwidth of different interconnections, memory, and Peak FLOPS, showing the *Memory Wall* and the *Interconnect Wall* 3
1.2 Zeta-bytes of data produced from 2010 to 2024, projected data production for 2025 and annual percentage difference. 4
1.3 Amount of data produced and categorization from different state-of-the-art applications 4
1.4 History and Timeline of memory technologies 5
1.5 Schematic overview of the abstraction layers of applying dynamic memory management on the hardware-software continuum. 6
1.6 Overview of the dissertation contributions across the HW/SW continuum discrete components 8
1.7 Schematic overview of the contributions of this thesis across the levels of applying memory management on the hardware-software continuum. . . . 14

2.1 Hierarchy Pyramid of Memory Technologies 20
2.2 Operating modes of Intel Optane DCPM 22
2.3 Programming model of Intel Optane DC Persistent Memory 23

4.1 Overview of the application-specific data placement approaches for (a) Data bases &Big Data, (b) Deep Learning, (c) Graph-based and (d) HPC workloads. 36
4.2 Overview of the data placement design choices and guidelines as derived throughout the literature 46

5.1 CPython's memory management organization 55
5.2 Motivational example showing CPython's static memory management approach 56
5.3 Overview of the memory footprint optimization methodology 58
5.4 Memory footprint optimization methodology 58
5.5 Visual representation of the code hierarchy, inter and intra-dependencies and example of dependency resolution. 64
5.6 Memory footprint optimization for Pyperformance Benchmark Suite . . . 68

5.7 Execution time optimization for Pyperformance Benchmark Suite 69
5.8 Experimental evaluation of memory footprint and execution latency optimizations on Convolutional Neural Network(NN) application over MNIST dataset. 71
5.9 Experimental evaluation of memory footprint optimizations on IoT biomedical application 71
5.10 Static (left) vs dynamic (right) memory management of the IoT biomedical application, before and after applying the optimization methodology. . . . 73
5.11 Comparison of initial and optimized memory footprint utilization 74
5.12 Energy consumption reduction percentage(%) for Pyperformance Benchmark Suite. 75

6.1 Overview of the evaluation methodology 85
6.2 Scalability evaluation on Intel Optane DCPM for different types of workloads. 88
6.3 Scalability evaluation on DRAM-only for balanced workload. 89
6.4 Throughput for balanced workload using a single Optane DCPM DIMM. 90
6.5 Energy consumption evaluation on Intel Optane DCPM for different types of workloads. 90
6.6 Energy consumption on DRAM-only for balanced workload. 91
6.7 Energy consumption of CPUs and memory system. 92
6.8 Number of Intel Optane DCPM read and write memory accesses. Workload a: balanced, b: write-heavy, c: read-heavy, d: scan-heavy. 93
6.9 Energy consumption on a single Optane DCPM DIMM for balanced workload. 94
6.10 System configurations for the LevelDB experiments: DRAM/SSD and DRAM/Optane DCPM. 95
6.11 LevelDB throughput for different memory configurations under balanced and read-heavy workload. 96
6.12 LevelDB energy consumption for different memory configurations under balanced and read-heavy workload. 98
6.13 Overview of the proposed exploration methodology and decision support . 105
6.14 Load time (left) and throughput (right) over TPC-C workload for the B+-Tree indexes deployed on Optane DCPM. 108
6.15 Execution throughput for the TPC-C workload for the examined B+-tree indexes deployed on Optane DCPM. 109
6.16 TPC-C average query latency 110
6.17 Load (top) and run (bottom) for the B+-tree indexes over the YCSB workloads deployed on Intel Optane DCPM. 111
6.18 Normalized NVM energy for the TPC-C workload for the examined B+-tree indexes deployed on Intel Optane DCPM. 112

6.19 Normalized NVM energy consumption for the TPC-C workload for the examined B+-tree indexes deployed on Intel Optane DCPM. 113
6.20 Normalized NVM energy consumption for the YCSB workload for the examined B+-tree indexes deployed on Intel Optane DCPM under different workloads. 114
6.21 Number of read (top) and write (bottom) access of Intel Optane DCPM DIMMs for the TPC-C workload for the examined B+-tree indexes. . . . 115
6.22 Number of read (top) and write (bottom) access of Intel Optane DCPM DIMMs for the YCSB workload for the examined B+-tree indexes. 116
6.23 Projected normalized NVM lifetime for the examined B+-tree indexes deployed over the different YCSB workload. 117
6.24 Normalized run throughput and normalized NVM energy comparison of string vs integer keys for the examined B+-tree indexes over the YCSB benchmark workload. 118
6.25 Intel Optane DCPM system's throughput evaluation for the examined B+-tree index implementations over YCSB balanced workload (A) for 64 threads and integer keys. 119
6.26 Pearson correlation of high-level metrics for all the examined B+-tree indexes and workloads (-1: perfect negative correlation, 0: No correlation, 1: Perfect positive correlation) 120
6.27 LevelDB throughput for different YCSB workloads 122
6.28 Investigating the impact of an application-level optimization for improved data organization (DDTR) on the results of data placement, in terms of execution time, energy consumption and impact on NVM lifetime. (a) The output of DDTR: 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 (c) Number of physical read/write accesses for each application version. 127
6.29 Overview of the proposed methodology 128
6.30 Average distribution of the data structures of the Pareto optimal solutions over heterogeneous memory systems for each application. 133
6.31 Design space and performance/memory size/number of NVM writes Pareto optimal solutions for each application deployed on a DRAM/Optane DCPM system 135
6.32 Design space and performance/memory size/number of NVM writes Pareto optimal solutions for each application deployed on a DRAM/PCM, DRAM/R-RAM and DRAM/STT-MRAM system. 138
6.33 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. 140

6.34 Impact of data organization optimization and placement on the lifetime of the Optane DCPM DIMMs . . . 141
6.35 Impact of the methodology on the application memory footprint: The Pareto optimal application versions have lower memory requirements compared to the initial versions. . . . 142
6.36 Total exploration time (hours) of each step of the methodology for each application. . . . 143
6.37 Exploration time over increasing number of target data structures. . . . 143
6.38 Overview of the target multi-tier system architecture and testbed . . . 146
6.39 Execution time for all the different memory tiers (top), number of accesses (reads and writes) to memory (middle) and DRAM/NVM energy comparison (bottom) for all the different dataset sizes and benchmarks examined. 148
6.40 Execution time over various memory bandwidth levels for all benchmarks and tiny,small and large input workloads . . . 151
6.41 Speedup/Slowdown of `sort`, `rf`, `lda` and `pagerank` applications for all workloads for varying number of cores and executors. The default configuration is 1 executor with 40 cores. . . . 152
6.42 Correlation of system-level metrics with execution time . . . 153
6.43 Correlation of hardware specs (latency and bandwidth) with execution time for all applications and workloads . . . 155

7.1 Overview of the proposed page scheduler over heterogeneous DRAM/NVM systems . . . 160
7.2 Normalized performance comparison of alternative placement scheduling policies. Results are normalized based on Oracle prediction. . . . 165
7.3 Root Mean Square Error(RMSE) comparison of alternative placement policies for all benchmarks.Y-axis is in log scale. . . . 165
7.4 Normalized energy consumption of alternative placement scheduling policies. Results are normalized based on History-based scheduler. . . . 166
7.5 Comparison of LRU vs Clustered-LRU replacement policies over alternative benchmarks. . . . 167
7.6 Overview of the Proposed Methodology . . . 172
7.7 Target RISC-V Architecture . . . 173
7.8 Impact of static vs dynamic memory management on LLC cache configuration for the IoT biomedical application. . . . 174
7.9 Impact of the L1,L2 and LLC size and associativity configurations on the execution time for the dynamic version for the IoT biomedical application. 175
7.10 Impact of Application-level static vs dynamic memory management and cache configurations to the L1 cache misses. . . . 175
7.11 Evaluation of cache replacement policies for static and dynamic memory management application versions. . . . 176

8.1 CXL Memory Module Architecture 182

A.1 Target system architecture 193
A.2 #Tasks with missed deadlines (Small number of tasks) 201
A.3 #Tasks with missed deadlines (High number of tasks) 201
A.4 Total rounds of exceeded deadlines (Demanding input) 202
A.5 Comparative evaluation of DMRM and SA based resource management on Gateway alternatives (log. scale) 203

B.1 Overview of SGRM Operation Flow among IoT Nodes and Edge Gateways 211
B.2 #Tasks with missed deadlines (small-scale workload) 217
B.3 #Tasks with missed deadlines (larger-scale workload) 217
B.4 Execution time prediction mechanism analysis 218
B.5 SGRM's tasks and Gateways scalability 218

C.1 DNN optimal partitioning schemes w.r.t. *a)* different optimization goals and *b)* device heterogeneity. 223
C.2 Overview of Online and Offline RoaD RuNNer Architecture. 226
C.3 Performance and Energy Comparison of RoaD-RuNNer framework against other approaches for CPU and GPU nodes for alternative DNN workloads. 231
C.4 Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages. 232
C.5 Execution Latency and Energy Consumption Prediction Accuracy for alternative DNN workloads and Edge nodes. 233
C.6 DNN percentage offloading over heterogeneous Edge nodes for latency and energy optimization objectives. 234

D.1 Simplified circuit diagram for clock configuration through `HSE` and `PLL` parameters. 241
D.2 Clock Frequency and Power for different `HSE`, `PLLM` and `PLLN` configurations.242
D.3 Overview of the proposed methodology. 243
D.4 Impact of different DAE and clocking configurations on latency and power of depthwise and pointwise layers. 245
D.5 Energy consumption gains of our approach over the TinyEngine [1] baseline. We compare against TinyEngine with Clock Gating over the examined CNN models. 248
D.6 Frequency distribution throughout layer progression over the examined CNN models for 10% and 50% QoS constraints. 248

List of Tables

2.1 Overview of the specs for DRAM technologies, NVM and storage systems. 21

3.1 Summary of related research for Python memory footprint optimizations . 26

4.1 Classification of existing NVM-related and data placement related surveys. 35
4.2 Classification of existing literature per application domain based on profiling and placement approach. 40
4.3 Classification of application specific data placement literature based on the integrated placement method. 41
4.4 Classification of existing profiling techniques over the state-of-the-art literature. 43
4.5 Classification of the existing literature based on placement granularity (adaptive/pre-defined). 44
4.6 Classification of the literature based on target evaluation systems(Hardware, Simulation and Emulation). 44
4.7 Classification of existing profiling techniques. 45

5.1 Memory specifications (RAM) of indicative state-of-the-art embedded boards 53
5.2 Python arenas and pools optimization for Pyperformance Benchmark Suite 70
5.3 Python and C code comparison 76

6.1 Overview of experimental setup 88
6.2 Energy consumption (KJ) of Optane DCPM DIMMs for each workload type for 16 threads. 91
6.3 Key design characteristics of the integrated persistent B+-tree indexes . . 101
6.4 Key system parameters and variables for multi-objective problem optimization 106
6.5 YCSB default workload specifications [2] 107
6.6 The design space of data structure implementations in the DDTR library used for the data organization optimization. 130
6.7 Overview of the applications used for the evaluation of the methodology . 131
6.8 Pareto optimal solutions for each application deployed on a DRAM/Optane DCPM system. 136

6.9 Subset of Pareto optimal solutions for each application deployed on a DRAM/PCM, DRAM/RRAM and DRAM/STT-MRAM system. 139
6.11 Examined Spark applications and dataset sizes 147
6.10 Idle access latency and memory bandwidth per tier 147

7.1 Overview of Benchmarks 164
7.2 Overview of the design space specifications 173
7.3 Optimal Configurations 176

A.1 Key system model parameters 194

B.1 Key System Model Parameters 209

C.1 Technical characteristics of heterogeneous Edge nodes and Cloud Server . 229

Chapter 1.

Introduction

1.1. The Growing Role of Memory Management in the Digital Era

Memory management in recent systems refers to the process of controlling, coordinating, and organizing computing platforms main memory resources. The primary and major goal of memory management is to provide efficient and effective utilization of the computing platforms memory, enabling programs and data to be stored, accessed, and manipulated through a systematic process. However, the advances in processing, memory and interconnection technologies over the last decade keep increasing the disparity of performance among the diverse processing cores, e.g. Compute Processing Units (CPUs), Graphics Processing Units (GPUs), RISC-V and ARM cores and Tensor Processing Units (TPUs) and the memory technologies and interconnection protocols, known as the "*Memory Wall*", as illustrated in Figure 1.1 [1]. Dealing with the memory wall requires an end-to-end approach that considers both software and hardware optimizations. As the demand for faster and more efficient computing continues to grow, efforts to overcome the memory wall remain a major topic of research and development in computer architecture and system design.

Furthermore, one of the significant challenges encountered in advanced technology nodes is commonly referred to as the "*Interconnect Wall*". This barrier arises due to the dominance of lengthy wires within the system, which significantly impact both delay and energy consumption at a system-wide level. In this context, the decrease of devices, encompassing both logic and storage components, has reached a stage where their direct influence becomes gradually negligible compared to the considerable loading posed by these quite active intermediate-length wires, especially when operating at high speeds [3–5]. It

[1] The memory wall

is worth noting that while devices are indispensable for driving these loads, they are primarily utilized for functions other than arithmetic and logic processing. Therefore, they should not be categorized as logic devices when considering the overall system dynamics. Hence, special focus should be given towards internal memory access and data transfer operations within the system's architecture, particularly focusing on interactions with main memory components [6–8]. The interconnect wall varies from micro-level interconnections to mm-level interconnections. This barrier, encountered in advanced technology nodes, is due to the dominance of lengthy wires within the system, significantly impacting delay and energy consumption at a system-wide level. Additionally, in on-chip configurations, the challenges posed by the interconnect wall are exacerbated, as optical alternatives, such as those employed in NVLink [9], PCIe [10], and CXL [11] protocols, are not readily available, further amplifying the impact of lengthy interconnects on system performance.

While the amount of applications and data produced has rapidly increased over the last years, the existence of effective memory management mechanisms is becoming more and more important. Figure 1.2 illustrates the amount of annual data produced in Zettabytes (ZB), showing that since 2010 the amount ZB produced has increased by 73.5× since 2024 and is expected to keep rising in the next years [2]. On top of that, unexpected social phenomena such as the COVID-19 pandemic, which exploded in 2020, only expedited this transition and has forced a rise of +56.59% in 2020 compared to 2019, therefore forcing businesses and companies of all kinds to employ innovative digital solutions to survive and adjust to unpredictable events.

Additionally, over the last years, we have observed a cataclysm of data produced in terms of their heterogeneous nature. Video streaming, social media, gaming, and messaging data represent a small subset of types of data produced by different applications (Figure 1.3). Figure 1.3a shows the variation of data produced [3], while Figure 1.3b indicates the most dominant events that occurred in some of the most widespread applications per minute [4]. This co-explosion of the heterogeneity and the amount of data generated leads to a complex paradigm of data that needs to be transferred, stored, processed, and analyzed. Moreover, recent advances in emerging applications, such as Large Language Models (LLMs) [12] and interactive AI application [13] are going to further explode the amount of weight and data that needs to be handled in these huge transformers.

Static memory management solutions have been extensively researched and developed

[2] Worldwide data growth
[3] Data generation per day
[4] Event generation for famous applications

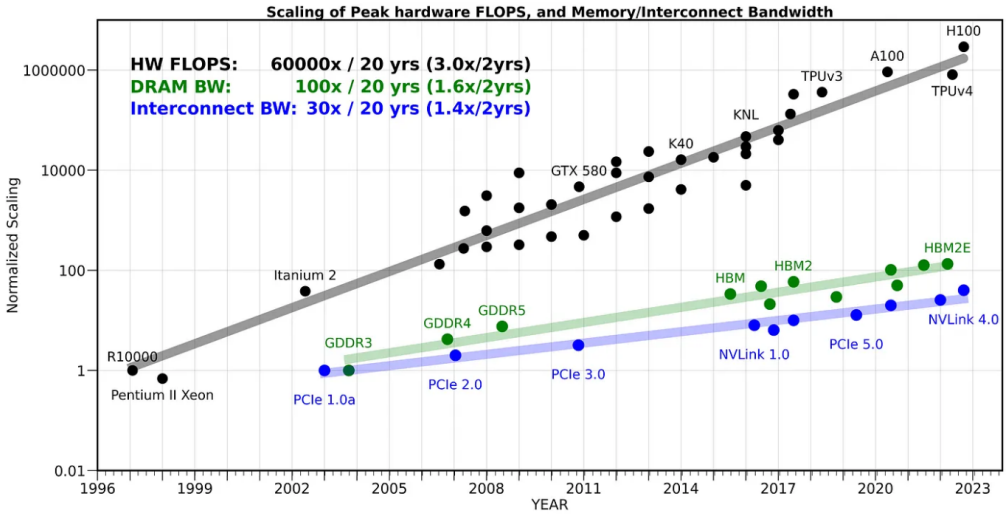


Figure 1.1.: Scaling of the bandwidth of different interconnections, memory, and Peak FLOPS, showing the *Memory Wall* and the *Interconnect Wall*

by the academic and industry communities [14–18]. Such kind of solutions offer advantages in terms of simplicity and predictability, making them suitable for scenarios where memory requirements are known and constant a priori. However, these solutions are often limited by their special characteristic to focus on worst-case data types like statically allocated arrays, as memory allocations are predetermined during compile time, leading to fixed memory layouts and sizes. Therefore,when application workloads become very dynamic, that static allocation option becomes too inefficient. This lack of adaptability can result in inefficient memory usage and potential wastage of resources, especially in scenarios where memory requirements vary dynamically. Furthermore, static memory management solutions may struggle to accommodate the evolving needs of modern software applications, which demand more flexible and dynamic memory allocation strategies.

Aiming to surpass existing limitations imposed by static memory management solutions developed the past decades [14–21], dynamic memory management solutions are increasingly recognized as essential for designing effective and adaptable software systems. Moreover, they enable efficient memory allocation and deallocation during runtime, accommodating varying memory requirements and optimizing resource utilization. With the complexity and dynamism of modern applications, dynamic memory management solutions offer the flexibility needed to address evolving demands and maximize system performance. Furthermore, with the rapid advancement of virtualization technologies such as hypervisors and containerization, the need for dynamic memory management has

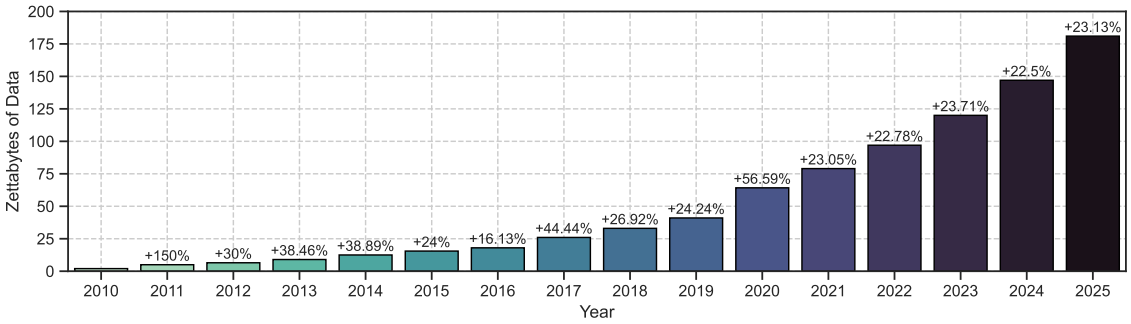
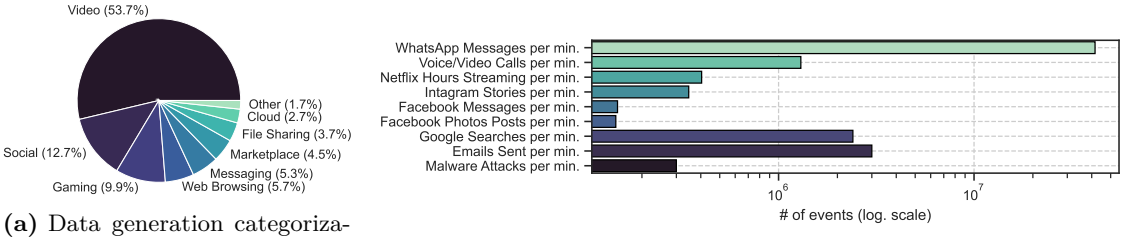


Figure 1.2.: Zeta-bytes of data produced from 2010 to 2024, projected data production for 2025 and annual percentage difference.



(a) Data generation categorization for 2023

(b) Events per minute of different applications

Figure 1.3.: Amount of data produced and categorization from different state-of-the-art applications

become increasingly critical. Hypervisors like Microsoft Hyper-V [22], and KVM [23] (Kernel-based Virtual Machine) enable the creation of multiple virtual machines (VMs) on a single physical server at run-time. Thus, as software continues to evolve, dynamic memory management solutions remain indispensable for achieving scalability, responsiveness, and robustness in memory management.

Moreover, in addition to the ever-increasing heterogeneity and dynamicity in the amount of data produced, recent dynamic memory management solutions need to adapt to the memory hardware evolution and the transition from the *Single-Technology Memory Era* to the *Heterogeneous Memory Era*, as depicted in Figure 1.4. Since the early 2010s Non-Volatile Memory (NVM) technologies have been introduced, such as 3D-XPoint [24], Resistive RAM (ReRAM) [25], Phase-Change Memory (PCM) [26], Spin-Transfer Torque Magnetic RAM (STT-MRAM) [27] and Ferroelectric RAM (FeRAM) [28]. Moreover, the adaptation of Intel Optane Data Center Persistent Memory (DCPM) [29] to recent HPC systems, which is the only commercial NVM technology up to now, has led to the first real-hardware heterogeneous memory platforms [30, 31].

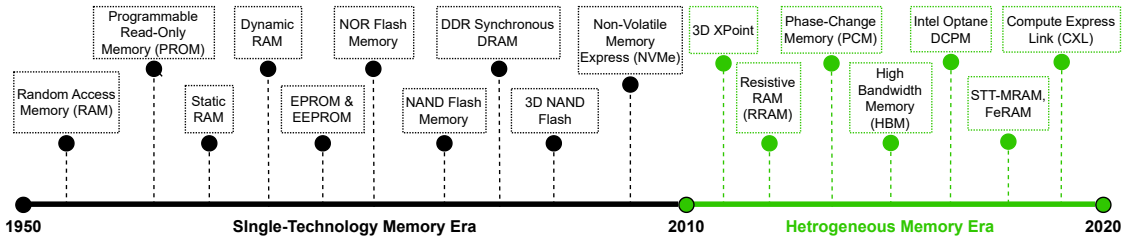


Figure 1.4.: History and Timeline of memory technologies

Such kind of technologies are utilized alongside the traditional DRAM memories, composing complex heterogeneous memory systems, where each memory technology imposes different characteristics in terms of performance and energy specifications. Furthermore, the advances in recent NVM technologies, as well as the integration of emerging technologies such as the Compute Express Link (CXL) [32] protocol and other memory technologies, e.g. Samsung Memory Expander [33], are expected to further extend the heterogeneity of state-of-the-art heterogeneous memory systems and impose diverse characteristics in the overall system's behavior.

The increasing demand of memory technologies and devices has led to the next-generation memory market. The Memory Market size is estimated at $148.95 billion in 2024 and is expected to reach $215.13 billion by 2029, growing at a CAGR of 7.63% during the forecast period (2024-2029) [5]. Major vendors in the next-generation memory companies include SAMSUNG, Micron, Everspin Technologies and Fujitsu [6], highlighting the strong interest of the industrial community.

1.2. Summary of the State-of-the-Art and Challenges over Dynamic Memory Management

Dynamic memory management plays a crucial role in modern computing systems, allowing efficient allocation and deallocation of memory resources at runtime. In the state-of-the-art, dynamic memory management encompasses a variety of techniques and algorithms aimed at optimizing memory usage and improving overall system performance, sustainability and hardware wear. Aiming to satisfy the aforementioned criteria, dynamic memory management solutions require an effective co-design of multiple components of the hierarchy of the computing continuum, i.e. varying from application level to

[5] Memory Market Growth
[6] Memory Market Vendors

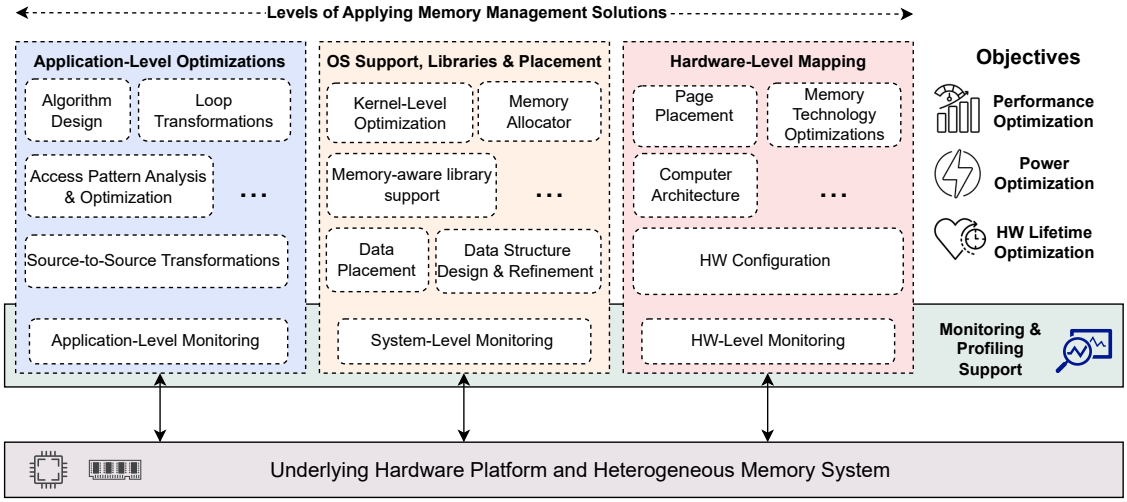


Figure 1.5.: Schematic overview of the abstraction layers of applying dynamic memory management on the hardware-software continuum.

hardware-based solutions, as illustrated in the Figure 1.5.

At the application level, dynamic memory management involves techniques such as code transformations [34–36], garbage collection [37], [38], [39], virtualization [40, 41] and memory-aware algorithms to optimize memory usage and minimize overhead. Applications often utilize dynamic memory allocation to adapt to varying workload demands and efficiently manage memory resources. In the OS-middleware layer, dynamic memory management includes indexes and library design [42–52], memory allocation policies and placement [53–71], virtual memory management [72,73] and inter-process communication mechanisms to provide efficient memory access and resource sharing among different processes and threads. Operating systems employ sophisticated memory management techniques such as data placement, paging, segmentation, and memory mapping to abstract physical memory and provide a virtual memory space for applications. Additionally, middleware components such as memory allocators and memory managers play a crucial role in managing memory resources efficiently and ensuring fair allocation among competing processes. Near-hardware dynamic memory management involves techniques implemented in hardware or firmware layers to optimize memory access and utilization. This includes memory controllers, cache management algorithms, and memory-interconnect technologies that aim to minimize memory latency, reduce energy consumption, and improve overall system performance [74–81]. Dynamic memory management at this level often involves fine-grained optimizations tailored to specific hardware architectures and memory technologies.

Moreover, over the last few years, recent advancements in memory technologies have drastically moved the way that memory is conceptualized and managed over state-of-the-art systems, ranging from Edge to Cloud and HPC computing [82–85]. More specifically, recent systems have witnessed a paradigm shift regarding memory concepts, driven by the ever-increasing diversity in terms of memory technologies, type of data produced and application workloads [86, 87]. This kind of diversity has opened the landscape of applying more dynamic and adaptive memory management solutions to each level in the hardware-software continuum [34–39, 42–71].

Application-Level Optimizations : The highest level of applying memory management optimization involves tailoring and fine-tuning application data organizations and algorithms to the specific software requirements [34–39]. Considering the memory implications and the potential impact of application-level decision-making on the underlying hardware infrastructure during the design phase of the applications is crucial for achieving high performance and boosting sustainability. Fully effective application-level optimizations usually require comprehension of the examined workload and its inherent characteristics, thus making the integration of optimizations even more challenging

OS, Libraries and Placement Optimizations : The intermediate-level optimizations, usually referred to as the *middleware*, play a dominant role in efficiently coordinating and managing memory resources [42–71]. The most dominant components that support software memory management are integrated in this operating level. Intelligent memory allocation policies, garbage collectors, compilers, memory-specific library support, and memory-sharing mechanisms are just some examples of the landscape of memory-related components. Libraries can be integrated to facilitate efficient memory usage by providing well-optimized functions, interfaces, and data structures. Furthermore, strategic memory placement, considering factors like locality and access patterns and taking advantage of state-of-the-art mechanisms such Machine Learning (ML)-based solutions, can significantly enhance the overall system performance, reduce the energy consumed, and boost the hardware's lifetime.

Hardware-Level Optimizations : The lowest level of optimizations aims to apply memory management solutions on the hardware level [74–81]. This level involves leveraging advancements over memory technologies and architectures. Computer architecture, memory hierarchies, page placement, cache memory configurations, and memory access patterns are critical considerations at the hardware level. Memory management at the

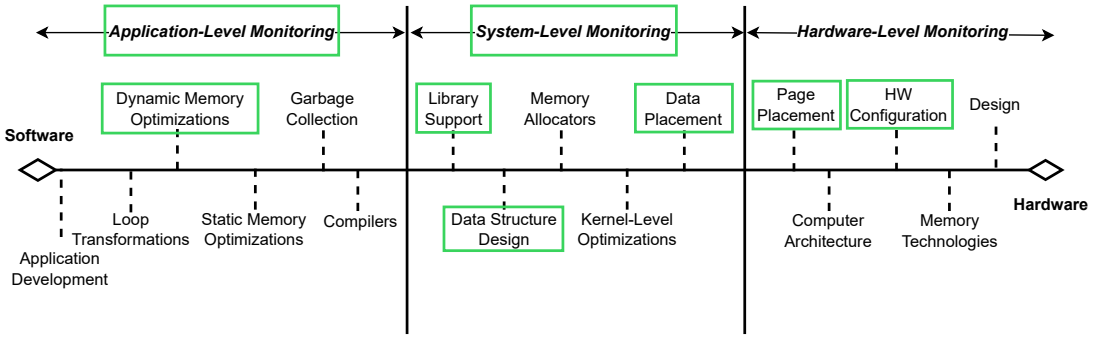


Figure 1.6.: Overview of the dissertation contributions across the HW/SW continuum discrete components

hardware level involves a sophisticated interplay of architectural decisions, interconnect optimizations, and data transfer strategies. The ongoing exploration of innovative solutions at this level is crucial for unlocking the full potential of memory systems in modern computing systems.

Summarizing the aforementioned, this aims to investigate dynamic memory management solutions that span various levels within computing systems. However, across the computing continuum still exist significant gaps that need to be addressed and tackled, in order to break the "Memory Wall" and create more efficient and sustainable systems. Focusing on the application-level, recent solutions do not focus on the impact of the ineffective coding in terms of memory management, to the overall system's performance and energy consumption. Moreover, traditionally, application developers tend to program without considering the special characteristics of the underlying architecture and memory sub-systems. Furthermore, recent operating systems need to provide effective support towards the application developers, in order to tackle the challenges imposed by the novel heterogeneous memory hardware integrated in recent platforms. Thus, operating systems require extensive solutions towards providing effective libraries for application development and placement policies for heterogeneous platforms, while providing solutions for the effective hardware configuration, aiming to boost performance, energy consumption and hardware's lifetime. Last, near-hardware solutions, such as page placement and cache sub-system configuration, in addition to the aforementioned gaps need to be addressed. In Section 1.3 we analyze the major contributions of this thesis, that aim to fill the existing research gaps.

1.3. Ph.D. Thesis Scope & Contributions

The scope of this thesis is to examine the applicability of dynamic memory management techniques over distinct levels across the hardware-software continuum, aiming to provide performance, energy, and memory resources optimization. Aiming to follow recent advances in heterogeneous memory technologies and state-of-the-art computing platforms on the Edge-HPC continuum and in order to fill the existing research gaps presented in Sec 1.2, in this dissertation we focus on optimizing memory management over different optimization abstraction layers (Fig. 1.5). More specifically, the novel contributions of this thesis focus on the following optimization levels: i) application-level optimizations, ii) OS support, libraries and placement optimizations, and iii) hardware-level mapping optimizations. Given the inseparable and strong relationship between hardware and software memory components, we aim to enhance the interoperability of different optimization layers and provide an end-to-end workflow and methodology. Figure 1.6 illustrates the components of the HW/SW continuum that this dissertation aims to contribute to. Furthermore, in addition to the aforementioned technical contributions, we provide an extensive analysis, classification of data placement solutions over heterogeneous memory systems and we derive guidelines for the design of data placement solutions.

In summary, the major contributions of this PhD thesis are the following:

- **Extensive Analysis, Classification and Guidelines of State-of-the-Art Data Placement over Heterogeneous Memory Systems**: We present an extensive survey for application-specific software data placement approaches that target heterogeneous main memory systems. Throughout the existing literature, we present the current state-of-the-art trends in software data placement, we highlight key design choices and features that affect decision making over various application domains. Through this extensive literature review, we indicate existing gaps in the literature, we present a qualitative comparison of different approaches and provide future guidelines and directions for extension of recent research and adaptation of the existing works in the upcoming main memory technologies.

- **Application-Level Memory Management and Source-to-Source Transformations for Python Workloads**: A novel solution for applying source-to-source transformations for Python applications aiming to support memory footprint optimization. This approach leverages a set of static (offline) and dynamic (online) analysis optimizations. Through this research, our ambition is to support developers and establish memory footprint-friendly solutions as the default in application development design and extend the range of target Edge computing devices that

Python applications can be deployed on. Python serves as a prime example of higher-level languages targeted by this more generic approach, emphasizing the broader scope and applicability of the proposed memory management strategies.

- **Data Structure Refinement, Exploration and Placement over Heterogeneous DRAM/NVM Memory Systems**: In this level, we present a set of solutions for dynamic data type and indexes refinement, design place exploration and placement over heterogeneous DRAM/NVM memory system. We target multi-objective optimization, aiming to optimize and derive trade-offs among performance, energy consumption, and hardware wear. In this thesis a set of different application domains is targeted, varying from database workloads to ML-driven applications and in-memory Spark analytics. The scientific burden of this research aims to establish an overall rethinking of the data structure organization and placement over DRAM/NVM memory systems, derive key outcomes for different application domains, and boost fine-grained and coarse-grained data placement policies.

- **Hardware-Level Page Placement and Configuration**: An LSTM-based framework for page placement over heterogeneous DRAM/NVM memory systems and memory management boost through effective cache configuration. In the scope of this thesis, the LSTMs are utilized for memory accesses prediction and hot/cold pages classification, while taking advantage of historical behavior of neighboring pages in the memory. Through this research, the objective is to re-design existing page placement policies of recent systems, aiming to provide performance improvements and energy savings. Last, the impact of cache configurations is exploited over dynamic memory management, aiming to open the research path for full end-to-end memory-cache management co-design.

Besides its major contributions, this thesis also examines four additional problems, i.e., i) the integration of market-based resource management solutions for task offloading on Edge computing systems, ii) the integration of game theory- and auction-based resource management solutions for task offloading on Edge computing systems, iii) DNN partitioning and offloading in heterogeneous CPU/GPU Edge/Cloud computing environments and iv) decoupled access-executed enabled Dynamic Voltage Frequency Scaling (DVFS) for tinyML deployments on STM32 microcontrollers (MCUs).

In the following subsections, we illustrate the major contributions of this thesis. We explain the problems that these contributions tackle and how they extend prior SoA solutions, as described later in depth in Chapter 3.

1.3.1. Extensive Analysis, Classification and Guidelines of Data Placement over Heterogeneous Memory Systems

Addressed problem: *Lack of extensive analysis, classification and guidelines for data placement solutions over heterogeneous memory systems*

The rapid growth of applications deployed over heterogeneous memory systems is rapidly increasing, thus leading to the rise of data placement solutions. Such solutions can be either general purpose or application-specific, while several choices in the design of a data placement solution significantly affect its overall behavior, e.g. placement granularity, optimization objective, static and/or dynamic placement. While multiple works try to propose data placement solutions [53–59, 63–71, 88–101], there is no dedicated research to survey the existing software data placement approaches in heterogeneous memories.

The proposed work extends prior research by:

- Classifying the existing literature based on various design features and parameters and highlight research trends and approaches per application domain.
- Providing an extensive discussion and qualitative comparison of the existing methodologies, aiming to reach conclusions and extract recent guidelines on data placement. We also derive a trade-off design space, indicating the key design choices for application specific data placement.
- Highlighting existing gaps in the literature and providing future guidelines and directions based on how existing placement approaches can be extended and adjusted to upcoming memory technologies.

1.3.2. Application-Level Memory Management and Source-to-Source Transformations for Python Workloads

Addressed problem: *Lack of source code level memory footprint optimization frameworks for Python applications targeting edge devices.*

Python programming language provides the application developers with a wide variety of libraries and abstractions that encapsulate low-level optimizations and allow the expression of extended functionality with less amount of code [102–104]. Such kind of features

features enable rapid application development, due to the limited programming effort required. However, porting computationally demanding applications developed in high-level languages to resource-constrained edge devices becomes more and more challenging, due to limited memory resources. While prior research has proposed memory footprint optimizations for Python programming language, they do so either by i) integrating optimizations on the interpreter's and memory manager's functionality [105, 106], ii) proposing garbage collector-based optimizations [37–39] and/or iii) focusing on application-specific and library optimizations [34–36]. To tackle these limitations, we propose a systematic methodology for memory footprint optimizations over resource-constrained Edge devices, aiming to enhance the dynamic memory management.

The proposed solutions extend prior research by:

- Presenting an application-level memory footprint optimization systematic methodology for memory footprint optimization, without modifying either Python interpreter or memory manager;
- Delivering an automated framework, that provides static and dynamic analysis-based optimizations;
- Presenting in-depth insights on critical Python parameters that influence the memory consumption;
- Provide side-effect optimizations in terms of performance and energy consumption.

1.3.3. Data Structure Exploration & Placement over Heterogeneous DRAM/ NVM Memory Systems

Addressed problem: *Absence of data structure exploration, dynamic data type refinement and placement frameworks that target heterogeneous DRAM/NVM memory systems over different application domains.*

The deployment of applications derived by different application domains, such as databases, in-memory analytics and ML-based workloads, over heterogeneous DRAM/NVM systems tends to become the mainstream trend both in the academic and industrial community, aiming to take full advantage of the underlying infrastructure and provide optimizations both on performance and energy consumption. However, the effective co-utilization of both hardware and software requires a set of explorations and optimizations, ranging from

indexing structure design and dynamic data type organization and configuration, workload mapping, configuration, and deployment to multi-tier hybrid memory systems and data placement. While prior works have investigated portions of these challenges, none of them has proposed an end-to-end flow for co-designing the aforementioned aspects. In the scope of this thesis, we provide an end-to-end optimization flow for performance, energy and NVM lifetime-aware dynamic data structure and indexing refinement and placement over heterogeneous DRAM/NVM systems.

The proposed solutions extend prior research by:

- Proposing multi-objective optimization and exploration of dynamic data structures and B+-tree indexes over heterogeneous DRAM/NVM memory systems;
- Deriving different solutions and extracting trade-offs among performance, energy and NVM lifetime;
- Providing extensive and in-depth analysis of the inherent characteristics of different data structures and indexes that affect placement over heterogeneous DRAM/NVM systems;
- Presenting workload characterization of in-memory applications deployed on multi-tiered heterogeneous memory systems.

1.3.4. Hardware-Level Page Placement and Configuration

Addressed problem: *Limitations in page placement frameworks and cache subsystem co-design over heterogeneous DRAM/NVM hardware systems.*

With the rapid growth of emerging technologies, the adaptation of sophisticated memory management techniques close to the end hardware is essential. Page placement is the de-facto approach for storing and loading data, i.e. pages, from the main memory. In the context of heterogeneous DRAM/NVM systems, page placement becomes more and more challenging, as the placement of a page strongly affects performance, energy consumption, and the NVM's wear. Moreover, a placement policy needs to operate efficiently with minimal overhead, thus ML-based solutions have been introduced. Even though prior research has examined different page placement policies that rely on ML solutions, none of them has examined the combination of multiple ML models for optimizing page placement in terms of performance and/or energy consumption. Additionally, the co-exploration of main memory optimizations over state-of-the-art processors and effective

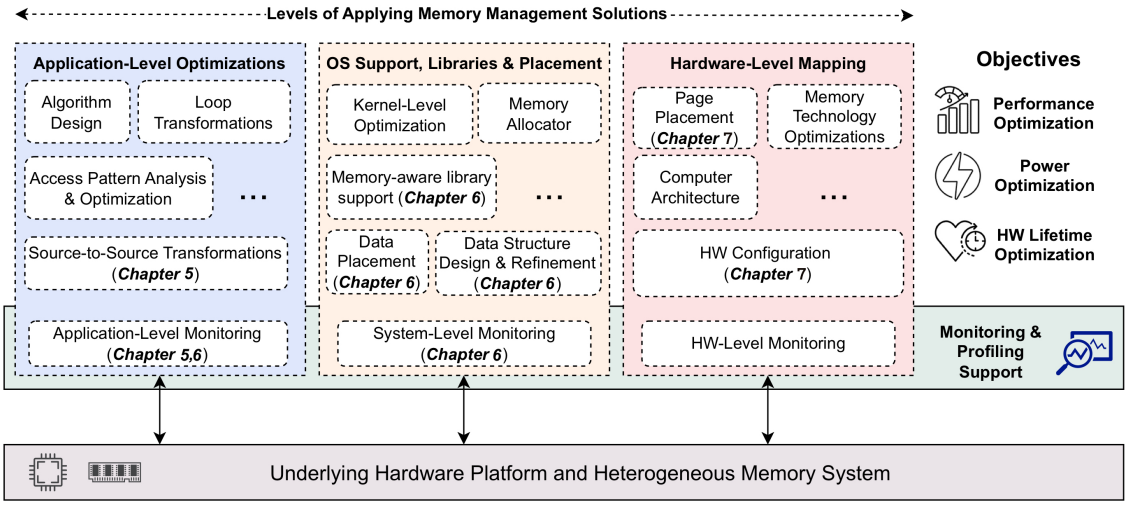


Figure 1.7.: Schematic overview of the contributions of this thesis across the levels of applying memory management on the hardware-software continuum.

cache configurations is also considered.

The proposed solutions extend prior research by:

- Providing an extensive analysis on the impact of page placement over heterogeneous DRAM/NVM memory systems
- Delivering an adjacent LSTM-based predictor, enhanced with knowledge of pages with behavioral similarity in terms of memory access patterns, based on k-nearest neighbors clustering, to be deployed on top of heterogeneous memory systems;
- Developing a cache configuration exploration framework for boosting dynamic main memory management techniques.

1.4. PhD Thesis Structure and Organization

Figure 1.7 presents an abstract and high-level overview of the positioning of the contributions of this thesis concerning the different optimization layers across the hardware-software continuum. The rest of this PhD thesis is organized as follows:

- **Chapter 2** provides a brief background on dynamic memory management and the major fundamental principles of recent memory technologies.

- **Chapter 3** presents an extended literature review of the state-of-the-art proposed solutions, challenges, and limitations on the distinct levels of applying memory management solutions. Moreover, it summarizes the major contributions and the research gaps that this thesis covers.

- **Chapter 4** provides an extended survey for classification, analysis and discussion of the state-of-the-art of recent data placement over heterogeneous memory systems, while providing generic guidelines for placement solution design.

- **Chapter 5** presents a systematic methodology for application-level memory management and source-to-source transformations for Python workloads based on static and dynamic analysis.

- **Chapter 6** illustrates an end-to-end optimization approach for dynamic data structure and indexing design and exploration, application mapping, and software data placement over heterogeneous DRAM/NVM memory systems.

- **Chapter 7** focuses on the hardware level and presents an LSTM-based methodology for page scheduling over heterogeneous DRAM/NVM memory systems and memory management boost over effective cache configuration over emerging architectures.

- **Chapter 8** concludes this PhD thesis, summarizes the presented results and outcomes, and discusses the future extensions of this work.

- **Appendix A** examines the problem of task offloading in Edge computing systems and proposing DMRM, a new algorithm based on economic and pricing models for dynamic resource management of IoT networks under CPU, memory, bandwidth and latency constraints..

- **Appendix B** examines the problem of task offloading in Edge computing systems and proposing SGRM, a game theory-based framework for dynamic resource management of IoT networks under CPU, memory, bandwidth and latency constraints.

- **Appendix C** presents Road-RuNNer, a framework for DNN partitioning and offloading for edge computing systems. RoaD-RuNNer relies on its prior knowledge and leverages collaborative filtering techniques to quickly estimate performance and energy requirements of individual layers over heterogeneous devices.

- **Appendix D** proposes a novel methodology for CNN deployment on the STM32 family, focusing on power optimization through effective clocking exploration and configuration and decoupled access-execute convolution kernel execution.

Chapter 2.

Background and Fundamentals of Dynamic Memory Management

2.1. Background on Dynamic Memory Management

In this chapter, we present a brief background on the fundamentals of dynamic memory management, through pinpointing challenges in emerging memory technologies and architectures and highlighting the importance of static and dynamic memory management techniques.

2.1.1. Importance & Mechanisms of Static and Dynamic Memory Management

Memory management in recent systems can be performed in two discrete ways: i) *Static Memory Management* and/or ii) *Dynamic Memory Management*. Static and dynamic memory management are two contradicting approaches in handling memory resources within a computing system.

Static Memory Management: In static memory management, the memory allocations and de-allocations are determined, configured and defined during the compilation phase. This approach supports predictability and simplicity, however, they lack flexibility, as the allocated memory cannot adapt to changing runtime conditions. Static approaches have been examined in the past for memory management [14–21]. Nevertheless, static approaches tend to over-allocate resources and are not adaptive to run-time changes [107], as the static data type handling which is too pessimistic, thus worst-case scenarios are assumed. Thus, as dynamic applications with dynamically allocated data types that drive a

large set of workloads, such as graph analytics and graph data-bases, genomics, molecular dynamics, AR/VR etc., are dominating in current data processing and high-performance platforms, the integration of static memory management as a solution cannot be considered as an economically desirable option. However both in industrial and academic context static memory management solutions are still employed in recent systems, due to their simplicity on the integration and the extended tool support. Last, the integration of emerging memory technologies in state-of-the-art platforms, which are sensitive to the decisions propagated from the above layers, requires effective solutions that consider their specific characteristics.

Dynamic Memory Management: Dynamic memory management solutions involve allocating and deallocating memory resources during the program's runtime. Such approaches allow for greater flexibility, adaptation to systems with limited memory resources and enabling the generation of data structures that dynamically grow and shrink. Even though dynamic solutions impose high risk in terms of leakage, where memory is not properly released, and fragmentation, which can significantly impact memory utilization, their integration into state-of-the-art systems is inevitable. Typical dynamic applications operate on dynamic data structures, like tensors, graphs, trees, hash tables, and more [42–52], while a wide variety of applications, namely dynamic irregular applications, have very frequent and unpredictable (i.e., not statically known a priori) data-dependent memory accesses and control flow [108,109]. This dynamicity in terms of data access patterns and the amount of data produced that needs to be kept alive in the main memory of a system pushes further the necessity of effective memory management at run-time [110–112]. Furthermore, the special characteristics that new technologies impose, such as the limited write endurance of NVM memories [113–115], require even more fine-tuned solutions in order to minimize the negative impacts that dynamic memory management can potentially have, in terms of performance, energy consumption, and hardware's lifetime. In the scope of this thesis, we focus on dynamic memory management.

Dynamic memory management solutions can be integrated over multiple levels of the computing hierarchy. At the application level, dynamic memory management involves strategies employed by the software to effectively allocate and deallocate memory during runtime. Multiple solutions, such as the integration of access pattern analysis mechanisms, the optimized allocation of memory objects in the critical source code parts, the design of application-level data types and source-to-source transformations can significantly affect the overall memory consumption and behavior. Going more in depth in the computing hierarchy, i.e. on the operating system-level and libraries, dynamic memory management solutions can be integrated. Memory placement strategies, determine where in the memory allocated data should reside, while system libraries provide interfaces for dynamic memory management functions and may include dynamic data structures, memory pooling or allocators to enhance the efficiency of memory allocation.

On the hardware level, dynamic intra-memory management involves the interaction between the CPU and memory subsystem during runtime, including various techniques such as cache management algorithms, memory prefetching, and memory access scheduling, aimed at optimizing memory access, allocation, and utilization. Page placement mechanisms and replacement policies form typical approaches to handling data on the hardware level. More specifically, in novel heterogeneous memory systems, such kind of solutions become more and more crucial for the effective performance of computing systems. A well-coordinated approach at the application, system, and hardware levels ensures that dynamic memory is utilized effectively, providing high performance, sustainability and improvements in the hardware's wear.

2.1.2. Rise of Emerging Memory Technologies & Heterogeneous Memory Architectures

The memory sub-system in HPC systems and supercomputers is one of the main components that limit the scalability and contribute to the rise of energy consumption [116]. Traditional memory technologies, including DRAM and SRAM are widely used as main memory and on-chip cache in memory hierarchies of computing systems. Even though traditional DRAM technologies provide low access latency the increased leakage power and refresh pore requirements impose significant limitations in the scalability of DRAM, thus imposing significant challenges towards reaching increased exascale performance. Moreover, integrating more DRAM modules, aiming to perform more complex simulations, analytics and in-memory processing is dramatically affecting the sustainability and the maintenance cost of state-of-the-art systems.

Rise of NVM technologies. Research and development efforts in computing systems have recently shifted towards semiconductor memory technologies that aim to complement the existing memory and storage hierarchy, albeit with inherent trade-offs. Non-volatile memory (NVM) technologies, such as 3D-XPoint [24], Resistive RAM (ReRAM) [25], Phase-Change Memory (PCM) [26], Spin-Transfer Torque Magnetic RAM (STT-MRAM) [27], and Ferroelectric RAM (FeRAM) [28], alongside adaptations like Intel Optane Data Center Persistent Memory (DCPM) [29], offer diverse properties including persistent storage, high density, scalability, and zero leakage power. However, it is essential to note that these technologies come with crucial trade-offs; some prioritize higher density but offer relatively poor performance, while others exhibit good performance at the expense of lower density compared to DRAMs. These trade-offs underscore the importance of thorough research to identify optimal design-space exploration strategies and find the true sweet spots in memory technology development. Figure 2.1 illustrates the integration of NVM technologies, also known as Storage Class Memory (SCM), within

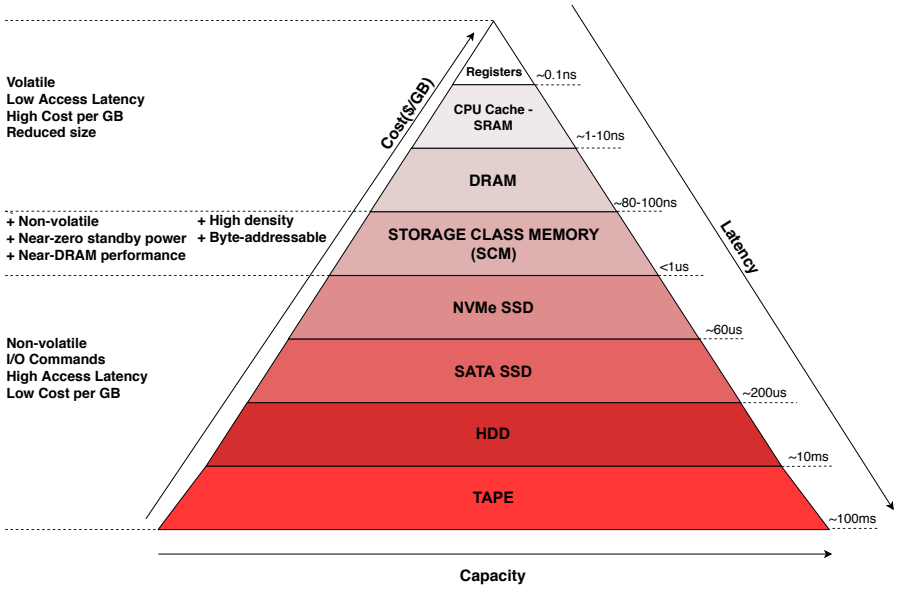


Figure 2.1.: Hierarchy Pyramid of Memory Technologies

the pyramid of the memory hierarchy.

Challenges and limitations of NVM technologies. Despite their advantages, NVM technologies also impose important limitations and challenges that need to be handled. NVM technologies come with high access latency and/or low bandwidth, thus applications deployed on NVM are expected to require higher execution time and/or be latency tolerant. Moreover, they suffer from asymmetric read/write properties, as write latency is significantly higher from read latency, while on the same time they impose limitations in terms of write endurance, i.e. they provide limited number of writes per memory cell. Additionally, write operations in NVMs cause increased energy consumption compared to read operations. Table 2.1 illustrates a summary of the specs of different NVM technologies compared to traditional DRAM/SRAM and storage systems, as derived from the existing literature, as industrial specs are not publicly available [117], [118], [119], [120], [121], [122], [123], [124], [125], [126]. Even though, academic specs are limited in terms of their accuracy, they are still the ony publicly available metrics. Moreover, upcoming advances in memory technologies, including DDR5 [127], LPDDR5X [128] and GDDR7 DRAM [129], custom solutions such as HBM-PIM (high bandwidth memory-process in memory) [130], AXDIMM (Acceleration DIMM) [131] and Samsung Memory Expander [132] are expected to further boost the heterogeneous characteristics of hybrid memory systems.

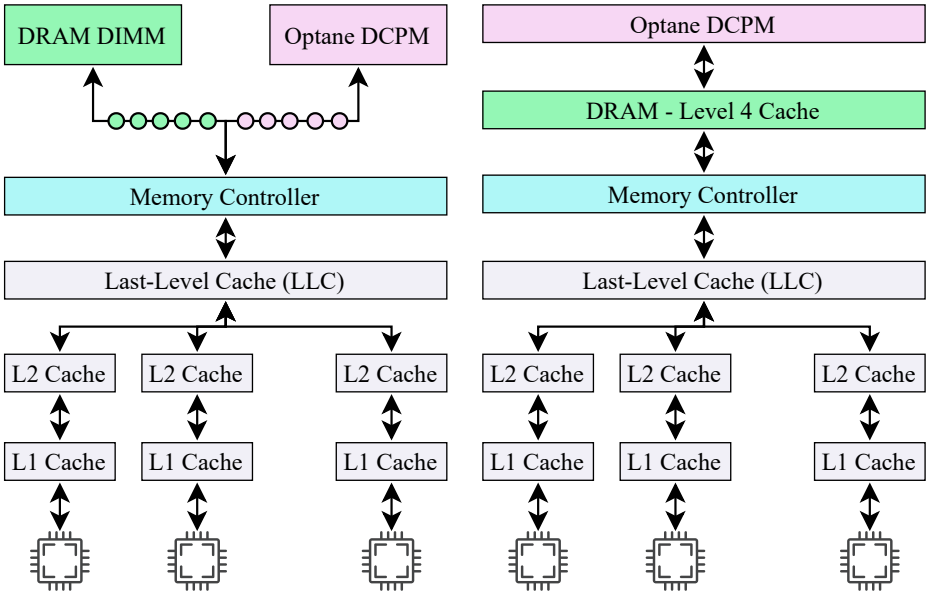
Table 2.1.: Overview of the specs for DRAM technologies, NVM and storage systems.

	SRAM	**DRAM**	**PCM**	**ReRAM**	**STT-RAM**	**FeRAM**	**MRAM**	**NAND Flash**	**HDD**
Read Latency	0.2-2 ns	10 ns	20-85 ns	10-20 ns	5-15 ns	25 ns	5-10 ns	15-35 μs	8.5-9.5 ms
Write Latency	0.2-2 ns	10 ns	150-1000 ns	100 ns	10-30 ns	75 ns	12 ns	200-500 μs	8.5-9.5 ms
Access Granularity	64B	64B	64B	64B	64B	64B	64B	512B	512B
Cell Size (F^2)	100-120	60-100	4-12	4-10	6-50	2-5	32	4-6	2/3
Write Endurance (writes/cell)	10^{16}	10^{18}	10^{8}	10^{10}	10^{12}-10^{15}	10^{15}	10^{15}	10^{4}-10^{5}	$> 10^{16}$
Program Energy/bit	High	2 pJ	100 pJ	2 pJ	0.02 pJ	2 pJ	120 pJ	10 nJ	NA
Leakage Power	High	Medium	Low	Low	Low	Low	Low	Low	Low
Volatile	Yes	Yes	No	No	No	No	No	No	No

Rise of heterogeneous and hybrid memory architectures. Next-generation HPC computing systems are expected to provide multiple tiers of memory storage, adopting heterogeneous configurations within individual hierarchy layers and across multiple layers. This approach combines a small amount of on-package memory with conventional DRAM and large amounts of non-volatile memory, offering diverse trade-offs and new design points for super-computing systems. In particular, two flavors of heterogeneity are prevalent: within one hierarchy layer, exemplified by hybrid configurations such as DRAM+IGZO in main memories, and across several layers, where heterogeneity extends between main and Storage Class Memory (SCM). Commercial platforms and state-of-the-art databases and in-memory systems, such as SAP HANA [133] and Memcached [134], as well as novel exascale supercomputers like Aurora [31] employing the DAOS architecture for storage [135], integrate Optane DCPM alongside conventional DRAM technology. This integration of new NVM devices with conventional DRAM addresses power, area, and cost challenges for large-scale systems, making heterogeneous memories an efficient solution across diverse computing environments.

2.1.3. Industrial Memory Technologies

In this subsection, we go beyond the proposed academic memory technologies and focus on industrial memory technologies. Taiwan Semiconductor Manufacturing Co. (TSMC) has introduced a set of libraries consisting of numerous pre-designed and pre-characterized semiconductor intellectual property (IP) blocks that are provided by TSMC [136]. Such kind of libraries contain standard cell designs, memory elements and other components that semiconductor designers can utilize as building blocks to create integrated circuits (ICs). These processes involve the fabrication of silicon wafers with specific technologies, such as 7nm, 5nm, or other nanometer-scale processes. The libraries include a variety of



(a) App Direct Mode **(b)** Memory Mode

Figure 2.2.: Operating modes of Intel Optane DCPM

standard cells, which are basic logic gates and flip-flops, as well as memories like SRAM and other commonly used components. Moreover, advances in SRAM memories have been introduced [3, 137, 138]. Furthermore, regarding DRAM progress, various vendors, such as Samsung [1] and Micron [2], continuously provide advances in DRAM technologies, while DRAM simulators, such as [139] have been established in order to support a wide array of DRAM standards, e.g. DDR3/4, LPDDR3/4, GDDR5, WIO1/2 and HBM. Regarding the NVM technologies, Intel Optane DCPM is the only commercial memory module up to now.

Overview of Intel Optane DCPM. The Intel Optane DCPM is the first and only commercially available NVDIMM developed by Intel and Micron. Persistent memory DIMMs co-exist with volatile memory DRAM DIMMs. It relies on 3D-XPoint technology and combines byte-addressability alongside with data persistence. The Optane DCPM can be configured in two operating modes, as depicted in Figure 2.2, i.e. the *AppDirect* (Fig. 2.2a) or the *Memory mode* (Fig. 2.2b).

- ***Memory Mode:*** In *Memory Mode*, the Optane DCPM operates as the main

[1] Samsung Memory Technologies
[2] Micron Memory Technologies

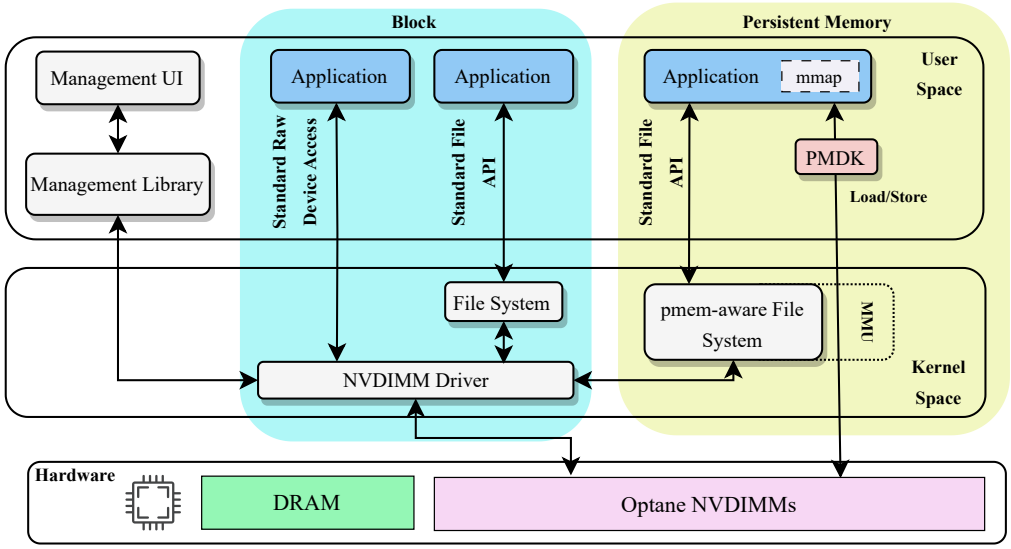


Figure 2.3.: Programming model of Intel Optane DC Persistent Memory

memory space without persistent properties. In this mode, the DRAM is used as an extra layer of cache on top of the Last-Level Cache. In this case, the memory management is handled by the system's memory controller and all data objects are allocated to Optane DCPM by default, transparently to developers.

- ***App Direct Mode:*** In the *AppDirect Mode*, Optane DCPM operates as a persistent storage and provides access to the storage space similar to memory-mapped file operations. The persistent memory and DRAM DIMMs act as independent memory resources under direct load/store control of the application. The application developers can rely on the Persistent Memory Development Kit (PMDK) for persistent memory allocations [140]. The persistent memory is utilized as byte-addressable memory and can be directly accessed by the executed applications.

Programming Model of Intel Optane DCPM: Figure 2.3 illustrates the programming model of Intel Optane DCPM [3]. The access on Optane Non-Volatile DIMMs (NVDIMMs) is performed through the NVDIMM driver when the former is configured on *Memory Mode*. Moreover, when the Optane is configured in *App Direct* mode either by pmem-aware file system by using the standard file API or by the *PMDK* and supporting Direct Access (DAX) for performing load/store operations.

[3]Intel Optane DCPM programming model

Chapter 3.

Review of State-of-the-Art Challenges and Limitations

3.1. Application-Level Dynamic Memory Management

3.1.1. Related Work

Several works in the existing literature propose dynamic memory management optimizations for embedded systems. Authors of [141] and [142] propose the Dynamic Data Type Refinement (DDTR) methodology, which enables the systematic customization and refinement of dynamic data structures for embedded applications, in terms of performance, memory footprint, and energy consumption [143]. A complementary methodology is the Dynamic Memory Management (DMM), which proposes customized dynamic memory allocators, to meet the application requirements and embedded system constraints, in terms of performance and memory footprint [144]. However, these approaches have been developed for C/C++ embedded applications, while Python and other modern high-level languages are not supported.

A few recent works propose dynamic memory management optimization methodologies targeting the elimination of memory leaks and fragmentation. For example, in [145] the authors leverage machine learning techniques for the automatic detection of memory leaks in C/C++ codes, while in [146] is proposed a replacement for malloc/free aiming to eliminate the memory fragmentation of C/C++ applications. A similar approach utilizes machine learning techniques to reduce the memory fragmentation in C++ server workloads, induced by long-lived objects allocated at peak heap size [147]. The aforementioned approaches target C/C++ applications, and they cannot be directly mapped to Python applications, due to the different structure of the memory allocation in dynamic languages, compared to the static.

Research Goal	Related Works
C/C++ Memory Management	[141], [142], [144], [145], [146], [147], [148]
Qualitative/Quantitative Analysis	[149], [150], [151]
Interpreter-oriented Optimizations	[105], [106]
Novel Garbage Collectors	[37], [38], [39]
Library and Application-specific Optimizations	[34], [35], [36]

Table 3.1.: Summary of related research for Python memory footprint optimizations

With regard to Python, several works focus on analyzing Python's behavior and dynamic memory management performance. The authors of [149] provide a detailed quantitative analysis of the overhead in Python without and with just-in-time (JIT) compilation and they identify the existing memory overheads due to Python's interpreter and built-in memory manager, while also discussing the limitations of Python's garbage collector. Similarly, the authors of [150] study the Python's interpreter and calculate the overheads of some of the language's features such as the dynamic typing and reference counting. These works focus on identifying the overhead of Python features in terms of performance and memory and they do not propose optimization techniques. Therefore, they can be considered as a first step towards a more through understanding of Python's inherent limitations in terms of dynamic memory management.

Regarding the dynamic memory management optimizations for Python, the authors of [35] propose a Python package for reducing the performance and memory overhead of loading, traversing, and manipulating tree data structures in Python applications. The authors of [36], propose a Python-based optimization framework for high-performance genomics, that combines the advantages of high-level languages such as Python with the performance of lower-level languages such as C/C++. In order to overcome the inherent limitations of the Python interpreter and dynamic memory management approach, the work in [105] proposes an alternative bytecode interpreter, based on CPython. Aiming to enable the efficient deployment of Python applications in embedded systems and micro-controllers, the authors of [106] introduce a run-time system that allows the execution of high-level languages and they discuss the static memory allocation behavior of Python. Additionally, the work in [37] proposes an abstract garbage collector (AGC), which detects and removes unreachable abstract addresses and works as a tracing garbage collector. A similar approach in [38] describes a dynamic memory leak detection method for dynamic languages. The work in [39] introduces a garbage collector, which is enabled by machine learning methods to allow efficient memory deallocation. The authors of [152] propose a hardware-software co-optimization dynamic memory management scheme for dynamic languages, including Python.

3.1.2. Key differences and extensions from the State of the Art

In the scope of this Ph.D. thesis the major key differentiation and novel contributions are the following:

- Although the existing research has highlighted the effectiveness of memory footprint optimization for Python libraries, no work to date, to the best of our knowledge, has proposed a memory footprint optimization framework that specifically targets Python applications deployed on edge computing devices. Thus, the aforementioned approaches that target Python are considered complementary to the optimizations proposed in this work;

- We deliver an automated framework, that provides static and dynamic analysis-based optimizations and we present in-depth insights on critical Python parameters that influence the memory consumption.

3.2. Data Structure, Indexes Design and Placement for Heterogeneous Memory Systems

3.2.1. Persistent Memory Data Structure/Indexes Optimization and Design

Several works have been conducted aiming to (re-)design different data indexes for persistent memories. Prior research has been conducted on the optimization of data structures at the application-level, based on design-space exploration [144] or leveraging AI techniques [153]. B+-tree [42–48] and hash-based [49–52] indexing data structures and key design challenges have been widely considered as the most dominant candidates for porting recent database workloads on DRAM/NVM memory systems. Authors of [42] focus on the conversion of concurrent DRAM indexes to persistent-memory indexes. In [154] a framework for implementing persistent data structures on asymmetric NVM is proposed. Moreover, as database indexes existed prior to the integration of NVMs to existing HPC systems, works such as [155] and [42] propose techniques and guidelines to effectively transform existing DRAM indexes to persistent memory indexes. Furthermore, in [156] propose a key-value store design focusing on persistent memories. Similarly, in [157] a library to integrate persistence to volatile data structures is presented. These researches have produced a substantial amount of persistent memory indexes that can be potentially utilized for database workloads.

3.2.2. Evaluation and exploration of persistent memory indexes

The growth of the number of persistent memory indexes has exploded the complexity of selecting sufficient indexes for database workloads. Authors of [158] evaluate B+-Tree-based range indexes focusing on performance-related aspects. Similarly, authors of [159] provide an experimental evaluation on persistent memory hash indexes. Furthermore, authors of [160] aim to investigate the impact of persistent B+-trees on the energy consumption of Intel Optane DCPM. Moreover, in [161] authors provide a data structure exploration methodology, however without focusing on database workloads and persistent memory indexes.

3.2.3. Data Placement over Heterogeneous Memory Systems

Various works have been conducted the provide data placement over heterogeneous memory systems. These works provide either i) general purpose placement policies working on different granularities, such as [88–98], mostly focusing on performance and/or energy consumption optimization, or ii) application-specific solutions, including big data and databases [53–56, 66, 67, 99], deep learning applications [68–70, 100], graph analytics [57, 63–65, 101] and HPC applications [58, 59, 71]. An extensive analysis, classification and categorization of the aforementioned data placement works across the literature is detailed in Chapter 4.

3.2.4. Key differences and extensions from the State of the Art

In the scope of this Ph.D. thesis the major key differentiation and novel contributions are the following:

- We propose a novel exploration methodology for the effective exploration and selection of persistent memory indexes. We target multi-objective optimization problem, aiming to co-optimize performance, energy and NVM-lifetime;
- We integrate a third level of optimization objectives, beyond performance and energy consumption, for investigating the impact of high-level data organization optimizations on the NVM's lifetime;
- We propose an extensive evaluation of Intel Optane DC persistent memory for applications consisting of persistent B+-tree indexes, thus highlighting the impact

of the latter on the underlying hardware.

- We propose a dynamic memory management methodology that combines an NVM-aware dynamic data structure refinement step, as well as an algorithm for boosting object placement over heterogeneous memory systems. We aim to optimize 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;

- We integrate data placement solutions that provide a set of Pareto optimal data structure placement solutions for heterogeneous DRAM/NVM systems in terms of performance, energy consumption, and impact on the NVM's write accesses;

- Our methodology is enhanced with a decision support flow, aiming to extend existing placement solutions by providing objective-weighted decision-making, based on user-defined thresholds and composing a multi-objective optimization problem. The outcome of our methodology is a set of solutions that co-optimize the overall performance, energy consumption and boost the NVM's lifetime.

3.3. Near-Hardware Solutions

3.3.1. Related Work

Various researches have been conducted aiming to evaluate the overall behavior of NVM technologies and Intel Optane DCPM. The existing works related to the NVM and Optane DCPM can be loosely classified into three categories:

- Empirical evaluation of the Optane DCPM performance through microbenchmarks and/or applications and investigation of challenges and limitations [42,116,158,162–167].

- New applications and data structures optimized for the Optane DCPM as persistent storage (mainly novel tree-based data structures) [43, 168]

- Novel workload schedulers, data placement algorithms and other runtimes for the Optane DCPM [161, 169, 170].

Recently, the Optane DCPM was extensively evaluated using real-world scientific appli-

cations. The typical hardware configuration applied in these works was the use of the Optane DCPM as the main memory, while DRAM was used as a cache. The persistent features of the Optane DCPM are not available in this case. Recent works conclude that the performance gap between an Optane DCPM and a DRAM-only system is relatively small for this hardware configuration [164, 165]. Works that entirely replace DRAM with Optane DCPM (i.e. DRAM was entirely deactivated) highlight the fact that there is a significant performance overhead, due to the Optane DCPM bandwidth saturation for real-world multi-threaded applications [116, 162].

In contrast to the aforementioned works, using Optane DCPM as a persistent memory requires changes in the application source code. Additionally, the API for placing data on an Optane DCPM is available for dynamically allocated data structures only (e.g. through *libvmmalloc*) [163]. Therefore, this configuration is only partially evaluated in the existing literature using real-world scientific applications [165,166]. The majority of works evaluate the Optane DCPM in this configuration either using micro-benchmarks, such as the Stream Triad or mini-applications [163]. A more extensive evaluation, considering both the performance and energy consumption is available in the literature, but targets graph applications specifically [165]. On the other hand, there are several works which evaluate B+ tree indexing data structures on the Optane DCPM, as a persistent memory. They typically compare indexes in terms of application-level performance-related metrics, such as throughput and latency [42, 158, 167].

Furthermore, several works have been conducted, aiming to target the problem of data placement over hybrid DRAM/NVM systems. Authors of [171] propose a runtime framework for adaptive granularity data placement for graph-based applications. Similarly, authors of [172] implement a set of libraries for object placement on heterogeneous memory systems. In [173] a memory management scheme for disaggregated memory systems is presented, aiming to support migration and hot/cold pages identification. Moreover, authors of [174] propose NUMA-aware hybrid allocation strategies for latency-sensitive and bandwidth-sensitive applications. In [175] authors design an object placement mechanism to reduce write traffic on NVM.

ML-based approaches have been proposed, in order to tackle the problem of object/page characterization and placement. Authors of [74] propose leveraging Recurrent Neural Networks (RNNs) for predicting memory access patterns, in order to improve memory prefetching. In a similar perspective, the authors of [75] propose a neural network based hardware prefetcher by predicting the access patterns of applications with linked data structures, compressed data formats and data dependent control flows. In [76], authors design a page scheduler for heterogeneous DRAM/NVM systems, based on deep learning techniques, namely *Kleio*.

Furthermore, dynamic memory management can be optimized through effective cache configuration. Prior research has investigated the impact of architectural configurations from various aspects. Authors of [77] investigate the impact of hardware technology configurations for three-level hybrid cache systems. In addition, authors of [78] introduce an exploration framework focusing on the microarchitecture configuration of the RISC-V BOOM processor, while in [79] a reinforcement learning-based approach is proposed for exploration. Moreover, authors of [80, 81], the effective coherence configuration for multi-core RISC-V cores is examined.

3.3.2. Key differences and extensions from the State of the Art

In the scope of this Ph.D. thesis the major key differentiation and novel contributions are the following:

- We conduct an extensive experimentation and analysis focusing on the evaluation and effective configuration of Optane DCPM, highlighting key insights and providing guidelines;
- We propose and implement an adjacent LSTM-based predictor, enhanced with knowledge of pages with behavioral similarity in terms of memory access patterns, based on k-nearest neighbors clustering;
- We integrate a more sophisticated main memory replacement policy, namely *clustered-LRU*, aiming to cluster pages based on their access pattern;
- We further integrate a novel systematic exploration methodology for co-design application-level memory management and cache configuration over RISC-V architectures to provide architects with decision support. Our methodology envisions the performance optimization of RISC-V processors while minimizing hardware overhead and cost.

Chapter 4.

Classification and Analysis of Data Placement over Heterogeneous Memory Systems

The rapid growth of emerging heterogeneous memory systems has led to the enormous rise of data placement solutions deployed on such systems. Effective data placement solutions rely on a set of decision making steps during their design process, ranging from the optimization objective to the granularity of the placement entities. Towards this direction, in this chapter we present an extensive survey aiming to: i) classify the existing literature based on various design features and parameters and highlight research trends and approaches per application domain, ii) provide an extensive discussion and qualitative comparison of the existing methodologies, aiming to reach conclusions and extract recent guidelines on data placement, while deriving a trade-off design space, indicating the key design choices for application specific data placement and iii) highlight existing gaps in the literature and provide future guidelines and directions based on how existing placement approaches can be extended and adjusted to upcoming memory technologies.

This chapter's text strongly relies on the following publication:

- *Katsaragakis Manolis, Christos Baloukas, Lazaros Papadopoulos, Francky Catthoor & Dimitrios Soudris. Software-based Data Placement for Heterogeneous Main Memory Systems: A Systematic Survey (To be submitted o ACM-CSUR).*

4.1. Introduction

The rise of heterogeneity in terms of memory architecture leads to a complex paradigm, where data can be either placed on both fast and/or slow memory, as each kind of memory exposes different characteristics on the software lying on top of the hardware. Despite the fact that heterogeneous main memory systems can provide significant performance benefits, the placement of data is challenging, as it composes a multi-dimensional exploration problem, where trade-offs are derived between various factors such as performance, memory bandwidth, capacity, latency, energy and cost [161]. Aiming to fully exploit these alternative characteristics, efficient data placement approaches are required. Such kind of placement can be achieved by both hardware and software approaches. We focus on software-based data placement approaches, due to the fact that software-based techniques are quite modular and portable over various heterogeneous systems, without imposing any extra hardware cost. Software data placement can be performed either by special middleware components or it can be performed directly by the application through explicit memory selection. The aforementioned placement decisions are propagated to the hardware layer to be mapped in the physical memory system. We focus on the software layer.

An extensive survey of application-specific placement literature can boost the existing research, expand the limits of data placement approaches and rethink key design aspects. To the best of our knowledge, this is the first work that surveys the research area of application-specific software data placement approaches for heterogeneous memory systems. Table 4.1 presents an overview of the NVM-related and data-placement related surveys. Several works have been conducted aiming to provide robust classification of various aspects of heterogeneous memory systems, across the hardware/software continuum. The majority of these works target recent advances in heterogeneous memory systems and NVM technologies [176–179], architectural NVM advances [180, 181], security and fault tolerance [182]. Regarding software design over heterogeneous memory systems, [183] targets software techniques for NVM technologies, while in [184] file system design options are examined. Moreover, database design techniques over NVM-based memory systems are examined in [185]. Finally, the authors of [186] survey data placement approaches in the Cloud.

Throughout the existing literature, to the best of our knowledge, there is no work, up to date, dedicated to survey the existing software data placement approaches in heterogeneous memories. The works of Table 4.1 can be considered as complementary to the present survey. However, memory heterogeneity is expected to be a prevalent characteristic, especially in modern data centers to address the memory wall [187]challenge. In addition, more NVM-based commercial memory products are expected to be-

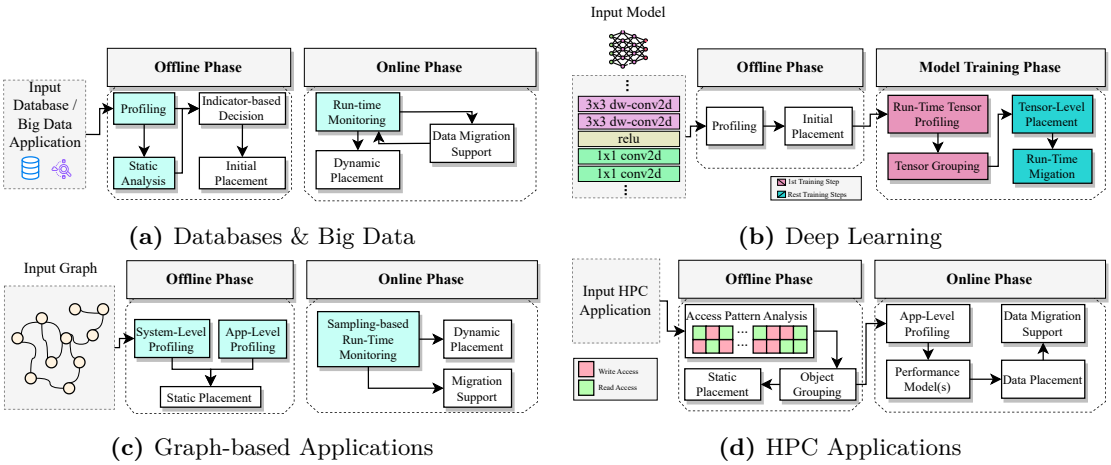
Related Surveys	References
Recent NVM Trends	[176–179]
Software techniques for NVM	[183]
Architectural NVM advances	[180, 181]
Security and Fault tolerance	[182]
Databases and Big Data on NVMs	[185]
File systems on NVMs	[184]
Data placement on the Cloud	[186]
Software application-specific data placement	This chapter

Table 4.1.: Classification of existing NVM-related and data placement related surveys.

come available soon, so the need for efficient, advanced and user-friendly data placement approaches is evident. This work provides a systematic review of software-based domain-specific data placement over heterogeneous main memory systems, which aims to:

- Classify the existing literature based on various design features and parameters and highlight research trends and approaches per application domain.
- Provide an extensive discussion and qualitative comparison of the existing methodologies, aiming to reach conclusions and extract recent guidelines on data placement. We also derive a trade-off design space, indicating the key design choices for application specific data placement.
- Highlight existing gaps in the literature and provide future guidelines and directions based on how existing placement approaches can be extended and adjusted to upcoming memory technologies.

The rest of this Chapter is organized as follows: Section 4.2 provides an extensive presentation of the general purpose and application-specific data placement solutions as met in the literature. Section 4.3 shows an extensive analysis and classification, while in Section 4.4 we present the Design Space of data placement approaches, as derived by the state-of-the-art research. Finally, Section 4.5 highlights data placement policies summary and indicates gaps in the existing literature and provides future extensions and guidelines regarding emerging data placement on discrete application domains and upcoming memory technologies.



(a) Databases & Big Data **(b)** Deep Learning

(c) Graph-based Applications **(d)** HPC Applications

Figure 4.1.: Overview of the application-specific data placement approaches for (a) Data bases & Big Data, (b) Deep Learning, (c) Graph-based and (d) HPC workloads.

4.2. General Purpose and Application-specific Placement

The target works are grouped into four different application domains, as met in the literature. The major domains are: i) `Databases & Big Data` , ii) `Deep Learning`, iii) `Graph-based` and iv) `High-Performance Computing (HPC) Applications`. Emerging application domains, are not considered, as they have not been widely established yet. We present an in-depth analysis of the major design choices and features of the corresponding application domains. Our main objective is to derive general approaches per domain, investigate special characteristics and indicate existing trends and research gaps.

4.2.1. General purpose placement

These works can be classified based on the placement granularity level: data structures, memory objects or memory pages [188] [189] [166] [190]. Each approach has different advantages and drawbacks. The authors of [188] 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 [189]. At data structure placement granularity, write-aware data structure placement for cached and uncached

NVMs is proposed in [166]. AI-based data placement at page granularity is proposed in [190].

4.2.2. Databases and Big Data

Big data [56, 66, 67, 99] and databases [53–55] compose one of the most dominant application domains, in terms of the number of works conducted for software data placement. Big data frameworks like Spark often use a collection of standard data types to store data that are known beforehand by the developers. These data types can be a prime target for optimization as they often constitute the biggest bottleneck in dynamic memory management. For instance, as mentioned in [66], Apache Spark uses Resilient Distributed Datasets (RDDs) as its data abstraction. Each RDD is a collection of Java objects each representing a data tuple and developers can choose the target memory technology, where each RDD will be allocated on and therefore it is straightforward to make custom data placement decisions based on the available memory resources. Figure 4.1a illustrates the placement flow, as summarized through the literature. Typically, there exist two phases for data placement: i) the *Offline phase*, which performs an initial static analysis of the code to identify optimization opportunities and ii) the *Online phase*, which takes advantage of run-time monitoring to fine-tune placement decisions and support migration between the various memories.

Offline Phase: Offline phase typically consists of a static analysis [53–55] of the application's code, where the target structures are identified and marked with specific tags for placement in either DRAM or NVM. Aiming to derive more fine-grained information regarding target data, a profiling step is triggered and is performed using tools, such as Intel VTune Profiler [66, 191], or custom counters to measure memory accesses to specific memory regions [56]. CPU events through hardware performance counters are also considered as a profiling alternative [56, 66]. Based on the derived profiling and analysis metrics, initial placement is performed. The placement approaches is presented in the Online Phase.

Online Phase: The Online phase consists of making the actual data placement using various mechanisms, as met throughout the literature. Heuristic-based placement [54, 55] and approaches based on analytical cost models [53] have been utilized. Moreover, the model of data placement based on hotness/coldness grouping has been also considered [66, 67]. Other works, such as [66], use garbage collection mechanisms to allocate data to the proper memory, while a custom memory allocator [56] that intervenes between data allocations and the OS has also been considered as placement approach. Regarding migration support, applications that target big data provide run-time mi-

gration mechanisms [56, 66, 67, 99]. Typically, placement is performed either on data structure and coarse-grained (e.g. database tables) granularity [53–56], data chunks, such as RDDs [66, 67] and page granularity [99]. Data migration is usually supported with a custom hot/cold mechanism that tries to identify changes in the access frequencies of a specific dataset and moves it to the appropriate memory, respectively. Hot placement units are migrated to the DRAM while cold units are moved to the NVM.

4.2.3. Deep Learning

Several software data placement methodologies for heterogeneous memory systems have been introduced aiming to target Deep Learing (DL) applications [68–70, 100]. Figure 4.1b summarizes an overview of the data placement flow as derived by all DL-specific approaches. Throughout the state-of-the-art literature, we observe that all works target DL training aiming to provide performance optimization, however none of them considers the inference phase. Moreover, placement is performed at tensor granularity, apart from [69], which groups tensors in pages and then pages are placed in each memory, respectively. The examined works target DRAM/NVM [68, 69, 100] systems and/or DRAM/HBM [69,100] systems. Deep learning approaches consist of both *Offline* and *Online* decision making.

Offline Phase: Offline phase consists of a profiling step and an initial placement decision, i.e., how the data will be placed at the beginning of the execution phase. Both authors of [100] and [70] integrate custom profiling, aiming to monitor application-level characteristics. Based on the profiling metrics initial placement is performed. In case that no initial placement logic is integrated, data are stored on the main DRAM memory by default. In [70] an initial placement is performed, which is re-considered during online phase, while in [100] a static(offline)-only placement is proposed, however an online placement is also considered as an alternative.

Online Phase: Online phase targets data management, organization and migration throughout the run-time phase of training. As derived throughout the literature [68–70, 100], existing approaches consider static Deep Neural Network(DNN) training, i.e, the DNN does not change across the training phase, and dynamically changing networks are not examined up to now. The static nature of target training DNN has significant impact, as the behavior of the network is expected to remain slightly identical to the first training step, across the whole training phase. Thus online profiling is required only in the first step of the training phase. Placement on [68, 69] is performed based on hotness and liveness of tensors, where short-lived data are stored in DRAM memory, while long-lived

data are stored in NVM. In [100] placement is based on Integer Linear Programming (ILP), aiming to optimize performance under DRAM size constraint. In [70] placement is performed based on NP-Knapsack problem and solved through approximation techniques.

4.2.4. Graph-based Applications

Graphs domain is a special category of applications that require specialized data placement approach. Various works have been conducted targeting graphs workloads [57, 63–65, 101]. Graph applications provide both poor spatial and temporal locality, as a vertex/edge processed is unlikely to be processed again before the remaining graph is processed and the neighboring vertices date are unlikely to share the same cache line [192]. Thus general purpose placement policies cannot perform efficiently and specific placement is required. Figure 4.1c presents an overall flow as derived throughout the existing literature. In contrast to prior domains, existing works consider either offline-only or online-only decision making. The target graphs applications consist of static edges and vertices, thus major graph changes are not expected to be seen throughout the execution. Therefore, placement can be performed only at offline or only at online phase.

Offline-only Phase: Offline decision making typically consists of a profiling step [101] or on application-level analysis [63,64]. The outcome of these processes is the essential factor which is fed as input to the static placement. In order to derive sufficient information, which can be utilized for the upcoming placement decision making, system-level metrics and features profiling can be derived [101]. Supervised learning approach is performed, aiming to group data based on hotness [101] and then investigate placement approaches based on this grouping. As graph analytics utilize big chunks of data, placement is performed at page granularity. Moreover, current researches provide placement based on application-level analysis, without any profiling [63, 64]. More specifically, high-level characteristics are utilized, such as the vertex-cut and organization of data on high level data structures, that can provide information regarding the expected random or sequential access pattern. The placement is performed at graph-vertex granularity. All works bind the corresponding data to each memory, respectively, and no data re-organization is considered.

Online-only Phase: Apart from offline placement, online-only placement is also considered as decision making approach [57, 65]. Aiming to minimize run-time overhead throughout the execution, sampling-based profiling is performed, which relies on hardware performance counters [57] and prediction based on the deterministic behavior of graph applica-

		Profiling & Analysis		Placement		Migration Support
		Offline	Online	Offline	Online	
Domain	Big Data & Databases	[53–55, 66, 67]	[56, 66]	[53, 54, 66, 67, 99]	[55, 56, 66, 67, 99]	[56, 66, 67, 99]
	Deep Learning	[70, 100]	[68–70, 100]	[70, 100]	[68–70, 100]	[68–70, 100]
	Graph Processing	[101]	[57, 65]	[63, 64, 101]	[57, 65]	[57, 65]
	HPC Applications	[58]	[58, 59, 71]	[58]	[58, 59, 71]	[58, 59]
	Programming Models	[62]	[60–62]	N/A	[60–62]	[60–62]

Table 4.2.: Classification of existing literature per application domain based on profiling and placement approach.

tions [65]. Placement on each memory technology is performed either on adaptive data chunks granularity [57], or at single edge/vertex level [65].

4.2.5. HPC Applications

HPC applications are considered as a separate domain, due to their native characteristics. In contrast to previous domains, which are by nature memory-intensive, HPC applications are additionally computation-intensive workloads [193], thus placement techniques should be designed and operate differently. Applications such as sparse tensor contraction [58], large plasma simulations [71] are considered as specific applications that require special placement over heterogeneous memory systems. The placement in HPC applications is fine-grained in terms of object granularity [58, 59, 71]. Similar to graph-based applications(4.2.4), throughout the existing literature, we observe that data placement can be performed both at Offline and/or Online phase.

Offline Phase: Approaches that provide offline placement do not consider any application profiling prior to the initial placement [58]. HPC applications have a set of data objects that have deterministic access behavior [58], i.e. the access pattern type (random or sequential) of the objects can be estimated a priori. Based on such observations, static placement of objects with deterministic access behavior can be performed [58] and data with random access pattern are placed to the fast memory (DRAM), while data with sequential access patterns are placed in the slow memory(NVM), thus eliminating potential performance degradation due to irregular patterns on the NVM.

Online Phase: Online placement typically consists of run-time profiling, placement and mitigation. Based on the special characteristics of the target application, profiling can be performed: i) on a single application iteration [71], ii) on performance counters, aiming to track memory access patterns and correlate the impact of them to bandwidth and latency [59] and iii) on application-level accesses [58]. Typically, run-time placement is

Data Placement Approach	References
Analytical Models & Heuristics	[53–62]
Hotness/Coldness Grouping	[66–71]
Machine Learning	[99, 101]
Integer Linear Programming	[100]
Application-level Placement	[63–65]

Table 4.3.: Classification of application specific data placement literature based on the integrated placement method.

performed based on data hotness as derived from profiling [58] and from performance models [71]. Moreover prediction mechanisms are also used to calculate the performance benefit and cost of placement [59]. Finally, data prefetching [58] and threads [59, 71] are utilized for minimizing run-time migration overhead.

4.3. Analysis, Discussion and Qualitative Comparison

In this section we present an analysis and discussion of the examined works. Our analysis relies on answering critical research questions(RQ). Our main goal is to derive robust outcomes regarding software-based data placement design choices and special characteristics over various application domains.

RQ1: How placement is affected by application domain?

Table 4.2 illustrates an overview of the static or dynamic behavior, regarding profiling, placement and migration decision making. The following observations are derived. Placement in database-related applications [53–55] is performed statically, i.e., no matter whether decision making is performed offline or online, the data persist in the memory where they are placed and no migration option is considered. Typically, databases are designed to operate over static workloads, where data are not expected to dramatically change in the short-term, thus efficient placement can be decided prior to the execution of the target application. In contrast, works targeting Big Data domain [56, 66, 67, 99] provide run-time migration support. Deep Learing-oriented approaches provide online placement and migration support [68–70,100], while a portion of them provides also offline placement [70,100]. Graph-based approaches consider offline-only placement [63,64,101] or online-only placement (with migration support) [57,65]. Last but not least HPC-based placement algorithms provide both combinations of offline and online placement [58,59,71], while in application domains utilizing specific programming models,

only online placement and migration is performed [60–62].

RQ2: What are the most widely algorithms and models utilized for software placement?

Throughout the existing literature, various techniques have been utilized aiming to perform data placement across heterogeneous memory systems, as illustrated in Table 4.3. Analytical models and heuristics [53–62] and placement that relies on data hotness/coldness grouping [66–71] are the most widely utilized approaches for software data placement throughout the examined literature. Other approaches, such as placement based on high-level application specific characteristics [63–65] are considered, while Machine Learning models [99, 101] and Integer Linear Programming (ILP) [100] have been also utilized. Analytical models and heuristics provide instant decision making, however, they are sensitive to memory mis-placements. On the other hand, ML-based solutions require prior sufficient training and potential re-train at runtime, thus adding extra overhead, while placement based on hotness/coldness and ILP require exploration across all data. Finally, placement approaches that rely on high-level characteristics strongly rely on the knowledge of the deployed application.

RQ3: How migration support is implemented and what special techniques are utilized for overhead minimization?

As illustrated in Table 4.2, various works provide run-time migration [56–62, 65–70, 99, 100]. Migration support can be implemented based on different techniques, aiming to combine efficiency and run-time overhead reduction. The most common approach on migration implementation is to integrate a parallel thread for this purpose [58, 59], as an independent entity can handle data movement, while imposing minimal run-time overhead and avoiding stalls of the application. Moreover, special techniques, such as periodic migration [56, 68, 69] have been addressed, in order to minimize frequent migrations.

RQ4: How profiling is performed, what techniques are utilized and how run-time overhead is minimized?

Profiling is a crucial step for the whole decision making, as it acts as a forerunner to the final data placement selection. Table 4.4 presents the different profiling options. Both offline and online profiling can be performed based on the following options: i) Instrumentation, i.e., the collection of detailed information for the target application, ii) Hardware Performance Counters-based profiling, i.e., monitoring a set of special registers of hardware events iii) Sampling-based profiling, i.e., tracking the target program's stack at periodic intervals and iv) High-level Analysis, i.e., analyzing source-code in order to

Profiling Technique	References
Instrumentation	[55, 60, 61, 65, 99]
Hardware Performance Counters	[56, 57, 62, 71, 101]
Sampling	[59, 66, 67]
High-Level Source Code Analysis	[53, 54, 58, 63, 64, 68–70, 100]

Table 4.4.: Classification of existing profiling techniques over the state-of-the-art literature.

derive high-level characteristics. We observe that high-level analysis is the most widely utilized approach. Due to the fact that we focus on application-specific placement, the in-depth knowledge of the overlying application can strongly boost the placement decisions, thus high-level analysis is widely considered. Instrumentation-based profiling [55, 60, 61, 65, 99] and hardware performance counter profiling [56, 57, 62, 71, 101] which can provide a full view of the underlying data behavior, are also considered, focusing specifically on cache misses, memory accesses, lifetime and other. However, such kind of solutions impose overhead and are difficult to scale in high workloads, due to their exhaustive nature. Aiming to reduce the overhead of profiling and monitoring, sampling-based approaches are evaluated [59, 66, 67], which trade some data loss, with scalability. Various tools have been met throughout the literature for profiling, e.g. perf [194], Intel's VTune Profiler [191], Intel's Performance Counter Monitor (PCM) [26] and custom tools built on top of existing frameworks. For instance, in [70, 100] custom profiling tools were integrated in the Tensorflow [195] framework.

RQ5: What kind of granularity levels exist and what advantages does each one impose?

Placement granularity is a crucial factor for data placement, as it strongly affects the overall performance. For instance, placing very small objects to Intel Optane DCPM, triggers phenomena such as write amplification, thus leading to performance degradation. Table 4.5 summarizes the different granularity levels on which placement is executed on, as derived throughout the existing literature. Typically, placement granularity can be either adaptive, i.e., the nature and the size of data placed can vary throughout the execution of the application, or pre-defined, i.e., specific data is considered as basic placement units. Adaptive data placement provides flexible placement, however more sophisticated decision making is required and extra overhead may be imposed. On the other hand, pre-defined data placement reduces decision making complexity, however this can lead to sub-optimal placement. Therefore, various placement units have been considered, i.e., page-level, object-level, data structure-level and tensor-level, or potential mixtures of them.

Placement Granularity					
Adaptive		Pre-defined			
Objects/Chunks	Data Structure	Objects/Chunks	Page	Tensor	Data Structure
[57]	[56]	[55, 58, 59, 62, 66, 67, 71]	[60, 61, 69, 101]	[68, 70, 99, 100]	[53, 54, 58, 63–65, 71]

Table 4.5.: Classification of the existing literature based on placement granularity (adaptive/pre-defined).

Evaluation System		Reference
Hardware	DRAM-Intel Optane DC	[53, 54, 56–59, 62, 64, 65, 69, 71, 99–101]
	DRAM-HBM	[69, 70, 100]
Simulation DRAM/NVM		[55, 68]
Emulation DRAM/NVM		[60, 61, 63, 66, 67]

Table 4.6.: Classification of the literature based on target evaluation systems(Hardware, Simulation and Emulation).

RQ6: Where are the target placement approaches evaluated on?

Table 4.6 summarizes the target system, on which the placement algorithms have been evaluated on. The majority of the literature uses real hardware for evaluation, which is composed by either a DRAM and Intel Optane DCPM [53, 54, 56–59, 62, 64, 65, 69, 71, 99–101], or by a DRAM and an HBM [69, 70, 100]. Evaluating over real hardware takes into account all the real aspects that a system may inject, which cannot be fully accounted for on simulated [55,68] or emulated [60,61,63,66,67] systems. However, other approaches target simulated heterogeneous memory system, by utilizing the Persistent Memory Block Driver (PMDB) [196], or emulated heterogeneous memory systems, based on Quartz [197] and HME [198] tools. Through simulation and/or emulation new-coming technologies and architecture can be evaluated.

RQ7: What are the target optimization objectives?

Performance is the most crucial factor over heterogeneous systems, thus the majority of the placement works aims to provide performance optimization [53–66, 68–71, 99–101]. For performance, various metrics are considered, such as application's throughput, execution time, IPC and speedup. Apart from performance optimization, other objectives are also targeted, i.e. energy consumption minimization [67], memory bandwidth optimization [67], minimization of memory accesses [56,66] and cost [53].

RQ8: How can existing placement approaches be utilized and/or reproduced?

Table 4.7.: Classification of existing profiling techniques.

Usage	References
API	[53, 54, 57, 59, 62]
Transparent	[55, 56, 58, 60, 61, 63–71, 99–101]
Open Source	[100]

Throughout the existing literature, we examine whether the proposed placement approaches can be reproduced or not. We observe that existing works in their vast majority do not provide open source code, which can be reproduced and utilized for further research, except from one work [100]. Thus, the rest of the approaches need to be implemented from scratch. Table 4.7 summarizes the usage of data placement approaches. The target works can be split into two major categories: i) `API-based`, i.e., those that provide API to the developers/users and require custom annotations/modifications and ii) `Transparent`, i.e., those that placement is fully transparent to the user and can be utilized as a black box. We observe that transparent approaches [55, 56, 58, 60, 61, 63–71, 99–101] clearly prevail, compared to those that provide custom API-related solutions [53, 54, 57, 59, 62]. The transparent approach actually provides a more generic usage of the target placement methodology, while user involvement is minimized as it can be utilized and integrated as a black box. Thus it is widely selected as a design choice.

4.4. Design Space of Data Placement Approaches

In this section, we summarize the key design aspects of data placement that need to be considered during the design of the placement algorithm, as derived over the existing literature. The selection options are illustrated as decision trees, where the selection of each aspect can potentially affect the selection of the others (or a subset of the others). Figure 4.2 summarizes the most dominant design options.

I. Optimization Objective: The first critical decision regards the target optimization objective that the placement framework will aim to reach. The most dominant objectives are performance and energy, however, other options and multi-objective optimizations are also considered. Based on the definition of the optimization objective, the selection of the rest of key design choices is highly affected.

II-III. Placement & Profiling Policy: The placement approach should consider

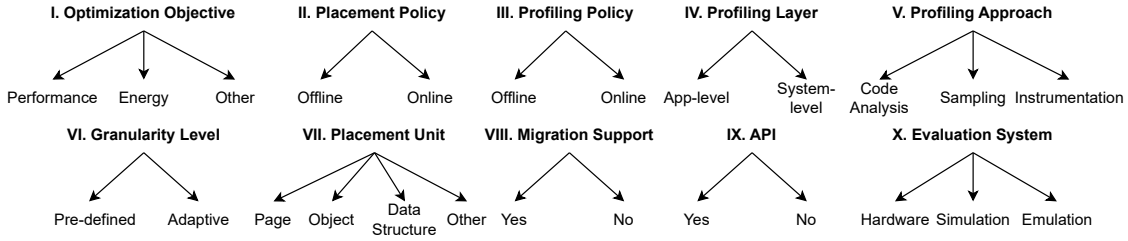


Figure 4.2.: Overview of the data placement design choices and guidelines as derived throughout the literature

whether the placement will be either offline or online, or it will be a composition of design-time and run-time decision-making. The placement policy strongly affects the decision of the profiling policy too. The application under deployment on a system with a heterogeneous memory system is traditionally analyzed using profiling tools, that can support offline placement or online placement or both.

IV. Profiling Layer: The type of profiling data collected depends on the level of data granularity of each specific methodology (e.g. placement of application-level objects or placement of pages). Examples of collected data include memory access pattern and cache miss ratio per object or page. The profiling data feed the placement algorithm, which may be based, for example, on analytical models used to tag application-level objects or pages in terms of the optimal memory type in which each one should be placed. Moreover, managing system-level objects/pages enables the build of methodologies entirely at OS-level, leading to approaches which are transparent to developers and do not require application source code modifications. On the other hand, approaches built at application-level often require high-level analysis and understanding of the executed code. Through source code annotations and managing data at finer level potentially provides improved results in terms of performance and energy.

V. Profiling Approach: Profiling information need to be collected for each application data object (i.e. page, application-level object or data structure, depending on the level of granularity of the specific approach) that will feed the data placement algorithm. Typical profiling information collected includes number and type of memory accesses, access pattern LLC misses and lifetime of each object. The profiling tools used to collect this information are either based on dynamic instrumentation, such as perf events [194], VTune Profiler [191] or on hardware performance counters [26] available in specific CPU architectures. Often, custom tools built on top of Pin or tools that monitor the performance counter are utilized. The profiling method significantly affects the degree by which an approach can be considered user-friendly. For example, the profiling of complex Database applications using dynamic instrumentation tools may be practically infeasible due to the

imposed overhead of monitoring huge workloads. However, assigning values monitored by hardware performance counters with application-level objects is not straightforward and requires a lot of effort [199]. Other approaches to application profiling include the development of custom profilers on top of end-to-end frameworks such as the Tensorflow [195] for identification of memory access patterns.

VI-VII. Granularity Level & Placement Unit: Furthermore, the granularity-level of placement is also a critical factor, as it can be performed by placing adaptive data or pre-defined data. The former may require more complex decision making, as it requires extra effort on efficiently selecting the correct placement unit dynamically. As placement unit, we can define any possible data that can be stored to the main memory, however throughout the literature, pages, objects and chunks, data structures and tensors have been met. Adapting data placement to access pattern changes during application execution can provide further improvements.

VIII. Migration Support: Although page migration can be supported relatively easy, migration of objects at finer granularity is more challenging, since it requires the implementation of a sophisticated runtime that monitors profiling information and triggers data movement with reasonable overhead [200]. Moreover, the movement of very short objects can trigger phenomena such as write amplification in NVMs [201,202], thus similar aspects should be considered during placement decision.

IX. API: Last but not least the usage of the designed placement algorithm should be decided. An API design that avoids significant application source code modifications is critical for complex scientific applications, while transparent approaches can be also considered. The integration of an API requires programming effort and efficient deployment by the developer, however it can potentially lead to fully optimal solutions.

X. Evaluation System: Finally, the target evaluation system, on which the placement will be deployed on, should be considered. Simulators [55, 68], emulators [60, 61, 63, 66, 67] and real hardware infrastructures are the evaluation system alternatives.

4.5. Guidelines and Future Directions

In this section, i) we indicate gaps in the existing literature on data placement over heterogeneous memory systems, regarding the target application domains and the optimization

objectives, ii) we provide future extensions and guidelines regarding emerging application domains and upcoming memory technologies.

4.5.1. Bridging gaps and future research

Application domains: Various domains have been examined for application specific data placement. However, there exist dominant domains that have not been examined yet. First, deep learning approaches consider only training phase, as shown in Section 4.2. However no work, up to now, that targets the data placement of deep neural networks on the inference phase has been examined. Furthermore, apart from deep neural networks, other ML domains should be examined. Recent advances in AI and Machine Learning, e.g. ChatGPT [203], tend to be more and more spread and widely utilized, thus leading to huge amount of data produced, which need to be stored efficiently. Regarding graph analytics, dynamic graphs [204] should also be considered.

Other domains such as Web applications, security and privacy applications [205], bioinformatics [206] and cloud analytics [207] are few examples that can be considered for application-specific placement. Furthermore, emerging application domains, such as Augmented Reality(AR)/Virtual Reality(VR) [208], blockchain-based applications [209] and crypto-mining [210] are expected to further rise within the next years, thus effective placement solutions should be considered. Last but not least, taking into account that heterogeneous memory systems target not only CPU systems, but also GPU systems, application domains such as computer vision and graphics require further exploration.

Optimization objectives: Existing literature mainly aims to provide performance optimizations, while few of them target energy, bandwidth, memory accesses and cost. Even though performance-oriented placement policies are widely examined, the rest of the objectives have not been extensively evaluated, thus it is not feasible to extract robust conclusions. Moreover, gaps exist regarding specific metrics that strongly affect the overall behavior of heterogeneous memory systems. For instance, in systems that utilize NVM technologies, endurance is a crucial factor, which can strongly affect performance and energy consumption. However, there is no work in the literature that aims to minimize hardware wear and protect the limited endurance of such technologies. Even though NVM accesses [56, 66] can be utilized as a high-level metric to estimate the impact on the endurance, however extra features are required in order to consider endurance as optimization objective.

4.5.2. How can placement policies adapt to upcoming memory technologies?

The revolution in upcoming memory technologies is expected to create a novel complex environment of heterogeneous storage. Technologies, such as CXL disaggregated memory (like Samsung's CXL Memory Expander [132]), upcoming DRAM solutions, including DDR5 [127], LPDDR5X [128] and GDDR7 DRAM [129] and custom solutions such as HBM-PIM [130] and AXDIMM [131] are just a subset of the expected technologies to be released within the next few years [129] and deployed on existing platforms, thus composing a multi-heterogeneous memory system. The memory management on such systems is expected to provide various trade-offs in terms of performance, energy, power and endurance. Existing placement algorithms can easily adapt and fit on systems that will follow the current trend of heterogeneous memory systems, i.e. a system consisting of a fast but energy inefficient memory and a slow but energy efficient memory. However, the integration of such kind of technologies is expected to setup create multi-tier memory systems of onboard or disaggregated memory [211]. Thus, for systems that will be composed of multiple memory technologies, multiple criteria should be considered and, therefore, new data placement approaches and major re-design of the existing approaches should be considered.

Chapter 5.

Application-Level Memory Management and Source-to-Source Transformations for Python Workloads

The advantages of processing data at the source motivate developers to offload computations at the edges of IoT networks. However, the computational resource constraints of edge computing devices limit the opportunities for deploying applications developed in high-level languages at the edge. Over the past decade, Python has experienced a meteoric rise to become one of the most popular programming languages globally. Renowned for its simplicity, readability, and versatility, Python has gained widespread adoption across various domains, from web development and data science to artificial intelligence and automation. Its extensive ecosystem of libraries and frameworks, coupled with a vibrant and supportive community, has contributed to Python's popularity, making it an ideal choice for both beginners and seasoned developers seeking efficiency and scalability in their projects. In contrast to C/C++, applications developed in Python and other dynamic high-level languages, often require an increased amount of memory size, due to their inherent static memory management approach. Therefore, developers targeting the deployment of applications in dynamic high-level languages at the edge need support by methodologies and tools, which will allow these applications to exploit the limited memory resources efficiently.

In this chapter, we present a memory optimization framework for applications developed in the Python programming language targeting edge devices. Aiming to avoid the inherent pitfalls of static memory management that Python's integrated memory manager imposes, the framework targets the reduction of the required memory footprint, by integrating a set of static and dynamic optimizations. The evaluation results, based on a set of representative real-life benchmark suites and applications, show 64% average memory footprint reduction, over the CPython's minimum baseline of 24MB. Additionally,

we investigate the impact of memory size reduction on execution time and energy consumption. The results show 51% lower execution time and 47.3% reduction in the energy consumed.

This chapter's text strongly relies on the following publication: Katsaragakis, Manolis, et al. "A memory footprint optimization framework for Python applications targeting edge devices." Journal of Systems Architecture 142 (2023): 102936. [212]

5.1. Introduction

Over the last years, there is an ever-growing number of interconnected IoT devices, providing new services and applications in a variety of domains, such as autonomous driving, agriculture, biomedical and healthcare. As the edge-to-cloud continuum paradigm evolves, there is an increasing need of processing and storage of data at the edges of the IoT networks, which are typically based on memory-constrained embedded devices. This approach has several advantages compared to the typical offloading of computations to the cloud. Consuming data directly at the source, contributes to the energy efficiency and low latency of the whole IoT system. On the contrary, the wireless transmission of data to the cloud is typically expensive in terms of latency and energy consumption, while it may impose security issues.

Providing high Quality of Service (QoS) based on real-time computations, while operating in resource-constrained edge devices is challenging. Developers are expected to address issues such as the processing power and memory size limitations of the underlying hardware while targeting real-time performance and low energy consumption [213]. These constraints, which are often contradictory, motivated the development of several design methodologies at various abstraction levels [214].

As the edge computing paradigm evolves, the popularity of high-level programming languages increases among IoT developers. For instance, Python relies on a wide variety of libraries and abstractions which encapsulate low-level optimizations and allow the expression of more functionality with less amount of code [102–104]. These features enable rapid application development, due to the limited programming effort required. Today, Python dominates in the areas of scientific computing and machine learning in both academia and industry [215], through libraries and frameworks such as the *scipy*, *numpy* and *Pytorch*.

However, porting computationally demanding applications developed in high-level lan-

guages to resource constrained edge devices is challenging. For instance, there are several studies indicating the high energy consumption and increased memory demands of Python compared to C/C++ [216–219]. Applications developed in Python need to be optimized to meet the constraints and limitations of the edge devices. In particular, Python relies on static memory management, which often leads to increased memory requirements compared to more flexible dynamic approaches, which are available in programming languages such as C and C++. Therefore, applications implemented using standard Python, often do not meet the constraints of typical embedded devices [143].

Table 5.1 shows the memory resources (amount of RAM) for some indicative state-of-the-art IoT devices. Applications are expected to adapt to the limited memory resources of such architectures. Even though the *microPython* implementation has been developed to target resource constrained embedded systems, the supported modules, and packages are very limited, while it is not compatible with CPython and other implementations [220]. Another feature of the CPython memory allocator, which is a result of its static approach, is that the allocated memory does not always get released back to the operating system, but instead it is offered back to the Python interpreter. The reason is that Python relies on a dedicated allocator for handling small objects that keeps memory reserved for future use. Therefore in long-running processes, we observe an incremental reservation of memory space, which may not be actually utilized, leading to increased memory fragmentation.

In order to deploy complex Python applications, such as machine learning algorithms and neural networks to the edges of IoT networks developers need methodologies and tools that will reduce the memory demands of these applications. Towards this direction, this work presents a novel framework that integrates a set of memory footprint optimization techniques for Python applications targeting IoT-embedded devices. The framework combines both static and dynamic optimizations applied at the application-level, which significantly reduce the memory requirements of Python applications. The novel contributions of this work are the following:

Table 5.1.: Memory specifications (RAM) of indicative state-of-the-art embedded boards

Device Name	Memory	Device Name	Memory
RT5350F-OLinuXino	32MB	Giant Board	128MB
Arduino Yún	64MB	Intel Galileo Gen 1 & 2	256MB
STM32 MCU		BeagleBone Black	
LC-CherryPi-PC-V3S			

- **The development of a systematic methodology for memory footprint optimizations for Python applications targeting resource-constrained embedded devices.** The methodology is based on a set of optimization techniques, which address the inherent pitfalls of static memory management on which the Python's default memory management relies. The methodology is supported by two novel tools that automate the process of redundant data removal and reduction of code hierarchy, which have a significant impact on the Python applications' memory requirements.

- **A thorough evaluation of the proposed memory optimization techniques,** demonstrated in a wide variety of representative real-world Python applications. Apart from demonstrating their effectiveness in terms of memory utilization, we also investigate their impact on their real-time performance.

The rest of the ch is organized as follows: An overview of the Python memory management approach and the motivation for the existing work is presented in Section 5.2. Section 5.3 describes the proposed methodology in detail. The evaluation results and a discussion highlighting the main observations are presented in Section 5.4. Finally, in Section 5.5 we conclude this work.

5.2. Background and motivation

Across the different Python implementations, various approaches in terms of the way that memory management is performed have been introduced, however, the core principles remain the same. In this work, we select the CPython version of Python, as it is the default implementation provided by Python developers [221], thus the most widely utilized version of the various implementations. We focus on the special internal characteristics and limitations of CPython's memory management.

Typically, in CPython, the code gets compiled to a low-level, platform-independent intermediate representation of the source code, namely the *bytecode*, which is finally loaded and executed by the Python interpreter engine. The Python memory manager relies on a garbage collector, which employs a reference counting technique for monitoring the active memory objects. With reference counting, the runtime keeps track of all of the references to an object. When an object has zero assigned references, it cannot be accessed by the program code anymore, so its allocated memory can be freed and returned to the system. Even though the automatic garbage collection is faster and hides implementation details, it imposes computation and memory overhead in order to track all application

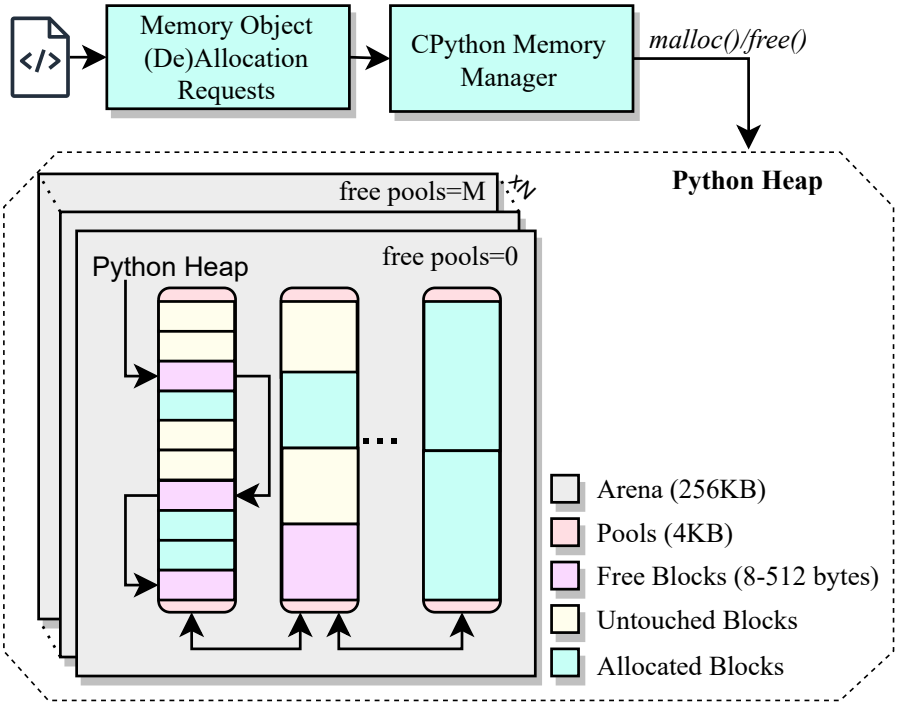


Figure 5.1.: CPython's memory management organization

references.

Furthermore, memory leaks occur in practice: the reference count of non-active objects may never become zero, even though these are not used in any subsequent computations. Typical causes of memory leaks are uncommon paths through the code structure. For instance, a function may allocate a block of memory and then free the block again, while the application algorithm does not use the specific block. Moreover, early termination of the application due to an error condition can lead to a memory leak. In general, if a Python application terminates early, all of its memory is released back to the operating system, however, there exist scenarios that lead to memory leaks. For instance, Python applications that utilize finalizers [222], which are special methods that are called when an object is about to be destroyed, may not be called if the program terminates abnormally. Moreover, the integration of third-party libraries into Python applications may not efficiently release the memory allocated, when the Python program terminates early.

Python's heap management is implicit to the application's developer and handled by the interpreter, i.e. the developer has no direct control over it. The dynamic memory

allocation/deallocation on Python's heap space is performed by the integrated memory manager through the Python/C API functions [223]. The Python memory manager is organized into three layers [224]:

- **Blocks:** Blocks refer to the fundamental memory management component, which is defined by a continuous number of bytes of virtual memory. Each block can be characterized as *untouched*, i.e. a portion of memory has not been allocated yet, *free*, i.e. an allocated block that has been "freed" and has no relevant data and *allocated*, which contains data required by the Python application. The blocks that are deallocated by the memory manager (i.e. they are "freed"), are added to a list data structure for future use by the manager (freelist). Typically, the size of the blocks varies from 8 to 512 bytes and it is a multiple of eight.

- **Pools:** Pools consist of blocks from a single-size class. Whenever a block is requested, the memory manager checks the pools for that block size. Each pool can be characterized as *free*, *used* or *empty*. Normally, the size of the pool is equal to the size of a memory page. Limiting the pool's size to the fixed size of blocks has a positive impact on memory fragmentation.

- **Arenas:** Finally, arenas are the highest level of memory components and contain pools of memory. Arenas are sorted by the CPython memory manager in ascending order, based on the availability of free pools in each arena, and, unlike pools and blocks, they do not have any explicit state. The typical size of the arenas is fixed at 256KB, which corresponds to the size of 64 pools, considering a 4KB page size.

Fig. 5.1 illustrates the organization of blocks, pools and arenas of CPython's memory

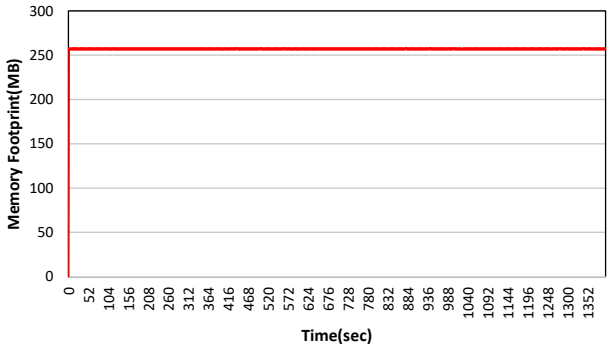


Figure 5.2.: Motivational example showing CPython's static memory management approach

manager [225]. The memory manager handles requests, by trying to provide memory from the arena with the lowest amount of available memory (i.e. the one which is mostly occupied). An interesting feature of CPython is the fact that the memory manager returns to the OS not pools or blocks, but arenas, only. In contrast to C/C++, when a memory block is deallocated, that memory is not actually returned back to the OS, but remains in a freelist. Although this approach typically reduces the memory allocation overhead in terms of performance, it is very static, leading to higher memory footprint utilization throughout the execution of an application. For IoT applications deployed on edge devices with memory constraints, such a static approach is often a significant limitation.

Figure 5.2 shows a typical memory utilization plot of a CPython application (a machine learning application derived from the IoT wearables domain [143]). Although the application is very dynamic, in the sense that it constantly allocates and deallocates memory blocks of various sizes, these fluctuations are not reflected in the OS. Even though mechanisms such as GC and Python's memory allocator are running in the background, this is not frequently observed from the operating system's perspective, thus leading to static memory utilization. For instance, the memory of objects freed by the GC are never becoming visible to the OS, as for a portion of memory to be given back to the OS, a whole arena should be freed. Therefore, from the OS's and system's perspective, the application is static it terms of memory requirements. Initially, we observe a spike for modules and data allocation, which is reserved for the rest of the execution, although many of the allocated parts are either tracked or removed by the garbage collector.

To address the aforementioned limitations, which are critical for CPython applications targeting edge computing devices, we propose a flow of application-level memory footprint optimizations integrated into a framework, which will enable a more dynamic memory management approach, to enable the efficient utilization of the limited memory resources of edge devices.

5.3. Memory Footprint Optimization Methodology

5.3.1. Overview

This section presents the systematic methodology for memory footprint optimizations for embedded Python applications and the tool-flow that supports the methodology. The input of the methodology is the source code of the application under optimization. An

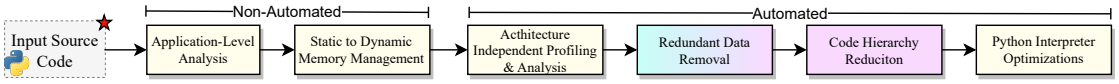


Figure 5.3.: Overview of the memory footprint optimization methodology

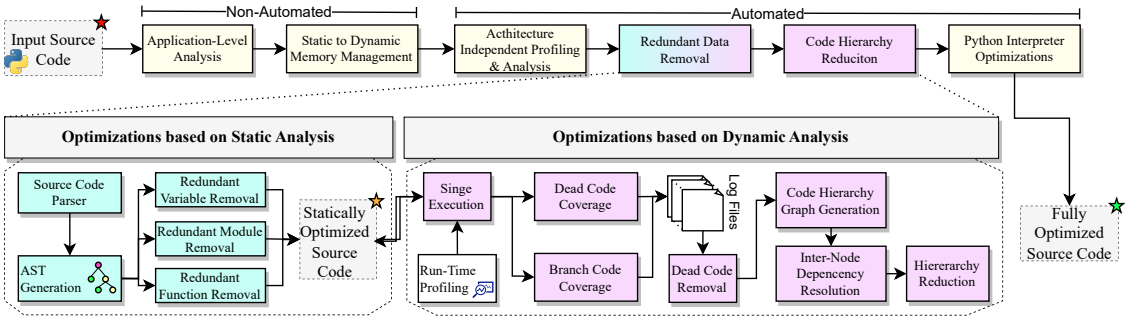


Figure 5.4.: Memory footprint optimization methodology

overview of the optimization techniques is presented in Fig. 5.3. The methodology consists of the following steps:

1. **Application-Level Analysis:** The first step of the proposed methodology includes the dynamic analysis of the application using well-established Python profiling tools, such as memory profiler [226], guppy [227], which enable the identification of the most critical application data structures in terms of memory requirements. This step identifies the source code functions and data structures, in which the applied refactoring and optimizations of the subsequent steps of the methodology are expected to have an increased impact.

2. **Static to Dynamic Memory Management:** In this step, the application source code is refactored, by converting the most critical data structures in terms of memory requirements, as identified in the previous step, from statically declared into dynamic. Apparently, this is a very application-specific step. Automating the refactoring is very challenging, because algorithmic adaptations may be needed in several cases, to avoid altering the functionality of the application. Therefore, its efficient implementation is left to developers, for now. However, the conversion from static to dynamic memory management is critical, since (i) it addresses the static memory management limitations in terms of memory utilization, as explained in the motivational example of Section 5.2 and (ii) it enables further memory optimizations. Expressing a significant amount of dynamic application behavior in terms of memory utilization normally enables several optimization opportunities, which are

Algorithm 1: Memory Footprint Optimizations Algorithm based on Static Analysis

```
   Data: Python Source Code (pythonApplicationFiles)
 1 Static-Analysis(pythonApplicationFiles):
 2     parsedCode = parseSourceCode(pythonApplicationFiles)
 3     AST = createAST(parsedCode) /* Create Abstract Syntax Tree(AST) */
 4     /* Traverse AST */
 5     for node in AST do
 6         if node is import then
 7             if node has no kid nodes then
 8                 /* Unused import detected */
 9                 remove(node,pythonApplicationFiles,
10                 optimizedCode)
11         else if node is variable object then
12             /* If variable is not referenced for load or assign */
13             if node.loadReferences == 0 && node.assignReferences == 0 then
14                 remove(node, pythonApplicationFiles,optimizedCode)
15         else if node is function definition then
16             /* Find AST nodes that reference to the current node */
17             node.num_of_references = 0
18             for newNode in AST do
19                 if newNode.reference == node && newNode !=node then
20                     node.num_of_references += 1
21             /* If there is no reference to this function from other objects, remove function */
22             if node.num_of_references == 0 then
23                 remove(node,pythonApplicationFiles,
24                 optimizedCode)
25     return optimizedCode
```

described in the next steps of the methodology.

3. **Architecture-Independent Profiling and Analysis:** After applying the necessary source code refactoring which creates a version of the application with a dynamic memory behavior, the application is profiled again. Object-trace profiling tools, such as the *Guppy-PE* [227], enable the collection of object and heap detailed memory sizing information. The profiling data generated in this step are fed as input into the subsequent memory optimization steps. Additionally, this step reveals the effectiveness of the refactoring applied during the *Static to Dynamic Memory Management* step.

4. **Redundant Data Removal:** The profiling analysis conducted by the previous step is used to identify parts of code, data, and imported modules that are useless for the actual functionality of the application, or that are duplicated versions of already existing parts. The former happens in cases of inefficient application development, while the latter is usually a result of the object-oriented nature of Python, in which the interpreter often creates instances of the same object in function calls. Keeping alive only the actual data and modules which are necessary for the application execution, enables significant gains in terms of memory footprint. The Redundant Data Removal step integrates techniques that rely both on static and dynamic analysis of the application under optimization. The implementation of this step in practice often requires extended source code refactoring. In this work, we fully automated this step. The corresponding tool which implements this step is described in detail in Section 5.3.2.

5. **Code Hierarchy Reduction:** An important factor that affects the overall memory footprint of Python applications is the memory overhead added by the interpreter for the static memory allocation of modules at the beginning of each program. Python's memory manager, by default, loads statically every single module used in the application, without taking into account which functions will be actually executed at run-time. Therefore, another way to optimize the total memory footprint of an application is to minimize the overhead added by these data. In practice, without modifying Python's memory manager, moving the functionality of the actual useful code from a module inside the application and reducing code hierarchy, reduces this overhead. This has to be applied selectively though, to keep the source code maintainability at a reasonable level while removing all relevant parts of the incurred memory footprint overhead. Again, this involves non-trivial reorganization of the code. Therefore, we fully automated this step and the corresponding tools we developed and integrated in the proposed framework are described in detail in Section 5.3.2 and Section 5.3.3.

6. **Python Integrated Optimizations:** Python's interpreter offers several flags and options to optimize the application execution in terms of required computing and memory resources. More specifically, we take advantage of *-O* flags to remove assert statements and discard docstrings. By implementing such optimizations in practice, an optimized Python bytecode is generated, i.e. a platform-independent low-level implementation of the target application to be executed on the Python's virtual environment. This step is fully automated, as it is provided directly by the Python interpreter.

The output of the methodology is the optimized Python application source code in terms

of memory footprint. Steps 2, 4 and 5 are expected to have the most significant impact on the memory footprint reduction. As stated earlier, step 2 is applied manually, while steps 4 and 5 are supported by novel tools which are integrated into the proposed framework. A detailed view of the methodology is shown in Fig. 5.4.

5.3.2. Memory Footprint Optimizations based on Static Analysis

Static memory footprint optimizations refer to a set of optimizations that are statically executed. Static analysis optimizations are integrated in Step 4 of the flow, as shown in Fig. 5.3. The main goal of the applied static optimizations is to detect and remove redundant imported modules and variables that do not contribute to any functionality throughout the execution of the Python application. Moreover, these optimizations act as a preprocessing step for the subsequent dynamic optimizations.

To implement the algorithm, we leverage a set of PyPi's projects, namely *pyflakes* [228] and *autoflake* [229]. *Pyflakes* analyzes programs statically, detects errors and operates by parsing source files. *Autoflake* utilizes *pyflakes* and it removes unused imports and variables from Python code. By default, *autoflake* removes only redundant imports for modules that are part of the standard library. This is due to the fact that other modules may have side effects that make them unsafe to be removed automatically. In our approach, we significantly extend the functionality of *pyflakes* and *autoflake*, so that we can detect redundant or duplicated imports and data beyond the standard library, as well as detect unused functions.

In practice, this is achieved by creating the Abstract Syntax Tree (AST) of the corresponding application. AST generates the abstract syntactic structure of the source code, where every single node denotes a construct. The source code is parsed once and the AST is generated. Next, all nodes of the generated AST are traversed. We distinguish three alternative types of AST nodes, namely (i) *Import Nodes*, which refer to AST nodes related to importing modules, (ii) *Variable Nodes* which correspond to nodes that refer to object initialization and, finally, (iii) *Function Nodes*, which are related to function definitions throughout the source code. For the former, we detect the number of nodes that depend on our target *import node*. If there exist no referenced nodes, the *import node* is redundant, i.e. it is not utilized by the rest of the target application. Therefore, the examined node and the corresponding source code can be safely removed. With regard to the *variable nodes*, each variable is represented as a Python object in AST. Variable references are utilized either to load the current, assign a new value, or delete the object. Initializing a new variable is characterized as a single load or an assign, depending on the kind of initialization. If no other references exist to the target object,

it is characterized as redundant and, thus, it can safely get removed from the source code.

The detection and removal of redundant *function nodes* is not straightforward. Existing tools for static analysis do not support any functionality for deleting unused functions. Therefore, we extended the functionality of the existing tools for *function node* removal. Initially, we detect all the *function nodes*, by traversing the AST. Next, we traverse again the AST and monitor all the references to object attributes or symbol names in the whole AST, which can be the function's potential callers. The *function nodes* that are not referenced by any other node in the source code, are marked as unused and, therefore, get removed from the source code. Self-references are excluded during this monitoring process, in order to be able to detect and remove recursive functions. Finally, Python offers dynamic code modifications and dynamic function definitions. Since the characterization of these cases as necessary or redundant is even more challenging, they are not considered during the static analysis process and their characterization is left as a future work.

To minimize the overhead of static analysis we provide in-memory node logging and tracing, thus avoiding extra disk storage for log files and time-consuming post-processing. Moreover, in addition to memory footprint reduction, removing redundant modules, data and functions leads to improved source code quality, as it increases code maintainability. Algorithm 1 demonstrates the core functionality of memory footprint optimizations based on static analysis.

5.3.3. Memory Footprint Optimizations based on Dynamic Analysis

This section describes in detail the optimizations enabled by the dynamic analysis, i.e. by profiling and executing the corresponding Python application. The optimized source code generated after applying the static analysis optimizations to remove redundant modules, variables, and functions (Section 5.3.2) is forwarded to the dynamic analysis for further optimizations. In particular, the dynamic optimizations include Redundant Data Removal enabled by dynamic analysis (Step 4 in Fig. 5.3) and the reduction of the code hierarchy (Step 5 in Fig. 5.3). The pseudocode of Algorithm 2 illustrates the core functionality of both optimizations, which are analyzed in the rest of this section.

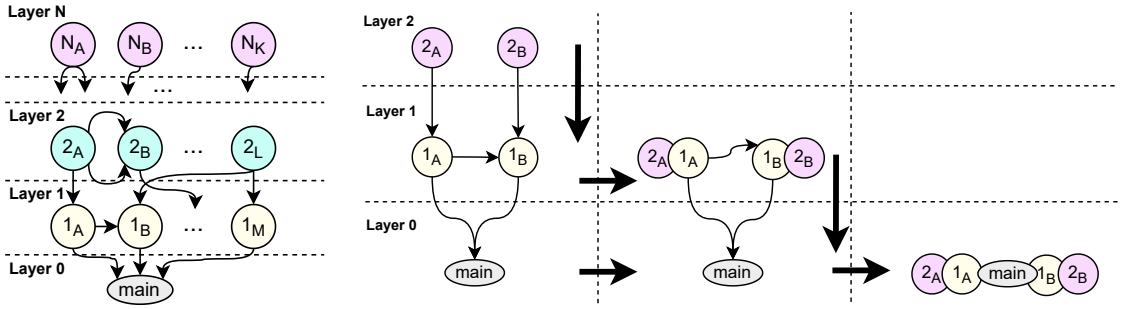
Redundant Code Removal In contrast to static programming languages, such as C/C++ where memory allocation and deallocation are directly done at run-time on the corre-

sponding code line, the Python's memory manager does not dynamically (de)allocate the required memory. Explicit memory management usually leads to higher memory requirements, as a significant amount of the estimated footprint is pre-allocated statically by the memory manager. Thus, the existence of redundant code and data can cause high static memory pre-allocation, which cannot be directly deallocated by developers.

To further identify parts of the source code which can be safely considered redundant, we rely on dynamic analysis results. Throughout the execution of the application under optimization, we monitor the program execution on a different thread and trace the sequence of the source code instructions executed. We generate annotated traversed source code listings, caller/callee relationships, and list the functions executed. Moreover, by taking advantage of *coverage.py* project [230], which is a third-party coverage tool, we extract the code coverage, i.e. the percentage of source code actually executed for a particular run and the corresponding source code lines. Existing tools, such as [230, 231] do not provide code removal features. Instead, they only report the coverage, as well as a percentage indicating the corresponding reliability. Therefore, we extend their functionality, to characterize alternative parts of code and provide automated code removal features. Through our implementation, we derive a detailed code coverage report and distinguish the following two cases:

- **Dead code coverage:** These parts of source code refer to all the non-executed code apart from the ones mapped to branch code coverage. As no condition can allow the execution of these parts of source code, it is automatically removed by deleting the corresponding source code lines. Interestingly, Python allows for classes to be dynamically modified at runtime, and evaluate arbitrary Python expressions from a string-based or compiled-based-code input by using the *eval()* function. Although Python's *eval()* is a very useful function, the code generated at runtime cannot be known a priori, thus these parts of code should not be removed. However, if the application input is fully deterministic, then it may be possible to characterize the corresponding parts of the source code as redundant, too.

- **Branch code coverage:** It refers to the non-executed parts of code placed under branches. As a branch we consider every line of code that triggers switching execution to a different instruction sequence as a result of executing a branch instruction. Branch code coverage depends on control flow structure and triggering inputs, with more deeply nested statements being significantly less likely to be covered [232]. The most representative cases are *if* branches, *for*, and *while* loops. Such parts of the code are not considered as redundant, thus they are not automatically removed. Similarly to the dynamically modified dead code detected in *Dead code coverage*,



(a) Inter, intra-dependencies **(b)** Example of inter and intra-dependency resolution

Figure 5.5.: Visual representation of the code hierarchy, inter and intra-dependencies and example of dependency resolution.

such parts of code can be removed for fully deterministic application inputs, only.

After the identification and removal of redundant code through dynamic analysis, the optimized source code is provided as input to the *Code Hierarchy Reduction* step.

Code Hierarchy Reduction An assumption on which the specific optimization relies on, is that the source code of the imported modules exists locally; either in the application's file directories or on Python's standard libraries on the device, and it is not stored as bytecode. In order to apply this optimization in practice, we generate the code hierarchy graph of all the modules for all the application layers, during the dynamic analysis. Fig. 5.5a illustrates a generic graph representation of a multi-layer code hierarchy. Each node corresponds to a unique module, while each edge represents a dependency among modules. For example, in Fig. 5.5a module $1A$ depends on module $2A$. Additionally, the same node can exist in alternative layers of the hierarchy, while there is no limitation on the number of dependencies that each node can have, i.e. multiple nodes can depend on multiple nodes. Finally, directly or indirectly, all imported modules refer to the main module of the application under optimization. Moreover, Python allows circular dependencies between modules, where each module is defined in terms of the other (i.e. module A requires module B and simultaneously module B requires module A). In fact, in the majority of cases, circular dependencies occur due to poor application development and they are often a cause of memory leaks, infinite *import* recursions and may have negative impact on the source code maintainability. To address these issues, we developed a tool which merges the corresponding modules in one, while respecting the sequence of inter-module function calls and data types access.

In order to retain the application functionality, the tool respects the dependencies between

the modules. We implemented a top-down hierarchy reduction design, starting from Layer N, where there exist no dependencies from upper layers. We identified two distinct cases for module dependencies, namely intra-layer and inter-layer dependencies. The former refers to dependencies within the same layer, while the latter refers to dependencies among alternative module layers. Initially, we resolve the intra-layer dependencies of *Layer* N, i.e. the corresponding source codes of the independent layers are transferred to the dependent layers. Next, we proceed with the inter-layer dependencies between *Layers* N and $N-1$, in which, similarly, the source codes of *Layer* N is transferred to the corresponding dependent nodes of *Layer* $N-1$. The *Layer* $N-1$ becomes the top-level layer of the code hierarchy. The aforementioned process is repeated, until the procedure reaches *Layer* 0, where the *main* module is located, which is responsible for the application's execution.

Fig. 5.5b illustrates a simple example of *3-Layer* hierarchy application. Starting from *Layer* 2, as there exist no intra-layer dependencies, the functionality of nodes $2A$ and $2B$, is transferred to modules $1A$ and $1B$, respectively (Step 1). Next, the process is continued on *Layer* 1, where there exists an intra-layer dependency among the combined nodes $1A2A$ and $1B2B$. Thus the corresponding source code is transferred to the dependent node, i.e. from $1A2A$ to $1B2B$ (Step 2). Finally, as no intra-layer dependencies exist, the new node is relocated to the *main* node in *Layer* 0 (Step 3.) and the code hierarchy is fully reduced, thus the interpretation overhead due to module allocation is reduced to minimum. The core functionality of the code hierarchy reduction tool is summarized in lines 17-30 of Algorithm 2.

Even though code hierarchy reduction is expected to reduce the memory footprint, trade-offs can be identified between the source code maintainability and memory footprint. The process of hierarchy reduction is executed recursively over the alternative modules and libraries of the target application and is finalized when the memory footprint is not reduced more than a user-defined percentage $\lambda\%$, which is configurable by the user.

Algorithm 2: Memory Footprint Optimizations Algorithm based on Dynamic Analysis

```
   Data: Python Source Code Files(pythonApplicationFiles)
 1 Dynamic-Analysis(pythonApplicationFiles):
 2     /* Single Execution */
 3     while executeApplication() do
 4         traceFile = traceApplication() /* Trace application throughout the execution */
 5         hierarchyLayers = generateHierarchyTree(pythonApplicationFiles) /* Generate code
             hierarchy tree */
 6     coverage = monitorCodeCoverage(traceFile, pythonApplicationFiles) /* Code Coverage */
 7     for code in coverage do
 8         if code is in non-executed branch then
 9             continue /* Avoid Branch Code Coverage */
10         else
11             deleteCode(pythonApplicationFiles, optimizedCode) /* Remove Dead Code */
12     /* Hierarchy Reduction */
13     for layer in hierarchyLayers do
14         for node in layer do
15             /* First, check for intra-layer dependencies */
16             if layer has intra-dependencies then
17                 for dependentNode in layer do
18                     if dependentNode depends on node then
19                         transferCode(node, dependentNode, pythonApplicationFiles,
                             optimizedCode) /* Resolve intra-layer dependencies */
20             for dependentNode in layer-1 do
21                 /* Check for inter-layer dependencies with layer-1 */
22                 if  dependentNode depends on node then
23                     transferCode(node, dependentNode, pythonApplicationFiles,
                         optimizedCode) /* Resolve inter-layer dependencies */
24     return optimizedCode
```

5.4. Experimental Results and Evaluation

5.4.1. Experimental Setup

The experiments were conducted using the Anaconda version of Python 3.7 language based on CPython implementation. We utilize a Raspberry Pi4 Model B, equipped with Quad core Cortex-A72 (ARM v8)@1.8GHz and 4GB of DRAM as a state-of-the-art embedded board. Raspberry Pi4 is one of the most widely used embedded boards

over a wide variety of application domains [233]. Moreover, the memory management of Python applications, is directly handled by the Python's engine, thus the memory gains are expected to remain the same and are representative for a wide range of devices, including IoT devices, with even lower memory capacity.

Concerning the profiling and analysis (step 3 in Fig. 5.4), we rely on well-established tools, such as Python's *memory_profiler* [226], which is a pure Python module for monitoring the memory consumption of processes, as well as line-by-line analysis of memory consumption for Python programs. For the memory monitoring of a process from the operating system's perspective, we use the *Linux perf* [194], a profiling tool that provides an abstraction of the underlying hardware and provides low-level system metrics.

The proposed memory footprint optimization framework is extensively evaluated based on the following applications and benchmarks:

- A set of benchmarks from the *pyperformance* [234] suite. The specific benchmark suite collects real-world Python applications relevant to all Python implementations. The input of each selected *pyperformance* benchmark is the default (i.e. the one that the suite provides).

- A real-world Convolutional Neural Network(CNN) application, consisting of multiple convolution and activation layers for digit recognition, derived by [235]. As input, we provide the MNIST dataset [236], which consists of images of digits ranging from 0 to 9, represented as a matrix of shape 28x28x1.

- A real-world machine learning-enabled *IoT biomedical application* which is executed on a wearable device. In particular, the application provides activity prediction of patients, given as input sensor coordinates (x, y, z) placed on the patient's chest and a set of features per patient. The possible classification states are: lying (1), sedentary (2), dynamic (3), walking (4), running (5), and biking (6). The application uses a pre-trained Python auto-regressive model to provide predictions and the classifier used is the Random Forest (RF). RF [237] is an ensemble machine learning method that is often used for classification. Multiple decision trees are constructed, each of which predicts according to the given input. The final prediction is based on majority voting classification of the individual trees. The RF classifier is implemented using the scikit-learn software library. We used a data-set consisting of 197,633 coordinates, which are split into data blocks of 128 coordinates each one, thus generating 1544 patients data points and 15 features per patient. The data blocks are independent (i.e. there are no data dependencies between them). Therefore, the output of the application is 1544 activity predictions, which are produced

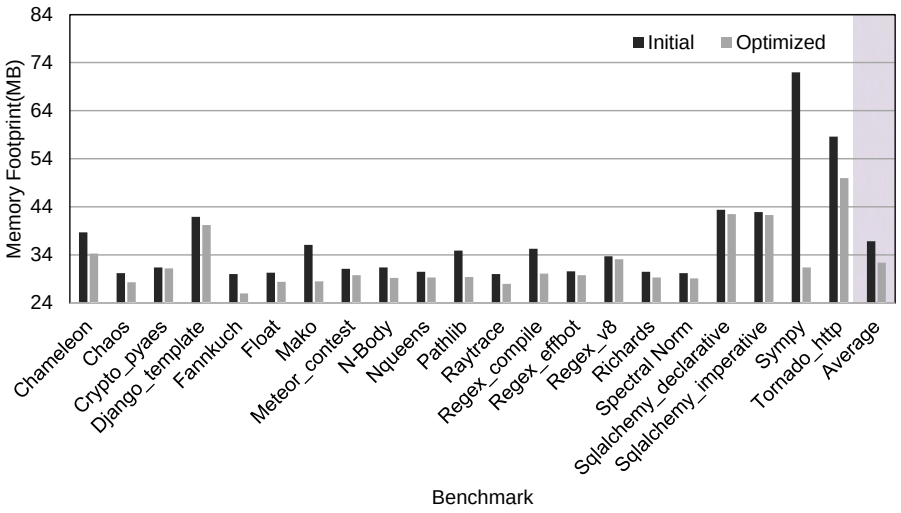


Figure 5.6.: Memory footprint optimization for Pyperformance Benchmark Suite

sequentially. We used 20 estimators (i.e. trees), without any limitations in terms of tree depth. All these parameter settings were derived from the real-life application provided by IMEC [238].

Experiments showed that the minimum possible memory footprint required for an empty Python application is 24MB. This size corresponds to a set of 96 arenas of 256KB, which are allocated at the initialization of every application, in order for the interpreter to operate efficiently, and are never deallocated until the end of the execution. Therefore, this threshold is used as a baseline for our experiments.

5.4.2. Pyperformance suite & CNN Application

Memory footprint optimization evaluation: Figure 5.6 illustrates the memory footprint optimization results before and after applying the proposed framework on each application of the *pyperformance* benchmark suite. X axis denotes each benchmark, while Y axis denotes the maximum memory footprint of the application in MB. The results show that the average memory footprint reduction is 34.6%. The benchmark with the highest memory footprint reduction is *Sympy*. *SymPy* [239] is a Python library for symbolic mathematics. Detailed analysis of the specific benchmark shows that the *Sympy* library allocates a large amount of redundant data, while in practice it utilizes only a small amount of them. Furthermore, *Sympy* is known to behave insufficiently when the user requests the handling of very long expressions [239]. The reduction of

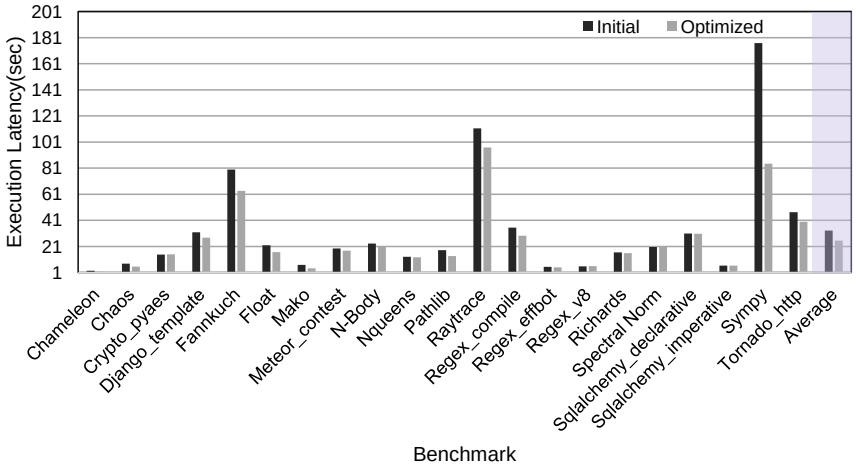


Figure 5.7.: Execution time optimization for Pyperformance Benchmark Suite

code hierarchy and the removal of useless data/code resolves the aforementioned issues, reaching a memory footprint optimization of 84.5%. Furthermore, from Figure 5.6 we observe that the benchmarks *Django_template*, *Tornado_http*, *Sqlalchemy_declarative* and *Sqlalchemy_imperative* show the smallest optimization in terms of memory footprint. In order to operate efficiently, all four of these require a large set of input data that need to be processed at runtime. Although such a dataset was not available in the *pyperformance* suite during the evaluation of the framework, we can safely assume that a larger input dataset will lead to increase memory footprint reduction.

Impact on execution time: Figure 5.7 shows the execution time of each *pyperformance* benchmark before and after applying the memory optimization framework. The Y axis represents the execution time of the corresponding benchmark in seconds. Interestingly, the memory footprint optimizations result in execution time reduction, by 33.1% on average. We assume that the main reason is that removing redundant modules and reducing the code hierarchy simplifies the application data flow, having a significant positive impact on the execution time.

CNN application evaluation: Moreover, our experiments are also conducted on the CNN application over the MNIST dataset for embedded systems. Figure 5.8 illustrates the memory footprint and execution time optimizations, showing an improvement of 27MB, which corresponds to 49.1% decrease, having 24MB as baseline, as illustrated in Figure 5.8a. Regarding the execution latency experiments, Figure 5.8b presents the execution latency of the inference throughout the execution and we observe a speedup of 1.3x.

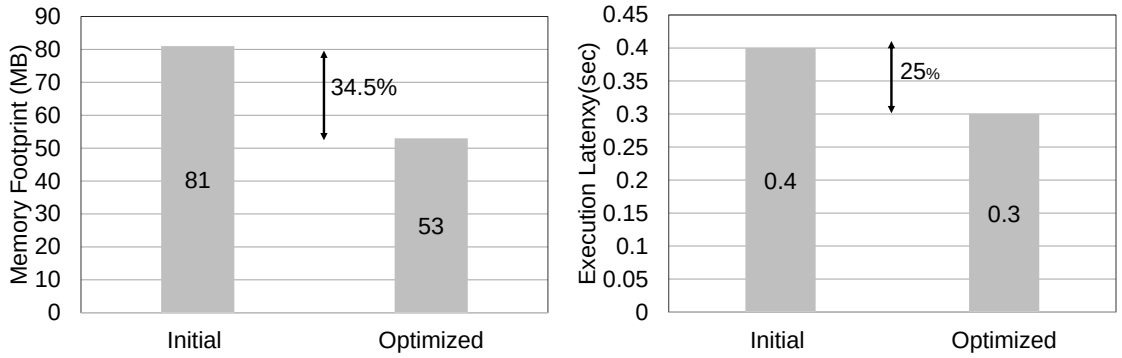
Table 5.2.: Python arenas and pools optimization for Pyperformance Benchmark Suite

Benchmark	Initial Avg. # of Arenas	Optimized Avg. # of Arenas	Initial Avg. # of Pools	Optimized Avg. # of Pools
Chameleon	155	137	623	512
Chaos	120	113	492	441
Crypto_pyaes	127	125	516	489
Django_template	168	161	673	636
Fannkuch	120	104	487	403
Float	121	113	488	421
Mako	144	114	587	459
Meteor_contest	124	119	503	477
N-body	126	116	511	467
Nqueens	122	117	495	469
Pathlib	140	118	563	470
Raytrace	120	112	485	448
Regex_compile	141	120	571	469
Regex_effbot	122	119	493	477
Regex_v8	135	132	544	523
Richards	122	117	491	469
Spectral Norm	121	116	485	449
Sqlalchemy_declarative	174	170	694	669
Sqlalchemy_imperative	172	169	686	677
Sympy	288	125	1139	465
Tornado_http	234	200	936	772
Average	**147.3**	**129.5 (-9.6%)**	**589.1**	**518.2 (-12.07%)**

Heap memory management utilization analysis: For a more thorough analysis, we monitored the impact of the optimizations on CPython's internal heap memory management. Table 5.2 illustrates the average number of allocated arenas and pools throughout the execution of each benchmark before and after the implementation of the framework. The experiments were conducted with a 256KB arena size. The pool size is set to 4KB, in order to be equal to the operating system's page size, aiming to reduce fragmentation; thus saving memory space. By applying the proposed framework, the average number of active arenas is reduced by 9.6% on average. Similarly, the average number of active memory pools is reduced by 12.07% on average. The application-level memory footprint optimizations lead to optimized results in terms of pool and arenas utilization. Also, by exploiting the dynamic memory aspects of an application, more pools are reduced throughout the execution, in contrast to static memory management, where pools cannot be frequently deallocated. Moreover, though the *Useless Data Removal* step, small-sized objects and variables are removed, thus leading to fewer allocations of small-sized memory blocks.

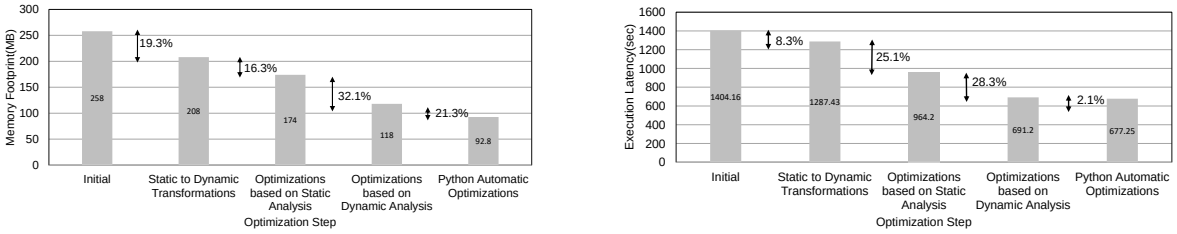
5.4.3. IoT biomedical application and use-case demonstration

The memory optimization methodology is evaluated on a real-world biomedical application, which is based on an RF classifier. We present a use-case demonstration of each framework's step, aiming to provide a simple example of how our approach is applied in practice and the impact of each step, respectively. Figure 5.9 shows detailed memory opti-



(a) Memory footprint **(b)** Execution Latency

Figure 5.8.: Experimental evaluation of memory footprint and execution latency optimizations on Convolutional Neural Network(NN) application over MNIST dataset.



(a) Memory footprint optimization **(b)** Execution time optimization

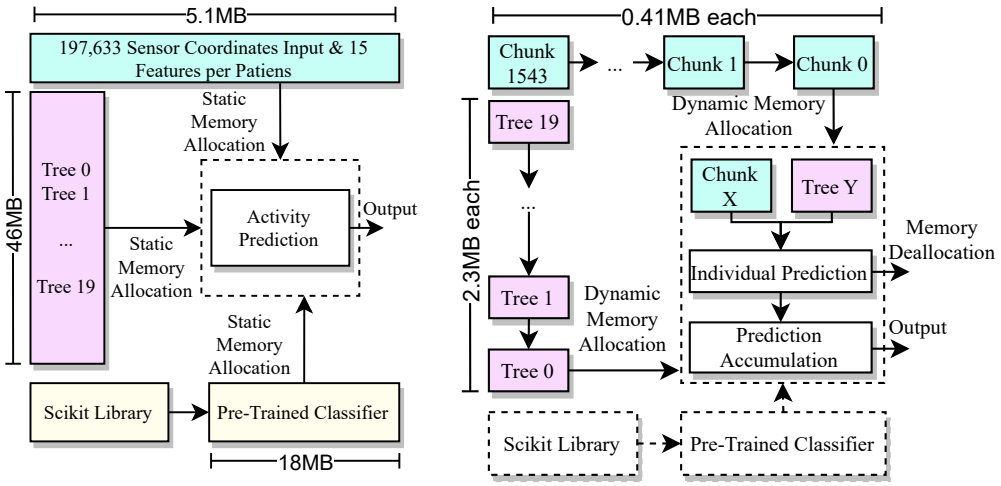
Figure 5.9.: Experimental evaluation of memory footprint optimizations on IoT biomedical application

mization results (Figure 5.9a, as well as their corresponding impact on the execution time (Figure 5.9b). This detailed presentation of results allows the evaluation of the contribution of each step to the overall memory footprint and execution time reduction. The initial application memory footprint is measured at 258 MB, all statically allocated at the beginning of the execution. The execution time of the original version of the application for the specific input dataset is 1404.1s. A detailed analysis of the overall memory footprint requirements showed that the majority of allocated memory is used by imported modules. More specifically, 185MB were assigned to the imported libraries, while 69.1MB were allocated to store the input data and prediction models.

Demonstration and impact of static to dynamic memory management After the application-level analysis (step 1), we implemented Step 2 of the methodology, to convert the most critical application data structures from static to dynamic and enable dynamic

memory allocation and deallocation. Figures 5.10a, 5.10b illustrate the comparison of the two approaches using a high-level application data-flow. As explained in Section 5.3, this step is manually implemented, as it requires an understanding of the application algorithm to avoid altering the application functionality. Originally, the prediction phase of the random forest algorithm was implemented statically, as follows: As soon as the data point is ready, it is fed to all the trees of the forest and then the output of each tree is gathered to perform the prediction, as shown in Figure 5.10a. However, if the prediction is executed sequentially, then only the information of a single tree is required to be allocated in the memory at each point of the prediction phase. Thus, possible performance can be traded for storage requirements, respecting the functionality and the accuracy of the algorithm. The RF classifier contains 20 prediction trees of 2.3MB each. Moreover, 18MB is used to store classifier metadata. Furthermore, in practice, not all classifier's information is required simultaneously. The required functionality and the necessary data can be provided to the algorithm dynamically when needed, only [143]. In other words, the dynamic loading of trees enables the dynamic allocation of memory required for the input data to each tree. More specifically, in the initial implementation, the test set and features memory size is 5.1MB. However, the transformation allowed the test to be split into 128 chunks of 0.41MB each, further reducing the RAM size requirements. Listing 5.1 shows a code snippet of how the source code is structured before(top) and after our transformation is applied(bottom). The modified version of the application is illustrated in Figure 5.10b. This static-to-dynamic conversion reduced the overall memory footprint by 19.3% compared to the initial memory size, which corresponds to 50 MB, while enabling and increasing the impact of the optimizations of step 4 and step 5. Additionally, the execution time is reduced by 8.3%.

Demonstration and impact of static and dynamic analysis The redundant data removal and code hierarchy reduction steps had also a significant impact on the memory size optimization of the application. Even though we propose a unified methodology, the automated proposed approach can operate independently to the non-automated steps. As we observe in Figure 5.9a, the optimizations based on static analysis reduced the memory requirements and the execution time by 34MB and 323.2s, respectively. The optimizations based on dynamic analysis reduced the required memory size by 56MB and the execution time by 273s. Aiming to provide further insights on the impact of the static and dynamic-based optimizations, we investigate how our application is transformed and what is the impact of each step, respectively. Listing 5.2 depicts a code snippet of the redundant data removal. In this example, the function `estimateECGFearutres()` is never called through the source code, thus represented in the AST as leaf and is removed statically. On the other side, the `if` statement is input-dependent, thus cannot be removed statically, therefore is removed on dynamic analysis. Last but not least, the part of the



(a) Static Memory Management **(b)** Dynamic Memory Management

Figure 5.10.: Static (left) vs dynamic (right) memory management of the IoT biomedical application, before and after applying the optimization methodology.

code after the `return` function, which is outside the branch, can be removed only through dynamic analysis. This leads to fewer instructions invoked by the interpreter, while it resolves data dependencies for later steps.

As far as the code hierarchy reduction is concerned, this is implemented automatically through our framework. The application's version fed as input in this step is executed by importing 17 Python modules. Through redundant module removal, 23MB of memory is freed. Only the functionality of the utilized functions is moved to the main module, thus maintaining active only the necessary data. This leads to both less memory footprint and gains in execution time. Listing 5.3 illustrates a representative example of code hierarchy reduction. For instance, the `estimateAR()` function is the only one utilized from the AR module, thus its functionality is transferred to the main module, while the rest of the AR is fully removed.

Furthermore, by exploiting the automated optimizations that Python's interpreter offers (step 6), the overall memory footprint reached 92.8MB. Compared to the original version of the application, the memory footprint reduction reached 64% (Figure 5.9a), while the execution time was reduced by 51.8%, as shown in Figure 5.9b. The execution time reduction is highly correlated to the memory footprint, due to the following reasons: i) the number of lines of code is significantly reduced, thus the interpreter's overhead and execution overhead is reduced, ii) the number of accesses to the main memory is reduced, thus CPU stalls are also reduced, iii) non-required memory alloca-

tions/deallocations and imports are avoided and iv) data and can fit within the cache, leading to more cache hits and faster execution times. Last but not least, Figure 5.11 illustrates the comparison of the initial and final memory utilization of IoT biomedical application. We observe that through our approach, the dynamic nature of the examined application can be exploited, thus leading to more effective utilization of memory. For low-resource devices, this dynamic behavior is critical for the efficient execution of an application.

5.4.4. Discussion

As shown in the above experiments, the proposed framework enables significant memory footprint optimizations, which also have a positive impact on execution time. For example, as can be noticed by Table 5.1 and Fig. 5.9a, the initial version IoT biomedical application could not be deployed on a Giant Board, on an Intel Galileo, or on a BeagleBone Black. However, after applying the proposed framework, the optimized version requires significantly less amount of resources and can be successfully executed.

To further investigate the impact of the proposed memory optimizations on other metrics, we monitored the energy consumption of the *pyperformance* benchmark and the IoT biomedical application, before and after applying the optimizations. The applications were deployed on an Nvidia Tegra X1 embedded device with 4 ARM Cortex-A57 processors running at 1.9 GHz and 4 GB RAM. The energy was measured by integrated power sensors plugged into the device. Figure 5.12 illustrates the energy gain percentage of the optimized *pyperformance* benchmarks, compared to the initial implementation. We observe that the energy consumption follows the behaviour of memory and performance

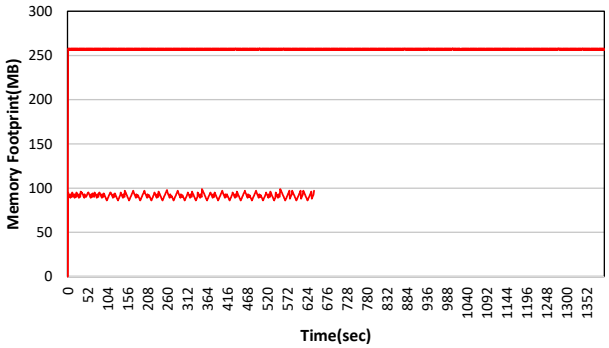


Figure 5.11.: Comparison of initial and optimized memory footprint utilization

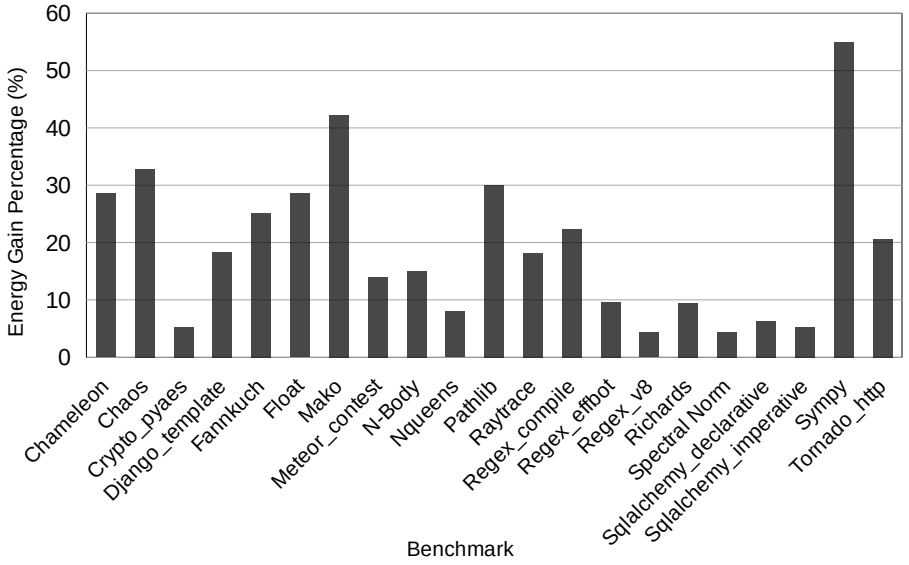


Figure 5.12.: Energy consumption reduction percentage(%) for Pyperformance Benchmark Suite.

optimizations [160], leading up to 55.1% less energy consumption and 19.3% on average. Similar experiments were conducted for the IoT biomedical application, showing 47.3% lower energy consumption.

Furthermore, an interesting question is to which extent the optimized Python implementation can compare to the corresponding C implementation in terms of execution time and memory requirements. Therefore, the optimized version of the IoT application (i.e. the optimized Python source code after applying the proposed framework) was developed from scratch in C language. Both the Python and the C implementation of the IoT application were executed on an Nvidia Tegra X1. Table 5.3 summarizes the results of the comparison between Python and C implementation, using the same application input.

Regarding the execution time, the C implementation is 1.5x times faster, compared to the Python implementation. This is due to the fact that C is a compiler-based language, thus no code translation is required at run-time, and code optimizations can be applied by the compiler and further optimize the program using static information about types and memory layout of objects. In contrast to Python, which is an interpreter-based language, C avoids the run-time overhead of the interpreter and optimizations prior to execution cannot be achieved. This is a commonly observed behavior, as the identified overheads account for 64.9% of the overall execution time, while the remaining 35.1% is used for

Table 5.3.: Python and C code comparison

	Python Implementation	**C Implementation**
Memory Footprint	*92.8 MB*	*2.9 MB*
Total Memory Accesses	$31.4x10^6$	$74.9x10^5$
Execution Time	*677.25 sec*	*451.3 sec*
Lines of Code (LoC)	*1x*	*3.4x*

the execution of the program [149]. Therefore, there is at least 2.8x increase in execution time on average moving from a C-like program to a Python program running on CPython due to language and interpreter overheads. This observation is inline with corresponding results reported in the existing literature [151].

Additionally, we measured the number of total memory accesses of the corresponding applications by utilizing the *Linux perf* tool. More specifically, we monitored the performance counters of the processor throughout the execution of C and Python execution, respectively. We observe that Python requires an order of magnitude more memory accesses compared to the C implementation (23.8% more accesses). The underlying layers that CPython contains, induce a significant overhead for read and write accesses of the executed application.

Although the memory requirements of the Python implementation are higher and performance is lower than in C, it is friendlier to application developers. Indeed, as shown in Table 5.3 the C implementation of the IoT application requires 3.4x less LoC than the corresponding Python implementation. The fact that Python provides a variety of libraries and software abstractions makes it suitable for fast code development and contributes to its increased popularity in the embedded and HPC communities. The abstractions offered by Python can significantly contribute to highly maintainable source code, which is a critical feature for applications under development and for commercial software products. Even though it is probably not possible to achieve the same memory and execution time with corresponding C implementations, it is still important to reduce Python overheads as much as possible in an automated way, to allow developers to exploit Python's user-friendliness features in edge devices of limited resources.

5.5. Conclusion

In this work, we described a methodology supported by a tool flow for the memory optimization of Python applications targeting devices with limited computational resources at the edges of IoT networks. The methodology is enabled by dynamic memory management techniques and relies on two novel tools which automate the removal of redundant data and the reduction of the code hierarchy of Python applications. Although the optimizations focus on memory footprint reduction, we show that they have a significant positive impact on other metrics, including application execution time and energy consumption, which are also critical in the area of edge computing development.

```
# Static data load and processing
acc_data = pd.read_csv(acc_file) # Static load of data
classifier = open("all_trees.pckl") # Static load of RF trees
num_blocks = len(data['ACC'])/(4*32)
for i in range(num_blocks-1):   # Feed blocks to trees
    ... # Processing
-----------------------------------------------------------------
# Dynamic data load and processing
for gm_chunk in pd.read_csv(acc_file, chunksize=128, iterator=True): # Load
    data on chunks of 128 elements
    for i in range(0,20):
            # Load tree one by one
            pickle_in = open("tree"+str(i)+".pckl", "rb")
            ... # Processing
```

Listing 5.1: Code comparison before (top) and after (bottom) applying static to dynamic memory transformations step on IoT biomedical application

```
def estimateACCFeatures():
    ...
# Removed at static analysis as it is never called
def estimateECGFeatures():
    ...

class AR:
    def __init__(self, model_path, sensor_position, clf_method):
        ...
        # Removed at dynamic analysis as its execution is input dependent
        if clf_method is None:
            clf_method = 'rf'
        ...
        if sensor_position == 'chest':
            return
         # Rest of code removed at dynamic analysis as its execution depends on
             'return' function trigger
        ...
```

Listing 5.2: Redundant code removal code snippet

```
# Initial code hierarchy
... # Other imports
from AR import * # AR is initially imported
...
# Only estimateAR() is utilized from AR
current_ar = estimateAR(acc_data)
...
-----------------------------------------------------------------
# Reduced hierarchy
... # Other imports
# Only estimateAR() functionality is moved to main module
def estimateAR(self, df):
    ...
current_ar = estimateAR(acc_data)
```

Listing 5.3: Code comparison before (top) and after (bottom) applying code hierarchy reduction

Chapter 6.

Data Structure Exploration & Placement over Heterogeneous DRAM/NVM Memory Systems

Over the last few years, the rapid growth of heterogeneous memory systems is composing a complex computing paradigm, where applications need to be deployed and their data need to be effectively accessed and stored across the memory hierarchy. The effective application deployments on heterogeneous memory systems, requires effective co-design in terms of the dynamic data type implementation for distinct workloads, the placement of the target data and the hardware configuration. In this chapter, we provide an in-depth solution for dynamic data type exploration, placement and heterogeneous memory system configuration fos applications deployed on heterogeneous DRAM/NVM memory systems. More specifically, the core contributions of this chapter are the following: i) we present an energy consumption evaluation of Intel Optane DC Persistent Memory for Indexing Data Structures. ii) We propose a novel methodology for persistent B+-tree indexes exploration for co-optimization of performance and energy consumption. We evaluate our proposed methodology over a set of different B+-tree indexes and real-life application workloads, by investigating their ability to operate efficiently in terms of performance, energy consumption, and NVM memory accesses and their impact on the underlying hardware infrastructure. iii) We propose a performance, energy and NVM lifetime-aware data structure refinement and placement over heterogeneous memory systems achieving up to 58.7% lower execution time, 48.3% less energy consumption and 72.6% less NVM write operations compared to the results obtained by the initial versions of the applications. Finally, iv) we present an extensive exploration and characterization of various in-memory Spark applications and workloads over heterogeneous multi-tier memory systems. Driven by the obtained results, we identify a set of guidelines derived by deploying Spark in-memory analytics over heterogeneous memory tiered systems.

This chapter's text strongly relies on the following publications:

- *Katsaragakis Manolis, Konstantina Skovola, Christos Baloukas, Papadopoulos Lazaros, Catthoor Francky, & Soudris Dimitrios. Performance and Energy Consumption Exploration Methodology for Persistent B+-tree Indexes over Heterogeneous DRAM/NVM Systems, IEEE Transactions on Sustainable Computing(TSuC), 2024 (Under review).*
- *Katsaragakis Manolis, Christos Baloukas, Papadopoulos Lazaros, Catthoor Francky, & Soudris Dimitrios. Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems, ACM Transactions on Architecture and Code Optimization (TACO), 2024 (Under review).*
- *Katsaragakis Manolis, Baloukas Christos, Papadopoulos Lazaros, Kantere Verena, Catthoor Francky, & Soudris Dimitrios (2022, December). 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) (pp. 75-84). IEEE. [160]*
- *Katsaragakis Manolis, Masouros Dimosthenis, Papadopoulos Lazaros, Catthoor Francky, & Soudris Dimitrios (2023, May). On the Implications of Heterogeneous Memory Tiering on Spark In-Memory Analytics. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 945-952). IEEE. [240]*

6.1. Energy Consumption Evaluation of Optane DC Persistent Memory for Indexing Data Structures

6.1.1. Introduction

The emerging memory technologies provide higher density than DRAM, enabling increased aggregate memory capacities with fewer nodes, having positive impact on the energy consumption, resilience and sustainability. Additionally, the data persistence features of the NVM technologies can be used to provide fault tolerance support to applications. On the other hand, the Optane DCPM provides, in general, higher access latency and lower bandwidth compared to DRAM [162, 163, 170]. However, recent studies indicate that the performance of the Optane DCPM depends a lot on the workload access pattern and size [170].

The contribution of the Optane DCPM technology to enable more complex and effective HPC simulations has been investigated in several recent works [116, 163, 164, 166, 241]. They indicate that using Optane DCPM as a volatile memory and DRAM as a cache enables close to DRAM performance. However, directly replacing the DRAM with Optane DCPM, without the use of DRAM as a cache, significantly reduces performance, due to effects such as the write throttling and concurrency contention [165].

Alternatively, the Optane DCPM can be used as a persistent storage medium to enable the storage and processing of massive volumes of scientific data gathered by simulations or instruments [242, 243]. For instance, Large Hadron Collider and the Square Kilometre Array are examples of scientific simulations that generate amounts of data in the order of Petabytes per day [244]. Knowledge extraction through analytics and visualization techniques from stored data can benefit from new disruptive technologies, including the memory and storage capabilities offered by advanced persistent memory architectures. Recent studies indicate that in order to take advantage of the scalability opportunities that storage systems enabled by persistent memories can offer to applications, the indexing data structures of the storage system need to be adapted in order to address the challenges and limitations that the NVM technologies impose [48, 245, 246]. For example, a fundamental challenge when NVMs are employed for persistent storage is the data consistency. Since modern CPUs are designed for volatile DRAM architectures, they typically cache and reorder memory writes, as they target higher performance. Therefore, existing indexing structures, designed for DRAM only systems, cannot be directly deployed on heterogeneous hierarchies, as they do not take into account the persistent nature of NVMs, as well as other challenges, such as the asymmetry in terms of read and write latency and the limited write endurance of NVMs.

Towards this direction, several works exist in the literature that propose indexes specifically designed for NVMs [48, 245–248]. Some recent works provide performance-based evaluation of B+ tree indexing data structures for persistent memories [158, 167], while others convert DRAM-based B+ trees, tries, radix trees, and hash table indexes into NVM-oriented and evaluate their impact in terms of metrics such as the throughput and cache utilization [42].

This work targets HPC developers who consider the use of Optane DCPM as a persistent storage for applications which access large amounts of data through indexing data structures. The existing works in the literature, either focus on the evaluation of the Optane DCPM as a volatile main memory, or provide a performance characterization of its persistence features using the Optane DCPM as a storage device. Although these works are very relevant, they still miss the important energy component in the global analysis. Therefore, to address this gap, this work focuses on the energy consumption evaluation of the Optane DCPM configured as a persistent storage. The two major contributions of

this work are the following:

- A thorough evaluation of the Optane DCPM as a storage medium, in terms of energy consumption, using representative and well-established B+ tree indexing data structures, triggered by various types of YCSB workloads.

- A combined performance and energy consumption evaluation of the LevelDB key-value store deployed on an Optane DCPM and an energy and throughput comparison with the corresponding deployment on an SSD.

To the best of our knowledge, this is the first work that entirely focuses on the energy efficiency aspects of the Optane DCPM. Although there exists several works that investigate the performance behavior of this new memory architecture, as detailed in the following sections, the energy consumption characteristics of the Optane DCPM have been evaluated to a very limited extent, yet.

6.1.2. Evaluation Methodology

In order to select representative B+ tree indexing implementations designed for persistent memories, we first identified their main design goals as described in the relevant literature. These goals are outlined below:

- **Write operations minimization**. Write operations in the Optane DCPM are expensive in terms of latency. Therefore, indexes developed for persistent memories utilize an array of methods to minimize write operations like reduced logging when performing atomic operations, or avoiding the need for CPU cache flushes that are often needed to maintain consistency of the data on the persistent memory.

- **Efficient locking mechanisms**. In order to achieve high concurrency, read operations should be done in a consistent but efficient way across multiple threads to utilize the available bandwidth as much as possible. On the other hand, write operations should make use of as fine locks as possible, constraining the critical sections to necessary code only.

- **Efficient index traversal**. Several decisions play an important role in index traversal, like sorting, the type of tree, key splitting etc. Some indexes are implemented targeting increased data locality, while others store a part of the tree on DRAM for fast traversal.

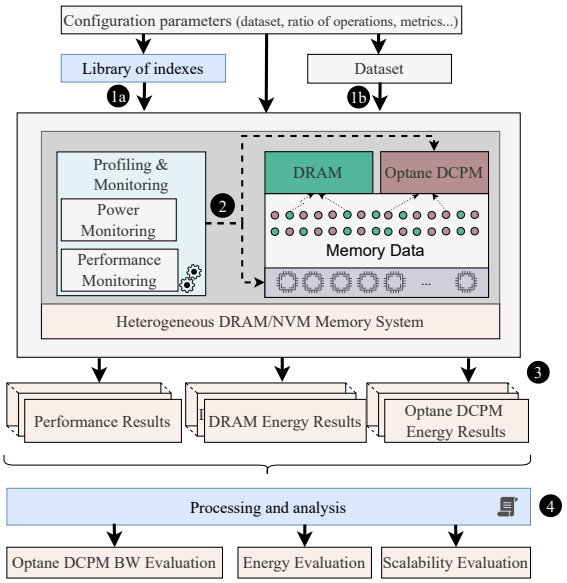


Figure 6.1.: Overview of the evaluation methodology

These goals are materialized differently for each index implementation as they try to offer persistence and concurrency in an efficient way. Many different implementations have been proposed including trees, hash tables, tries, radix trees etc. [42]. In this work we focus on tree implementations, as they are the most commonly used in practice.

We design a systematic methodology to enable the effective execution of four widely used indexes in the literature, which is illustrated in Fig. 6.1. The methodology consists of four distinct steps. The inputs of the methodology are a set of indexes 1a and a dataset 1b, which are configured by a set of parameters, such as the workload type (ratio of operations, skewness, etc.), the selected indexes for evaluation and the metrics of interest (e.g. throughput, energy consumption, scalability). The library of indexes is based on the source codes provided by [42] and includes the B+ tree implementations listed below. These indexes were selected based on the criterion of representing different ways of implementing the aforementioned design goals.

- **wBTree** is a type of main-memory B+ Tree, aiming to reduce the overhead caused by extensive NVM writes and CPU cache flush operations. To minimize the write operations wBTree uses write atomic B+ trees that can achieve node consistency either through atomic writes in the nodes or by redo-only logging, thus reducing the necessary write operations on Optane [47]. Furthermore, the keys are maintained in

a sorted order throughout the execution and the nodes employ a small indirection slot array and/or a bitmap, to quickly check for a particular key's existence. The use of the indirection slot array has the added benefit of not requiring the movement of index entries, for most insertions and deletions, thus further reducing the amount of write operations.

- **Masstree:** is a cache-efficient, highly concurrent trie-like concatenation of B+ tree nodes, and provides high performance even for long common key prefixes. We use the Masstree version converted for NVM by [42]. The Masstree combines a trie and B+ tree implementations to achieve higher performance for long keys. Particularly, long string keys are split into more than one node, so that searching is much faster, because of the faster comparisons between the common part of the keys. Masstree uses a versioning system for concurrency that improves locking performance. Furthermore, to reduce write operations, Masstree inserts new keys to leaf nodes by appending a new key-value pair to the node in an unsorted order. Internal nodes remain sorted for faster traversal.

- **Fast&Fair:** is a persistent memory B+ tree that provides lock-free reads [42]. The reads detect and tolerate inconsistencies such as duplicated elements. By making read operations tolerate transient inconsistency, Fast&Fair avoids expensive copy-on-write, logging, and even the necessity of read latches so that read transactions can be non-blocking [249]. These characteristics of Fast&Fair reduce write operations, while improving locking performance.

- **FPTree:** is a B+ Tree alternative that implements selective persistence by requiring only leaf nodes to be stored persistently. Internal nodes can be rebuilt in case of power failure. Therefore, the FPTree places all internal nodes on the DRAM for fast traversing and consequently higher performance, while the leaf nodes are placed on a persistent memory. Furthermore, the leaf nodes implement *fingerprinting*, where a one byte hash of each key contained within the node is placed at the first cacheline-sized section of the node [48]. Fingerprinting improves lookup performance by speeding up the process of finding if a key is included in a particular node, without having to traverse the node itself. For concurrency, FPTree uses Hardware Transactional Memory (HTM) [250] and fine-grained locks for the internal and leaf nodes respectively.

The *Profiling & Monitoring* component (❷) of the methodology is responsible for power and performance monitoring and it is based on Intel's Processor Counter Monitor (PCM) [26], a tool that allows energy/power sampling over DIMMs through hardware counters [251]. The sampling rate for the profiling is set to 0.01s, by default. More specifically, we use

the *PCM-Power* component to monitor the energy consumed in the DRAM and Optane DCPM DIMMs. For each individual DIMM, the *PCM-Power* reports the on-DIMM energy consumed for the period of time based on our defined sampling rate. The memory accesses on the Optane DCPM are measured by the *ipmctl* utility, which is the standard for configuring and managing Optane DCPM modules, by monitoring the corresponding read and write accesses of each DIMM, respectively [252]. Finally, to calculate the energy consumption of an SSD, we use profiling information about I/O reported by the Intel VTune Amplifier regarding the page read, write and flush operations and the power values from the manufacturers' data-sheets.

The measurements are grouped (❸) and processed (❹) in order to provide the throughput, energy consumption and scalability results for each evaluation experiment. The generation of the aforementioned results is performed automatically, based on a set of customizable Python and shell scripts. The source codes of the tool with usage instructions and the library of indexes are publicly available [1].

6.1.3. Evaluation

Experimental Setup Through the evaluation process we aim at the following:

- To evaluate the energy consumption behavior of the Optane DCPM for indexing data structures triggered by various types of workloads.
- To study the performance and the energy efficiency of the Optane DPCM for a key-value store (LevelDB), using the corresponding deployment on an SSD as a baseline.

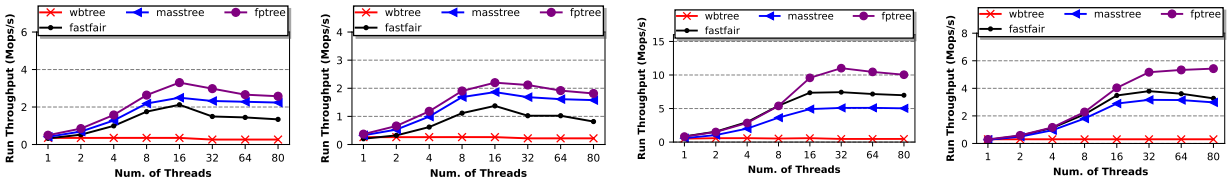
Table 6.1 shows an overview of the experimental setup. The experiments were conducted on a single node with a 2x20 core Intel Xeon Gold 5218R CPU @2.10GHz with 4x32GB DDR4 DIMMs and 2x256GB Optane DC NVDIMMs. Intel Optane DC is configured in *App Direct* mode with ext4-DAX file system. We utilized the version 1.11 of PMDK [140] and gcc-9.4. The monitoring and profiling tools were reported earlier, in Section 6.1.3.

The YCSB is used to trigger the indexes under different types of workload: balanced, write-heavy (which corresponds to insert operations), read-heavy (i.e. lookup operations)

[1] https://github.com/mkatsa/PENVMTool

Table 6.1.: Overview of experimental setup

Experimental Setup	
CPU, DRAM	Xeon Gold 5218R CPU, 2x20 cores @2.10GHz, with 4x32GB DDR4 DIMMs
Optane DCPM Configuration	2x256GB DIMMs, AppDirect Mode, ext4-DAX
SSD	Intel SSD S4500 Series, 480GB
Indexes	wBTree, Masstree, Fast&Fair, FPTree
Key-value store	LevelDB
YCSB Workloads	Balanced (50% reads, 50% writes), Write-heavy (99% writes), Read-heavy (100% reads), Scan-heavy (99% scan)
Workload keys	8-bytes integers
Operating system	Ubuntu 20.04.2 LTS, kernel 5.4.0-121-generic



(a) Balanced workload **(b)** Write-heavy workload **(c)** Read-heavy workload **(d)** Scan-heavy workload

Figure 6.2.: Scalability evaluation on Intel Optane DCPM for different types of workloads.

and scan-heavy (i.e. range-lookup operations) [2]. Each workload consists of 64M operations with 8-byte random integer keys.

Baseline Performance-oriented Results First, we present a set of baseline performance results, which enable a comprehensive insight into the energy consumption evaluation, which follows.

Fig. 6.2 shows the throughput on the Optane DCPM for a variety of YCSB workloads. It demonstrates the scalability of each index under different workloads. It can be noticed that the *fptree* scales up to 32 threads for the read-heavy and scan-heavy workloads, by exceeding 10Mops/sec and 5Mops/sec, respectively. For the balanced and write-heavy workloads, the *masstree*, *fptree* and *fastfair* provide maximum throughput up to 3.3Mops/sec and 2.3Mops/sec at 16 threads, respectively. The *wbtree* does not scale, since the implementation used in this work is single-threaded and is demonstrated as a sequential baseline for the rest of the B+ tree implementations.

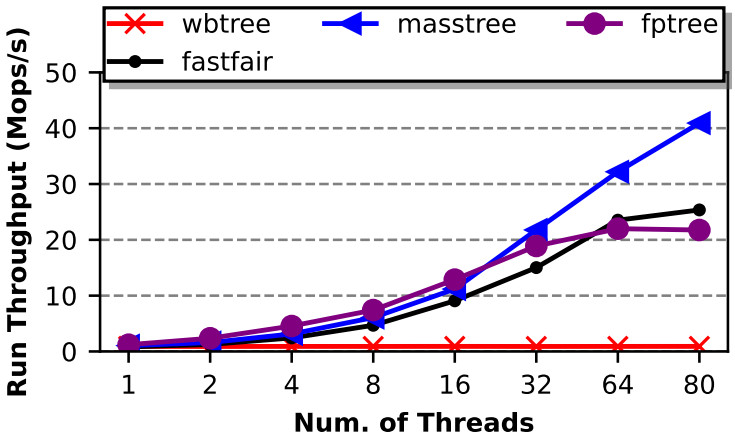


Figure 6.3.: Scalability evaluation on DRAM-only for balanced workload.

For comparison, we evaluate the scalability of the indexes on a DRAM-only memory system. These baseline results are shown in Fig 6.3. All indexes, and especially the *masstree*, scale well under the balanced workload up to 64 threads. Moreover we observe that the throughput achieved for *masstree* index is greater than the *fptree* on DRAM only, in contrast to the Optane DC execution, where *fptree* outperforms *masstree*. This is due to the fact that *fptree* places all inner nodes on DRAM and only leaf nodes on Optane, thus performing better on higher number of threads especially on read-only workloads. Similar results were observed for the rest of the workload types.

To investigate the impact of the Optane DCPM hardware configuration on the maximum throughput that can be reached, we evaluate the scalability performance of the indexes on a memory system with a single Optane DCPM DIMM, under balanced workload. By comparing the results of Fig. 6.2a (two DIMMs) with Fig. 6.4 (single DIMM), it is noticed that the use of more Optane DCPM DIMMs on the same socket enables higher parallelism, allowing improved scalability and higher throughput. Indeed, on a single DIMM, the maximum throughput is obtained at 4 threads and barely exceeds 1Mops/sec.

The above observations are inline with performance evaluation results in the existing literature. In particular, the scalability limitations of B+ tree indexes on Optane DCPM have been studied in recent works, relying on custom microbenchmarks [158]. Our findings, based on the YCSB, further confirm the observed scalability limitations.

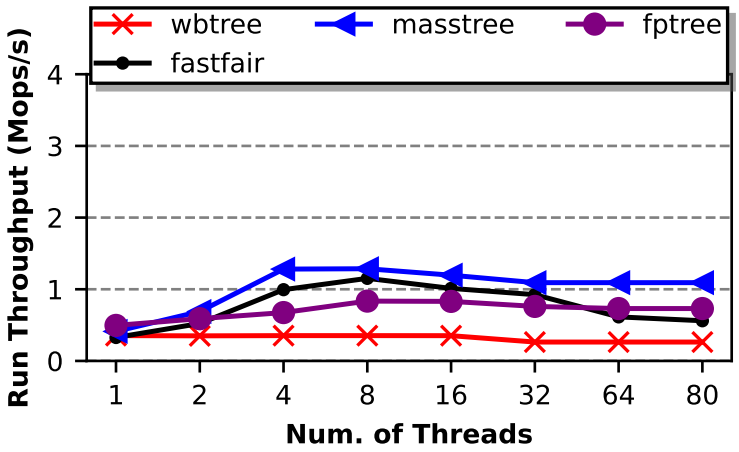
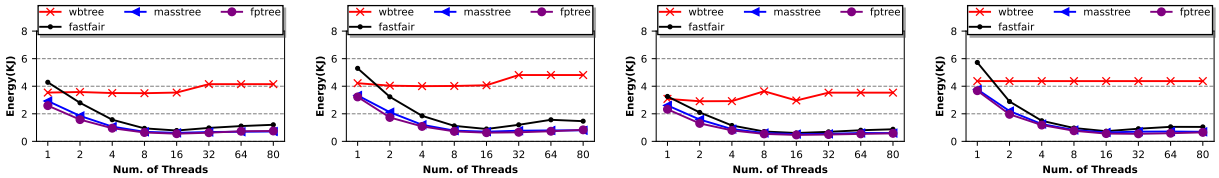


Figure 6.4.: Throughput for balanced workload using a single Optane DCPM DIMM.



(a) Balanced workload **(b)** Write-heavy workload **(c)** Read-heavy workload **(d)** Scan-heavy workload

Figure 6.5.: Energy consumption evaluation on Intel Optane DCPM for different types of workloads.

Energy Consumption Results Fig. 6.5 shows the energy consumed in the Optane DCPM DIMMs for each index under different types of workload, as measured through the sensors equipped in the DIMMs. In general, the energy consumption is gradually reduced up to 8 or 16 threads, indicating a direct correlation between the throughput reached and the energy consumed. The energy of *wbtree* index energy remains slightly constant. Since wbtree does not scale (Fig. 6.2), this leads to utilization of the memory resources for higher amount of time, therefore leading to increased energy consumption.

Table 6.2 shows the energy consumption values for 16 threads. We notice that the energy consumption for different indexes under the same workload does not vary significantly. However, the Optane DCPM consumes less energy under the read-heavy workload compared to the rest of the workload types. For example, triggered by the read-heavy, the *fptree* index yields 16% lower energy consumption compared to the corresponding balanced workload, indicating the impact that the increased throughput under a read-heavy workload has on the energy (Fig. 6.2).

Table 6.2.: Energy consumption (KJ) of Optane DCPM DIMMs for each workload type for 16 threads.

	Balanced	Write-heavy	Read-heavy	Scan
wbtree	3.539	4.069	2.966	4.372
masstree	0.615	0.701	0.524	0.645
fptree	0.554	0.627	0.465	0.562
fastfair	0.787	0.896	0.620	0.738

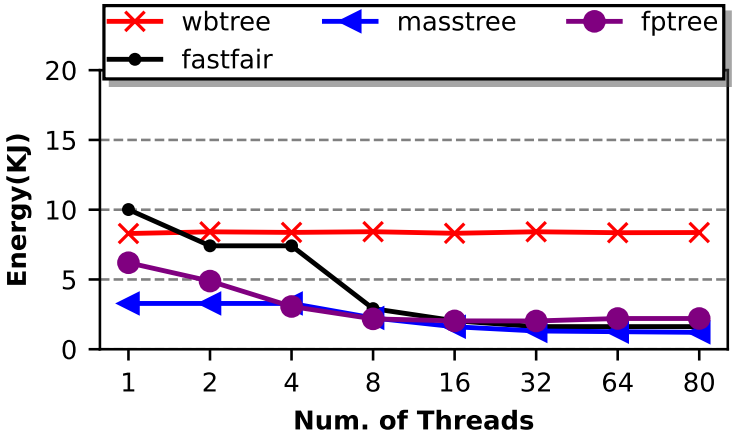


Figure 6.6.: Energy consumption on DRAM-only for balanced workload.

As a baseline for the Optane DCPM energy consumption, Fig.6.6 shows the energy for the balanced workload in a DRAM-only system (i.e. with Optane DCPM deactivated). Similarly to Fig. 6.5, the results correspond to the energy consumed on the DRAM DIMMs, only. The energy is significantly reduced after 4 or 8 threads, in accordance to the throughput increase demonstrated in Fig. 6.3. Additionally, by comparing the energy values of Fig. 6.6 with Fig. 6.5a, we notice that the energy consumed in the Optane DCPM is about half the one in DRAM. As an example, the energy consumption when deploying the *fptree* with 16 threads on Optane DCPM is 56.3% lower compared to the corresponding DRAM experiment.

The total energy consumption, i.e. of both the CPUs and the memory system, is shown in Fig. 6.7. The throughput still impacts the energy consumed, as it is minimized at 8-16 threads, where the maximum throughput is reached. However, after that point, the energy consumption slightly increases, as a result of the throughput decrease after 16 threads for most of the indexes, as depicted in Fig. 6.2.

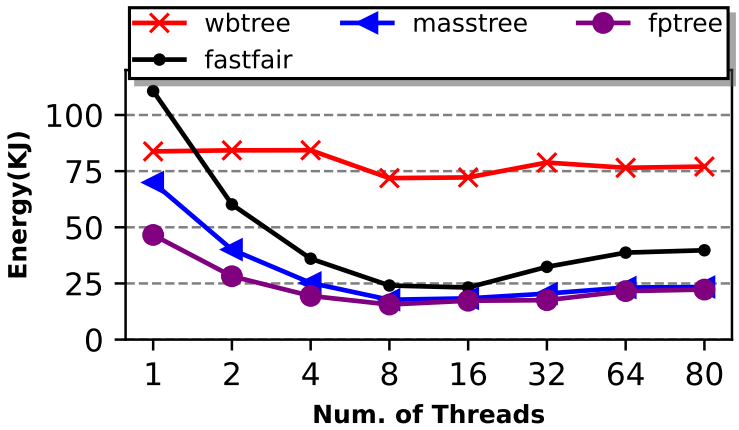
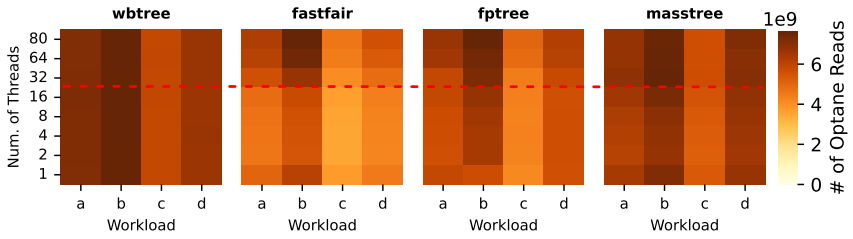


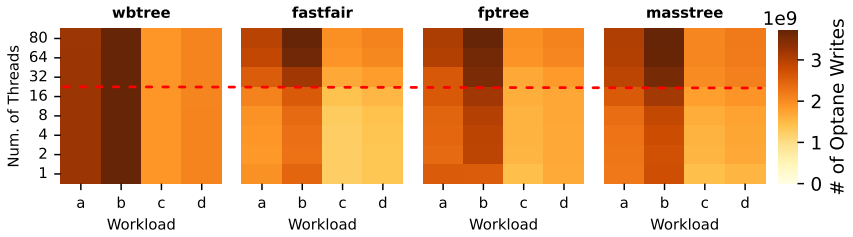
Figure 6.7.: Energy consumption of CPUs and memory system.

To further investigate the energy consumption behavior of the Optane DCPM as the number of thread increases, we monitor the number of read and write memory accesses in the persistent memory for each index, for an increasing number of threads. The profiling results are shown in Fig. 6.8. It is noticed that the number of read and write accesses beyond 16 threads increases significantly, due to the internal mechanisms of the indexes and the overhead of concurrency, without a positive impact on performance, as shown in Fig.6.2a. In particular, when the bandwidth is capped like in the case of 1 or 2 DIMMs of Optane DCPM, a significant amount of CPU time is spent by trying to acquire exclusive access to protected regions by each index, accessing internal index variables for locking and synchronization and thus increasing the overall access count. Therefore, using threads beyond that point has negative impact both on the throughput and the energy consumption. Additionally, the increase in the memory access count is expected to have impact the endurance of the persistent memory hardware [253]. Interestingly, the writing hardware endurance of the Optane DCPM has not been evaluated in the existing literature, yet.

Finally, to investigate the impact of the available persistent memory bandwidth on the energy consumption, Fig. 6.9 shows the energy for the balanced workload when integrating a single Optane DCPM DIMM. An interesting observation is that although when integrating 2 DIMMs the throughput increases almost 2x (as shown by Fig.6.2a and Fig.6.4), the energy is reduced by 5x (e.g. for the *masstree* as shown by comparing the results of Fig.6.5a and Fig. 6.9). Therefore, it is reasonable to expect that increasing the Optane DCPM bandwidth by integrating even more DIMMs will further decrease the energy consumption and increase the throughput.



(a) Intel Optane DCPM read accesses



(b) Intel Optane DCPM write accesses

Figure 6.8.: Number of Intel Optane DCPM read and write memory accesses. Workload a: balanced, b: write-heavy, c: read-heavy, d: scan-heavy.

LevelDB key-value store Performance-Energy Exploration In this section, we perform a combined performance-energy consumption analysis of the Optane DCPM using the LevelDB key-value store[2].

The LevelDB is a fast key-value storage library developed by Google, where keys and values are arbitrary byte arrays, allowing any type of data to be sorted by a key. LevelDB is not an SQL database and it does not provide an interface for SQL queries making it more inline with the other indexes described in this work. The only difference is that the LevelDB keeps data stored on disk, while maintaining a configurable buffer in the memory for performance. The inherent support for data persistence on HDD/SSD and data caching in DRAM make it a suitable candidate for demonstrating the benefits of storing the whole database in an Optane DCPM instead of an SSD. The LevelDB uses a log-structured merge-tree (also known as LSM tree) as a data structure for storing data. The LSM tree comprises two parts: one tree in the memory serving as a buffer and another on the disk. The advantage and particular characteristic of the LSM trees is that each part can be configured differently for the platform they reside.

After integrating the YCSB in the LevelDB, we evaluate the two memory hierarchy configurations depicted on Fig. 6.10, in terms of throughput and energy consumption, under different workloads. In the first experiment, the LevelDB is stored on the SSD,

[2]LevelDB: https://github.com/google/leveldb

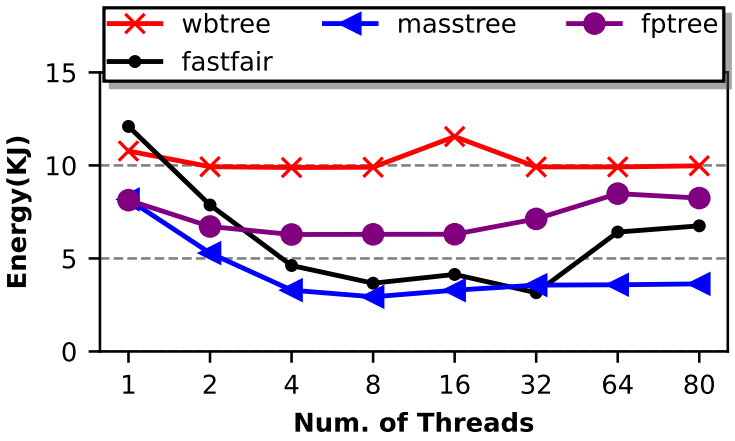


Figure 6.9.: Energy consumption on a single Optane DCPM DIMM for balanced workload.

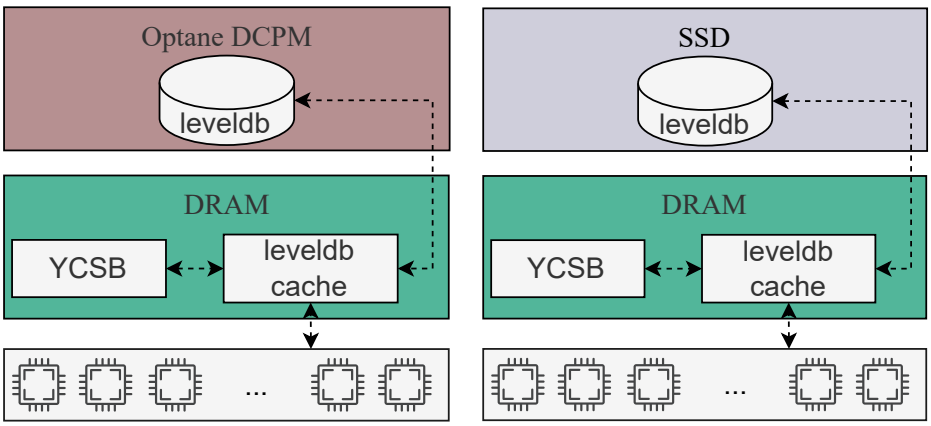
while in the second it is allocated in the Optane DCPM. The tools employed for profiling and monitoring are described earlier, in Section 6.1.3. For brevity, we present results for the balanced and read-heavy workloads only, but the observations apply to the write-heavy and scan workloads, as well.

The throughput results are presented in Fig. 6.11a and Fig. 6.11b, for balanced and read-heavy workloads, respectively. We notice the following:

- The maximum throughput for the Optane DCPM is reached at 16 threads for the balanced workload and at 32 for the read-heavy. This result is inline with the observations based on the indexing data structure experiments on Optane DCPM (Fig. 6.2a and Fig. 6.2c). For the LevelDB, increasing the number of threads beyond these points, results in performance degradation on the Optane DCPM, as noticed in most of the corresponding experiments using B+ tree indexes (Fig. 6.2).

- Increasing the number of threads beyond the point of the maximum throughput, results in an increasing memory access count on the Optane DCPM. Indeed, for the balanced workload, the number of accesses increases by 27.4% for 32 threads and by 39.1% for 64, compared to the number of accesses on 16 threads. Similarly, for the read-heavy workload it increases by 13.6% for 64 threads compared to 32. These results correspond to the ones demonstrated in Fig. 6.8 for the indexing data structures experiments. The increase in the memory access count is attributed to the overhead of the synchronization mechanisms and is expected to have negative impact on the Optane DCPM hardware endurance.

- Finally, Fig. 6.11a and Fig. 6.11b demonstrate the performance gains which can



(a) DRAM/SSD configuration. **(b)** DRAM/Optane DC onfiguration.

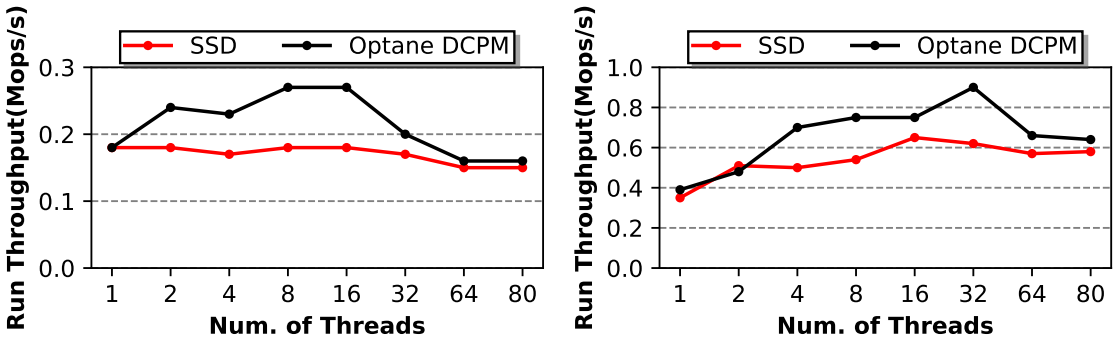
Figure 6.10.: System configurations for the LevelDB experiments: DRAM/SSD and DRAM/Optane DCPM.

be obtained for the LevelDB when stored in an Optane DCPM instead of an SSD. Performance gains in terms of throughput reach 37.3% for the balanced workload (at 16 threads) and 34.6% for the read-heavy at 32 threads.

The energy consumption results are presented in Fig.6.12a and Fig.6.12b for the balanced and the read-heavy workloads, respectively. Detailed measurements on (i) CPUs, DRAM DIMMs and SSD, (ii) Optane DIMMs only, (iii) CPUs, DRAM and Optane DIMMs are presented. We observe the following:

- As the number of threads increases, the total energy consumption for the balanced workload for the Optane DCPM experiment (i.e. CPU, DRAM, Optane DCPM) slightly drops reaching a minimum value of 2.6KJ at 8 threads, which is 16% lower than the energy for a single thread. The energy consumption on the Optane DCPM DIMMs is also in direct correlation with the throughput (Fig.6.11a). After that point, the energy increases following the decrease in throughput. In contrast with the Optane DCPM experiments, the energy consumption when the LevelDB is stored in the SSD is relatively independent of the number of threads for the balanced workload, slightly decreasing at 16 threads, by 2.1% compared to the single thread experiment.

- For the read-heavy workload, the total energy consumption for the Optane DCPM experiment (i.e. CPU, DRAM, Optane DCPM) drops to 0.7KJ at 32 threads, where the throughput is maximized. Compared to the single thread experiment, the energy is decreased by 31%. The energy consumed by the Optane DIMMs only, is



(a) Balanced workload

(b) Read-heavy workload

Figure 6.11.: LevelDB throughput for different memory configurations under balanced and read-heavy workload.

also minimized at 32 threads. Additionally, the energy when storing the LevelDB to the SSD drops up to 32 threads and increases after that point. Another interesting observation is the fact that the energy consumed by the Optane DCPM is about the same between the experiments of different workload types (i.e. about 0.5KJ for both the balanced and read-heavy workloads). For the read-heavy workload the energy consumed by the CPU and DRAM DIMMs is between 0.4-0.6KJ, depending on the number of threads, as shown in Fig. 6.12b. However, for the balanced workload, a much larger portion of the total energy is consumed by the CPU and the DRAM (about 2.5KJ, which is almost 6x higher than the energy consumed by the Optane DCPM DIMMs only, as shown in Fig. 6.12a).

- Significant energy gains can be obtained by storing the LevelDB to an Optane DCPM, instead of an SSD. For example, for the read-heavy workload, the energy consumption is reduced up to 71.2%. This demonstrates the efficiency of Optane DCPM as a storage medium, it terms of energy.

Discussion In this subsection we highlight the main key findings based on the analysis of the experimental results and we indicate some major open research directions.

The Optane DCPM is a relevant persistent memory architecture for data intensive HPC applications, providing advantages in terms of both throughput and energy consumption compared to the typical SSD storage. However, the bandwidth limitations of the Optane DCPM need to be taken into consideration. The evaluation results indicate that **fine-tuning the deployment of the indexing data structures on an heterogeneous memory system** is needed, in order to provide significant gains in terms of combined

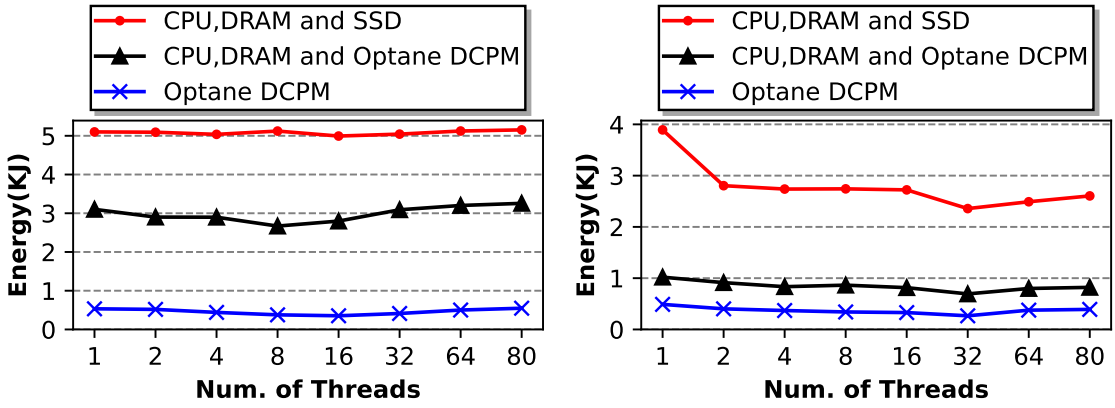
energy consumption and performance.

The hardware configuration is critical for the scalability and the energy consumption of applications deployed on the Optane DCPM. Based on the experiments of Fig. 6.4 and Fig. 6.2a, we can argue that by integrating more DIMMs into the memory system, higher throughput will be reached due to improved parallelism. It is reasonable to expect that improvements in throughput will result in lower energy consumption. However, it will be interesting to investigate the existence of a saturation point, after which integrating more DIMMs does not contribute to further energy efficiency.

The write indexing operations are costly in terms of energy consumption on the Optane DCPM compared to the read-only (e.g. lookup operations). However, the increased amount of energy is not consumed by the Optane DIMMs, as demonstrated in the LevelDB experiments of Fig. 6.12 (but also in Fig. 6.5). Instead, it is consumed by the CPUs, due to the synchronization and consistency overhead, requiring cache flushes and similar instructions to maintain the data coherence in the persistent device. This energy consumption overhead can be observed both in the B+ tree and LevelDB experiments. Hardware support for data consistency targeting NVMs is expected to benefit the energy efficiency of the whole system significantly.

Although the analysis of the experimental results was based on experiments using 8-byte integer keys, **similar observations can be obtained by the use of other types of keys**. As an indicative example, for 24-byte string keys on Optane DCPM, using the same experimental setup, we noticed the energy consumed by the Optane DCPM DIMMs to follow a similar pattern with the corresponding integer-key experiments. Particularly, it is reduce up to 8 and 16 threads for the balanced and read-heavy workloads respectively, and reaches a plateau after that point.

During the profiling analysis of Section 6.1.3, we have evaluated the scalability of the dynamic allocations using the *libvmmalloc* library of the PMDK [140]. We noticed that the ***libvmmalloc* does not scale with the number of threads**. This is inline with observations in the existing literature [158]. This limitation of the *libvmmalloc* will affect the scalability and the energy consumption of the indexing data structures on the Optane DCPM, as well as the performance of dynamic applications which make heavy use of the heap memory and are deployed to Optane DCPM through the PMDK libraries. Custom and scalable dynamic memory allocators for persistent memories are expected to enable higher performance and lower energy consumption for such applications.



(a) Balanced workload **(b)** Read-heavy workload

Figure 6.12.: LevelDB energy consumption for different memory configurations under balanced and read-heavy workload.

6.1.4. Conclusion

This work provides new insights about the Optane DCPM as a persistent storage device. Based on a comprehensive evaluation of indexing data structures triggered by various types of workloads, we investigate the energy consumption characteristics of the Optane DCPM. We show that significant performance and energy consumption gains can be obtained by the effective use of Optane DCPM integrated in a memory system, compared to a typical SSD. However, proper configuration is required, considering the bandwidth limitations of the Optane DCPM, in order to benefit from the advantages it offers. The outcomes of this work can be exploited by developers who target the Optane DCPM as a persistent storage medium for HPC applications.

6.2. Performance & Energy Consumption Exploration for Persistent B+-Tree Indexes over Heterogeneous DRAM/NVM Systems

6.2.1. Introduction

Over the last years, the digital landscape has undergone a profound transformation, marked by an exponential surge in database applications and workloads across various industries and domains. The rapid expansion of database applications, such as online transaction processing (OLTP) [254] and online analytical processing (OLAP) [255] workloads has significantly stressed existing computing systems. The huge amount of data and queries processed and stored by enormous databases, such as Google's Bigtable [256] and Amazon's DynamoDB [257], along with the computational demands of emerging technologies, such as GPT [258], raises concerns about the environmental impact of such digital ecosystems. For instance prior research [259] has shown that for training GPT-3 an end-to-end training time of 84 days is required, when deployed on a cluster of multiple GPUs, thus leading to resource explosion and huge energy waste, highlighting the necessity for energy-effective solutions. Moreover, the escalation of big data, cloud computing and HPC platforms has pushed the development of sophisticated data-driven applications that drive decision-making, while the rise of massive datasets and the complexity of analytical queries have stretched the boundaries of traditional computing infrastructures, pushing organizations to seek more robust, scalable and sustainable solutions [260].

Despite the low access latency that traditional DRAM technologies offer, the increased leakage and refresh power requirements limit DRAM scalability and introduce challenges towards reaching sustainable characteristics and further adjusting to the density roadmap [261]. Non Volatile Memory(NVM) technologies have been introduced to address these limitations. Intel Optane DC Persistent Memory(DCPM), which relies on 3D-XPoint architecture, is the only commercial NVM technology up to now and it has been widely integrated into existing systems alongside traditional DRAM DIMMs. For instance, state-of-the-art systems, such as the DAOS [135], the Alibaba Cloud [262] integrate up to petabytes(PB) of Optane DCPM, while Amazon and Google [263] offer Optane memory to Cloud users, aiming to provide both high performance, low energy consumption, increased capacity and lower cost per byte.

Even though NVMs and Intel Optane DCPM provide significant benefits, they also impose some limitations. Increased access latency/lower bandwidth compared to DRAM and reduced write endurance are some of the most dominant limitations that affect NVM

technologies. Furthermore, Optane DCPM provides asymmetry on the latency and energy of read/write operations [264]. Nevertheless, prior studies have indicated that the performance and energy consumption of applications deployed on Optane DCPM varies significantly based on the application's pattern [165], e.g. increased randomness leads to insufficient utilization of Optane DCPM. Optane DCPM can be either utilized: i) as a volatile memory and DRAM as a cache, enabling close to DRAM performance (`Memory Mode`) and ii) as a persistent storage to enable the storage and processing of huge workloads (`AppDirect Mode`).

The majority of memory indexes for database applications have been traditionally designed to target single-DRAM systems [42]. However, recent studies consider the implementation of specifically designed indexes or the re-design of existing indexes to target persistent memories [42–44,46,48–51], such as B+-trees and hash schemes. The major objective is to provide either performance optimization [44] and/or energy reduction [160]. The evaluation of persistent memory indexes is not as straightforward as they are sensitive to hardware/software parameters, e.g., the number of DIMMs, and can lead to high diversity in the evaluation of the indexes, as it is already observed in existing studies [42, 160], thus sophisticated exploration is essential. Additionally, existing works do not focus on the impact of the indexing structures on the underlying hardware, while the triptych of performance, energy and NVM accesses is not mutually targeted as a multi-dimensional exploration space, thus leading to non-versatile solutions which are essential for state-of-the-art database systems.

In this section, we focus into the challenges imposed by the burgeoning database workloads, emphasizing the critical importance of high-performance and sustainable computing as an approach to addressing the demands of this data-driven era. More specifically, we propose an exploration framework for persistent B+-Tree indexing structure refinement based on multi-objective optimization criteria, i.e. performance, energy and NVM's accesses, targeting database workloads over heterogeneous DRAM/NVM memory systems. We formulate and solve a Knapsack-like multi-objective problem, constrained by user-defined and system-level metrics. We target developers who consider the use of Optane DCPM as a persistent storage for database applications that utilize large amount of data and users. Even though existing research provides performance [42] and/or energy [160] characterization and evaluation, none of them provides a multi-criteria exploration approach for systems that targets database developers. The novel contributions of this work are as follows:

- We introduce a novel systematic methodology for multi-objective exploration of representative and well-established B+-tree indexing data structures.

- We develop a Knapsack-based decision-making approach to derive Pareto optimal

Table 6.3.: Key design characteristics of the integrated persistent B+-tree indexes

Index	Architecture	Sorting	Parallelism	Concurrency	Logging	String Keys
P-WBTree	B+-tree	*X*	*X*	*X*	Redo-only logging	Pointer to key
Fast&Fair	B+-tree	✓	✓	Lock-free reads; Blocking writes	*X*	Pointer
P-BWTree	Converted B+-tree	✓	✓	Non-blocking reads/writes	Log-structured storage	Inline
P-Masstree	Trie-like B+-trees	Partially	✓	Lock-free reads; Write exclusion	*X* (node-versioning)	Inline

solutions, targeting performance, energy, and NVM lifetime system and user constraints.

- We conduct an extensive experimental analysis, offering key insights into the impact of B+-tree indexes on underlying NVM hardware. Notably, we are the first to explore, co-optimize, and derive insights on the impact of B+-tree persistent memory indexes on NVM energy consumption, lifetime, and bandwidth.

6.2.2. Proposed Methodology

Our proposed methodology has been designed, aiming to satisfy the following objectives: (i) propose a systematic approach for effective indexing exploration for database workloads, (ii) evaluate the impact of different B+-tree indexes over heterogeneous DRAM/NVM architectures and (iii) provide an extensive evaluation, analysis and correlation of various metrics and derive key outcomes for the deployment of database workloads over DRAM/NVM systems. Our methodology aims to support database application developers for effectively selecting persistent B+-tree indexes to be integrated, based on their optimization objectives. Moreover, the following set of steps provides an extensible approach, towards the input indexes, the target workloads and the optimization objectives. Figure 6.13 illustrates an overview of our proposed exploration methodology. The steps of our methodology are automated, while the final outcome of the decision support requires the developer's interaction for selecting one or more of the derived solutions.

6.2.3. B+-tree Indexes Library

As input to our framework, we integrate a library consisting of different B+-tree implementations. All of the implementations are derived by [42] and are integrated into our library of indexes, aiming to provide an efficient library of scalable, lightweight and NVM-friendly indexes. The B+-tree implementations are selected based on the following

criteria:

- **Minimization of write accesses:** NVM technologies suffer from limited write endurance, i.e. the number of write accesses is limited, compared to DRAM technologies. Thus, an efficient persistent memory index should be designed, aiming to reduce the number of write operations of the underlying NVM technology.

- **Efficient locking mechanisms:** The achievement of high concurrency and scalability strongly depends on the way that read and write operations are performed. Therefore, the selection of a persistent memory index depends on the ability of the index to perform consistent and efficient read operations across multiple threads and on the efficiency of locking over write operations.

- **Efficient index traversal:** The key design choices of the selected indexing structure, e.g. traversal, sorting, logging, etc. strongly affect the impact of the index to the whole performance and energy consumption.

- **Logging support:** Logging support is crucial for persistent memory indexes as it enables records for transactional changes. Indexes can ensure data consistency, durability, and recovery in case of system failures, offering critical security against data corruption

Based on the aforementioned selection criteria, we select four B+-tree indexes derived by [42], namely `P-WBTree`, `P-BWTree`, `Fast&Fair` and `P-Masstree`, aiming to combine different characteristics in terms of locking, logging, traversal and write accesses on the NVM. Table 6.3 provides an overview of the B+-tree indexes' characteristics. The integration of the B+-tree indexes relies on the `libvmmalloc` allocator library, which facilitates dynamic memory allocation and deallocation. A short description of the integrated indexes follows.

- **Fast&Fair** introduces a persistent memory B+ tree that facilitates lock-free read operations [247]. The reads are designed to identify and tolerate inconsistencies, including duplicated elements. By enabling read operations to tolerate transient inconsistencies, `Fast&Fair` eliminates the need for costly copy-on-write, logging, and the necessity of read latches, allowing read transactions to be non-blocking. These distinctive features of Fast&Fair contribute to a reduction in write operations and an enhancement in locking performance.

- **P-BWTree** is a highly concurrent B+-tree developed for new hardware platforms which is completely lock-free, in order to scale efficiently on multi-core systems [245].

Tree nodes are represented as logical memory pages, replacing traditional pointers. A mapping table is used to keep track of the correspondence between logical pages and physical memory addresses of the head node of the composing linked lists.

- **P-Masstree** represents a cache-efficient, highly concurrent trie-like concatenation of B+-tree nodes, ensuring high performance, especially for extended common key prefixes [248]. To expedite searches, long string keys are distributed across multiple nodes, facilitating faster comparisons between the shared portions of the keys. `P-Masstree` employs a versioning system to enhance concurrency and optimize locking performance. Additionally, to minimize write operations, new keys are added to leaf nodes by appending key-value pairs in an unsorted order, while internal nodes remain sorted for expedited traversal.

- **P-WBTree** focuses on mitigating the overhead arising from frequent NVM writes and CPU cache flush operations [246]. To minimize write operations, `P-WBTree` incorporates write atomic B+ trees, ensuring node consistency through either atomic writes within the nodes or redo-only logging, thereby diminishing the necessary write operations. Additionally, key maintenance in a sorted order persists throughout execution, and nodes employ a compact indirection slot array and/or a bitmap for rapid verification of a specific key's existence. The utilization of the indirection slot array brings the added advantage of eliminating the need to relocate index entries for most insertions and deletions, thus further curtailing the volume of write operations.

6.2.4. Proposed Methodology

Alongside with the B+-tree indexes library presented in 6.2.3, as input to our methodology we provide a set of configuration parameters, e.g. number of reads, inserts and updates, and the objectives to be optimized. As optimization objectives, we focus on performance maximization, energy consumption reduction and/or write operation reduction, however, this can be expanded to other metrics. The corresponding workloads are fed as input to our heterogeneous memory system. Our approach consists of four distinct steps, namely: i) *Platform-aware Run-Time Profiling & Monitoring*, ii) *Design Space Exploration (DSE) Phase*, iii) *Correlation & Hardware-level Mapping* and iv) *Decision Support*. The insights of each individual step are detailed on the rest of this Section.

Platform-aware Run-time Profiling & Monitoring

The first step of our proposed methodology is the integration of runtime profiling and monitoring. We monitor the power consumed on the DRAM and NVM DIMMs and the execution throughput of the input workload for the different B+-tree indexes. The power is measured utilizing the Intel PCM [26], a tool that allows energy/power sampling over DIMMs through hardware counters. We set the sampling rate to 0.01s. More specifically, we use the `PCM-Power` component to monitor the energy consumption in the DRAM and Optane DCPM DIMMs. Additionally, we monitor multiple system-level metrics through `PCM-Memory`, such as the read/write bandwidth per memory DIMM. Furthermore, we provide application-level monitoring for deriving the execution throughput of the examined input. Last but not least, we monitor the number of NVM accesses imposed by the input B+-tree indexes. In the context of NVM technologies, closely monitoring power metrics and access patterns is crucial for managing energy consumption and mitigating the impact on the NVM lifespan, thereby contributing to the long-term sustainability of the memory system. We utilize the `ipmctl` tool for measuring the number of read and write accesses, which is the standard for configuring, managing and monitoring the NVDIMMs of Intel Optane DCPM. The generation of the aforementioned results is performed automatically, based on a set of customizable Python and shell scripts and the results are propagated to the Design Space Exploration step.

Design Space Exploration (DSE) Phase

Next, the actual DSE phase is triggered, where we conduct an extensive exploration of the B+-tree and workload alternatives. The metrics derived by the *Run-time Profiling & Monitoring Step* are post-processed and analyzed. Per B+-tree index and per workload alternative, a set of different solutions is derived. Each solution corresponds to specific performance, energy and NVM write accesses, thus composing a three-dimensional design space, where trade-offs among the examined optimization objectives can be derived. During the DSE phase, an extensive exploration is performed, aiming to derive the Pareto optimal solutions. The resulting design space is then fed as input for later steps and multi-objective optimization in the *Decision Support* step. Once the Pareto Optimal solutions are derived, they are propagated to the next step of our proposed methodology.

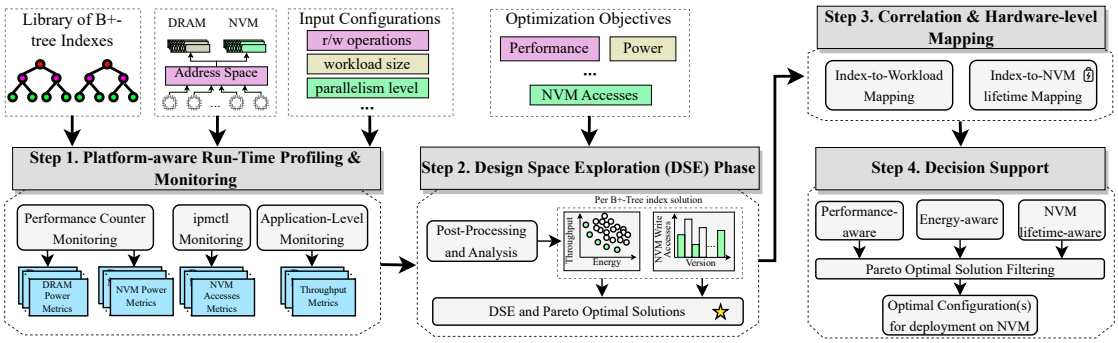


Figure 6.13.: Overview of the proposed exploration methodology and decision support

Correlation & Hardware-level Mapping

The Pareto optimal solutions derived from the *DSE Phase* are fed as input to the *Correlation & Hardware-level Mapping* step. The major objectives of this step are the following: i) Mapping Optimal Input Indexes to Corresponding Workload Types and ii) Mapping High-Level Index Characteristics to NVM Hardware. The former involves identifying how different indexes impact various workload types, while the latter focuses on translating the characteristics of B+-tree indexes, to the specific attributes and limitations of the underlying NVM hardware. In particular, we emphasize the implications of B+-tree indexes on the NVM's longevity. Given that NVM technologies, have limited write endurance, understanding and mitigating wear is critical. We track the frequency and pattern of read and write accesses to the NVM. This includes metrics such as write amplification (ratio of physical writes to logical writes) and read/write (r/w) intensity (number of reads/writes per unit of time). We estimate the lifetime of the NVM based on the observed write patterns and the endurance limits of the memory cells. We employ a methodology similar to that described in [265]. This methodology focuses on mapping the impact of write accesses to the aging of NVM, using detailed analysis and simulation techniques. This information is then fed as input to the final step of our methodology, the *Decision Support* step. In this step, we synthesize all collected data and insights to provide recommendations for system design and optimization, ensuring that the solutions are not only high-performing but also sustainable over the long term.

Decision Support

Finally, the *Decision Support* is triggered, which aims to derive Pareto optimal solutions through solving the multi-objective problem of the performance and energy co-

Table 6.4.: Key system parameters and variables for multi-objective problem optimization

Denotation	**Description**
$tree_i$	Persistent B+-tree i
w_j	Workload j
s_i^j	Solution of $tree_i$ and workload w_j
t_i^j	Throughput of $tree_i$ for workload w_j
e_i^j	Energy of $tree_i$ for workload w_j
l_i^j	NVM lifetime of $tree_i$ for workload w_j
$\mathcal{TU}$	User-defined minimum throughput threshold
$\mathcal{EU}$	User-defined minimum energy consumption threshold
$\mathcal{L}$	Minimum lifetime throughput threshold
x_i^j	1 if tree i of workload j is in the design space ($\mathcal{DS}$), 0 otherwise

optimization under NVM lifetime constraints. Towards this direction, we form a Knapsack-like formulation [266, 267], which aims to maximize a weighted combination of execution throughput and minimized energy consumption while ensuring the selected solutions satisfy constraints on minimum throughput, maximum energy consumption, and minimum NVM projected lifetime.

$$max(\alpha \times \sum_{s_i^j \in \mathcal{DS}} x_i^j t_i^j - \beta \times \sum_{s_i^j \in \mathcal{DS}} x_i^j e_i^j) \tag{6.1}$$

$$s.t. \quad \sum_{s_i^j \in \mathcal{DS}} x_i^j t_i^j \geq \mathcal{TU} \tag{6.2}$$

$$\sum_{s_i^j \in \mathcal{DS}} x_i^j e_i^j \leq \mathcal{EU} \tag{6.3}$$

$$\sum_{s_i^j \in \mathcal{DS}} x_i^j l_i^j \geq \mathcal{L} \tag{6.4}$$

$$x_i^j \in \{0, 1\} \quad \forall s_i^j \in \mathcal{DS} \tag{6.5}$$

Let $\mathcal{DS} = \{s_i^j\}$ be the set of discrete solutions, where each solution s_i^j is a combination of B+-tree $tree_i$ and workload w_j. Our system is designed to accommodate three distinct user-defined decision-making factors, each tailored to specific optimization objectives. Each individual combination of $tree_i$ and workload w_j, is characterized by a

single vector of execution throughput t_i^j, energy consumption e_i^j and projected NVM lifetime l_i^j. We additionally define x_i^j, a binary variable indicating whether solution s_i^j is included ($x_i^j = 1$) or not ($x_i^j = 0$) in the search space derived in *Step. 2*. Additionally, let $\mathcal{TU}$ be the minimum user required throughput, $\mathcal{EU}$ be the maximum user allowed energy consumption. Additionally, our solution is constrained by $\mathcal{L}$, i.e. the minimum NVM lifetime expectancy. Weights α and β are used for combining the user objectives and derive the Pareto optimal solutions. Table 6.4 briefly summarizes the key system parameters utilized in our multi-objective optimization.

The optimization objective is to maximize the execution throughput while minimizing the energy consumption, as indicated in 6.1. The constraints of the optimization problem ensure that the selected solutions meet specific requirements for system performance and resource usage. The first constraint mandates that the total execution throughput of the chosen configurations must be at least the minimum required throughput (Eq. 6.2), ensuring that system performance meets a certain standards. The second constraint limits the total energy consumption to be no more than the maximum allowed energy usage (Eq. 6.3), which helps in maintaining energy efficiency. The third constraint ensures that the projected NVM lifetime of the selected configurations meets or exceeds a minimum threshold, thereby guaranteeing the longevity and reliability of the NVM (Eq. 6.4).

Table 6.5.: YCSB default workload specifications [2]

Workload	Reads	Updates	Scans	Inserts
A - Update Heavy	50%	50%	N/A	N/A
B - Read Heavy	95%	5%	N/A	N/A
C - Read Only	100%	N/A	N/A	N/A
E - Short Ranges	N/A	N/A	95%	5%

6.2.5. Experimental Setup and Evaluation

Experimental Setup Our proposed methodology was evaluated based on the following criteria:

- Provide an extensive comparative evaluation of the examined B+-tree indexes, regarding their performance, energy consumption and NVM accesses when deployed over a DRAM/NVM memory system.

- Investigate the impact of the various B+-tree indexes on the lifetime of the under-

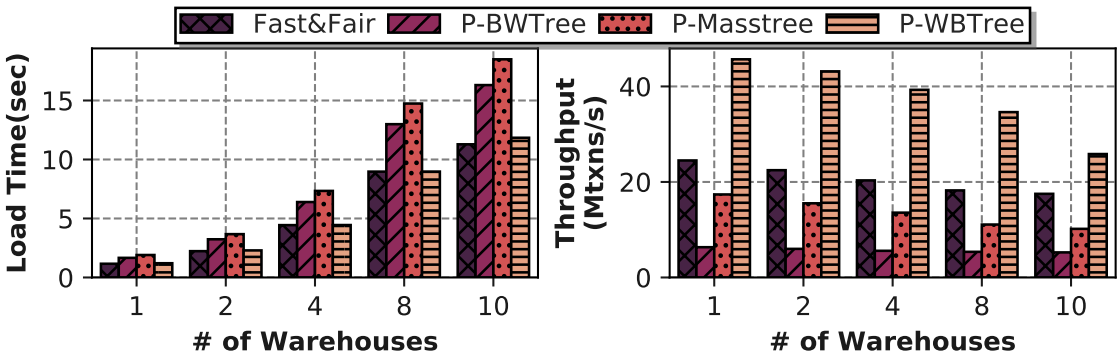


Figure 6.14.: Load time (left) and throughput (right) over TPC-C workload for the B+-Tree indexes deployed on Optane DCPM.

lying NVM hardware.

- Derive correlations among the performance, energy consumption and memory accesses and derive key observations.

The experiments were conducted on a high-end server consisting of two 20-core Intel Xeon Gold 5218R CPUs @ 2.10 GHz, with 4x32GB DDR4 DIMMs and 6x256GB Optane DC NVDIMMs. Intel Optane DCPM was configured in App-Direct mode with the EXT4-DAX filesystem. Version 1.11 of the Persistent Memory Development Kit (PMDK) [140] and gcc-13.3 were employed for the deployment of the applications on the DRAM/Optane DCPM system. Even though Intel has discontinued Optane DCPM, new NVM technologies are anticipated to enter the market in the future [268]. Therefore, the main principles of the proposed methodology can be applied to heterogeneous systems composed of technologies that trade performance for energy efficiency, including standards like CXL [269].

We utilize i) the TPC-C [270] benchmark, the industry standard for evaluating transactional workloads where multiple users execute transactions that involve the maintenance of a large database and ii) the Yahoo! Cloud Serving Benchmark (YCSB) [2], a well-known framework for triggering different types of workloads designed for operating over cloud-based key-value stores. The examined workloads mimic real application patterns, similar to existing researches [42, 271]. More specifically, we generate a set of workloads: i) update heavy (A), ii) read heavy (B), iii) read-only (C), and iv) short-range heavy (E), which have access patterns common to OLTP workloads. We have excluded the workloads D and F as they involve updates and some indexes do not support key updates by design. For instance, in `Fast&Fair` the key-value stores use insert for both insertion and update operations, thus the workloads that include updates have not been exam-

ined. TPC-C and YCSB serve as strong representatives for the database domains by respectively evaluating the transactional capabilities of relational databases in complex online transaction processing scenarios and assessing the performance of key-value and NoSQL databases under cloud-serving workloads. Table 6.5 illustrates the YCSB workload specifications in terms of operation ratios. For our experiments, we utilize 8-byte integer and 24-byte string keys. Last, we present a use-case integration of YCSB workloads to the LevelDB[3]. LevelDB is a no-SQL database, developed by Google, and stores keys and values as arbitrary byte arrays, allowing any type of data to be sorted by a unique key.

6.2.6. Experimental Evaluation

Performance Evaluation: Initially, we evaluate the examined B+-tree indexes regarding their performance and scalability, both in terms of load phase and execution phase. Fig 6.14 illustrates the scaling of load time (left) and run throughput (right) for the alternative B+-tree indexes over the TPC-C benchmark for a rising number of warehouses. First, we observe that as the number of warehouses rises, the load time is also increasing, while the execution throughput drops. The `Fast&Fair` index provides the fastest load time, by requiring down to 43.9% less time, compared to the other indexes. However, during the run phase, in contrast to loading phase, the `P-WBTree` provides higher throughput compared to the rest of the indexes. For instance, the `P-WBTree` is 1.82× faster compared to `Fast&Fair` on average during the run phase. During the load phase, the focus is on efficiently inserting a volume of data into the database, while the execution phase requires query and update operations to be optimized for quick access to specific

[3]LevelDB: https://github.com/google/leveldb

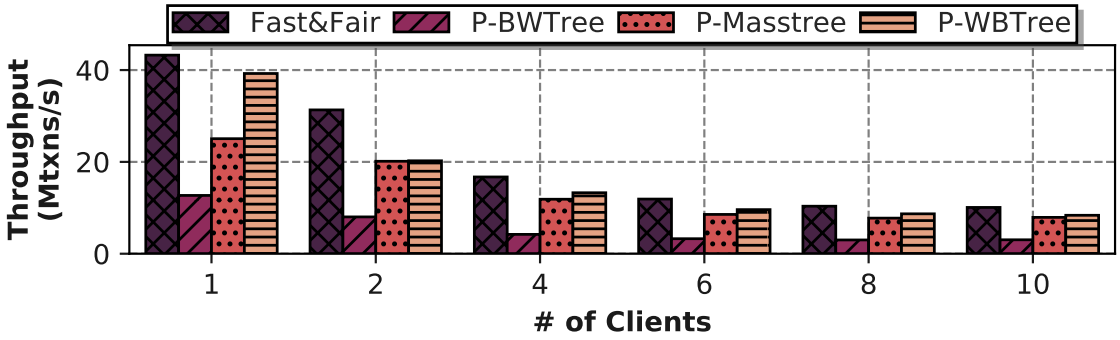


Figure 6.15.: Execution throughput for the TPC-C workload for the examined B+-tree indexes deployed on Optane DCPM.

records, thus `Fast&Fair` operates more efficiently on data insertion, while `P-WBTree` performs better on queries and update operations. Moreover, `P-BWTree` is not providing any performance degradation as the number of warehouses increases, however, it performs poorly compared to the other indexes.

$$q_{\text{avg}} = 0.45 \times q_{\text{new-order}} + 0.43 \times q_{\text{payment}} + 0.04 \times q_{\text{order-status}} + 0.04 \times q_{\text{delivery}} + 0.04 \times q_{\text{stock-level}} \tag{6.6}$$

$$q_{\text{lat}} = \frac{\text{time}}{q_{\text{avg}}} \tag{6.7}$$

Aiming to further analyze the the impact of the examined B+-tree indexes, we evaluate the impact on the query latency on the TPC-C benchmark workload. The query latency is derived based on the model presented in Eq. 6.6, 6.7. The variables $q_{\text{new-order}}$, q_{payment}, $q_{\text{order-status}}$, q_{delivery}, and $q_{\text{stock-level}}$ represent the number of queries involved in each of the respective transaction types in the TPC-C benchmark [270]. The coefficients 0.45, 0.43, 0.04, 0.04, and 0.04 are derived from the typical transaction mix in the TPC-C workload, reflecting the relative frequency of each transaction type: 45% for New-Order, 43% for Payment, and 4% each for Order-Status, Delivery, and Stock-Level. The *time* variable denotes the corresponding total time executed for the TPC-C workload. The violin plots of Figure 6.16 illustrate the average query latency for the TPC-C, as well as the distribution for multiple executions. X axis indicates the corresponding B+-tree index, while Y axis shows the query latency (ms per query).The `P-WBTree` provides the lower average query latency, by being 1.62×, 2.11× and 7.23× faster compare to `Fast&Fair`, `P-Masstree` and `P-BWTree`, respectively. Evan though `P-WBTree` provides lower query latency, compared to `Fast&Fair`, it does not achieve the highest throughput (Fig. 6.15), due to its limitations in multi-threading and concurrency.

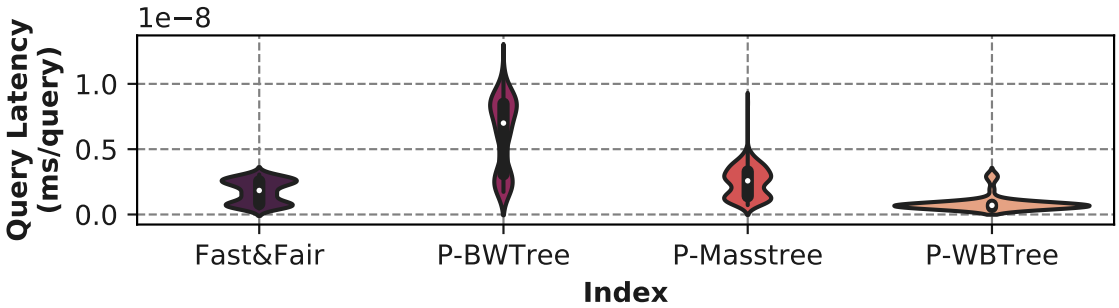
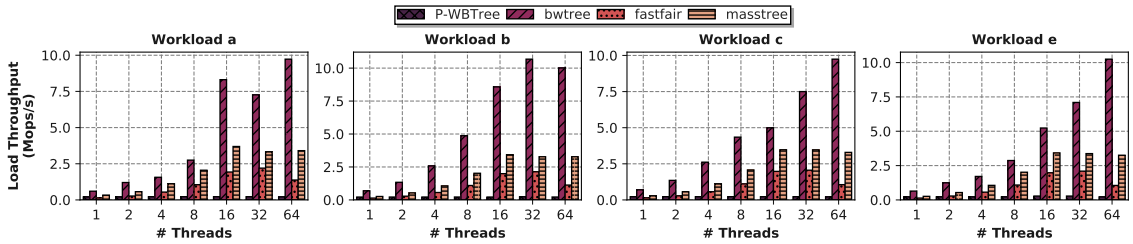
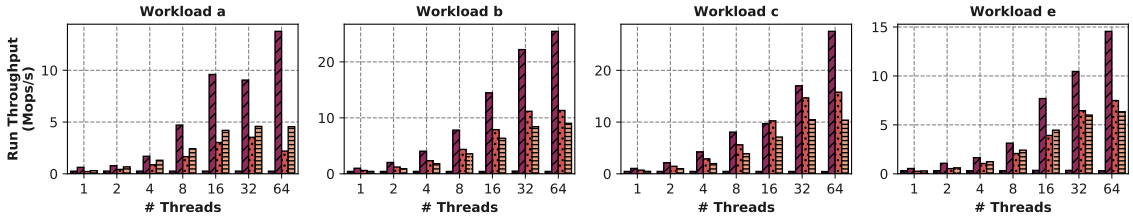


Figure 6.16.: TPC-C average query latency



(a) Load Throughput of B+-tree indexes for YCSB workloads



(b) Run Throughput of B+-tree indexes for YCSB workloads

Figure 6.17.: Load (top) and run (bottom) for the B+-tree indexes over the YCSB workloads deployed on Intel Optane DCPM.

Furthermore, we evaluate the ability of the B+-tree indexes to operate efficiently over a rising number of clients. Figure 6.15 illustrates the execution throughput over a rising number of clients. X axis shows the number of clients, while the Y axis indicates the run throughput achieved. First, we observe that the run throughput drops as the number of clients rises, i.e. as the number of parallel requests increases. Moreover, the `Fast&Fair` index outperforms the other indexes, by achieving higher throughput ranging between 9.2% and 74.7%. An important notice at this point is that `P-Masstree` scales better in terms of input size (num. of warehouses), while `Fast&Fair` outperforms the rest of the indexes as the number of clients increases. Thus, selecting the optimal B+-tree in terms of performance is strongly dependent on the input configurations and, based on the input workload size and the input users, different indexes should be utilized.

Additionally, we evaluate the examined benchmarks in terms of performance over the YCSB workload 6.17. Figure 6.17a illustrates the corresponding load throughput measured in Mega-Operations(Mops) per second of the various B+-tree indexes for the examined workloads. Similarly, Fig. 6.17b shows the run throughput. X axis in both plots shows the number of threads utilized for the workload deployment on the Intel Optane DCPM. Both in load and run phase, we observe that the `P-BWTree` achieves the highest throughput among the examined indexes, by achieving up to 71.2% highest load throughput compared to `P-Masstree` and up to 56.8% highest run throughput compared to `Fast&Fair`. The examined indexes keep scaling for up to 64 threads, except `P-WBTree`,

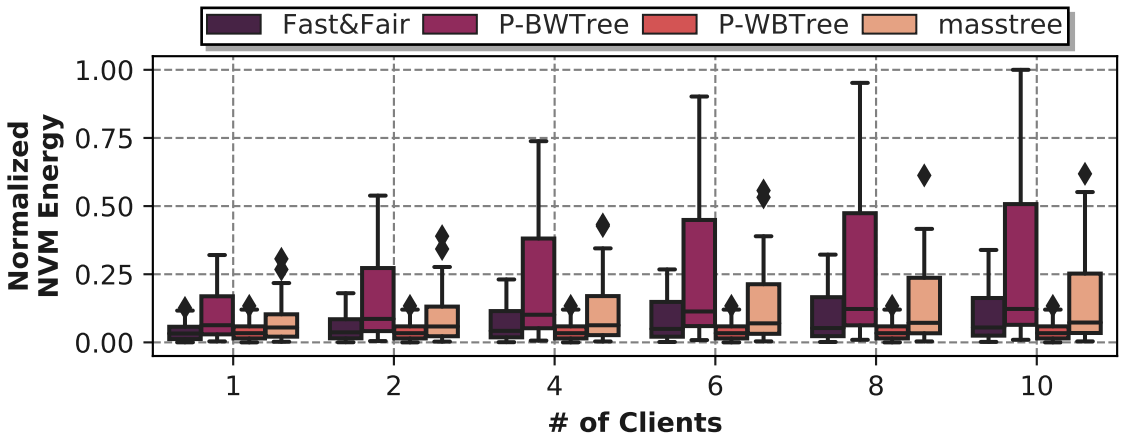


Figure 6.18.: Normalized NVM energy for the TPC-C workload for the examined B+-tree indexes deployed on Intel Optane DCPM.

which is initially designed for single-thread execution. A significant notice is the comparative behavior of `P-Masstree` and `Fast&Fair`. We observe that the former provides higher load throughput, while the latter scales better and provides higher throughput in the execution phase. This is due to the fact that the `P-Masstree` maintains unsorted nodes, thus avoiding the overhead imposed by sorting during the execution phase. Such kind of trade-offs can pave the way for utilizing different B+-tree indexes for database construction and query execution. Similar observations can be derived for the other indexes. For in-between workload configurations, e.g. workload a and b, the design space is significantly pruned between the optimal solutions, therefore the user should only investigate between the corresponding optimal indexes.

Energy Consumption Analysis: NVM technologies have been introduced, aiming to enhance sustainability by offering more energy-efficient and durable solutions for data storage and retrieval. Thus, we evaluate each individual B+-tree index based on their ability to support energy efficiency. Fig. 6.18 and 6.19 illustrate the normalized NVM energy consumption (Y axes) consumed by each B+-tree index deployed on the target NVM technology, i.e. on Intel Optane DCPM for various number of clients (Fig. 6.18) and Kilo-transactions (Fig. 6.19) for the TPC-C benchmark workload, maintaining the input workload constant. Since energy consumption experiments are sensitive to interference effects, we executed each discrete configuration 10 times, in order to accurately evaluate the energy consumption. Focusing on Fig. 6.18, our initial outcome relies on the observation that the average energy consumption is not strongly affected by the number of clients operating in parallel on the same data. However, as shown in Fig. 6.19, where the NVM energy consumption is evaluated over a rising number of transactions (X axis), the `P-WBTree` provides the optimal energy consumption, by achieving down to 37.9% less energy con-

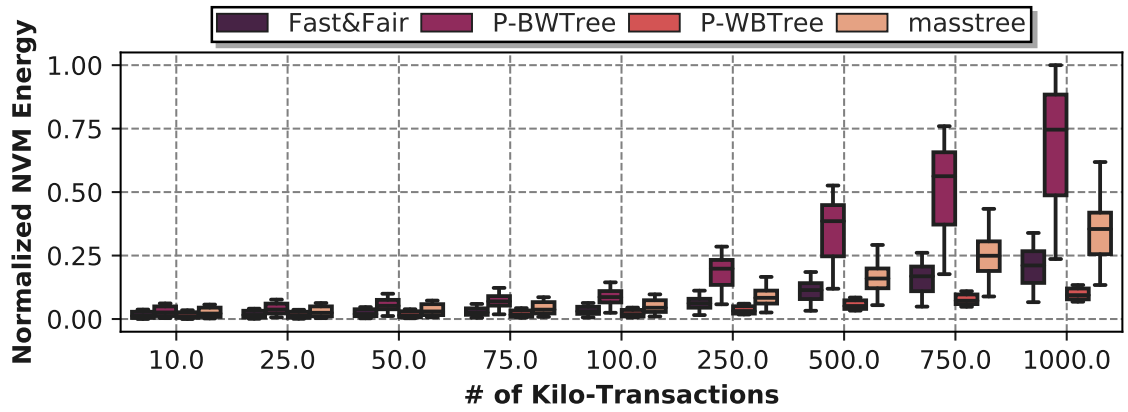


Figure 6.19.: Normalized NVM energy consumption for the TPC-C workload for the examined B+-tree indexes deployed on Intel Optane DCPM.

sumption compared to the other indexes on average.

Aiming to provide more insights on energy consumption, we monitor the energy consumption of the B+-tree indexes for the YCSB workload alternatives, as depicted in Fig. 6.20. First, the `P-WBTree`, which supports only single-threaded execution, consumes the higher energy consumption, as its execution throughput is very low, as highlighted previously in Fig. 6.17b. The `P-BWTree` provides the minimum energy consumption across all the examined workloads. More specifically, `P-BWTree` achieves 76.4%, 41.2% and 33.1% less energy consumption on average compared to `P-WBTree`, `Fast&Fair` and `P-Masstree`, respectively. The `P-BWTree` supports both non-blocking read and write operations, in contrast to `Fast&Fair` where blocking writes are utilized and `P-Masstree`, where write exclusion is selected for concurrency. This leads to an increased number of transactions required from and to the NVM, and mostly energy-expensive write operations, thus increasing the energy requirements.

NVM Accesses Analysis: Minimizing NVM accesses is crucial for optimizing energy consumption and extending the lifespan of non-volatile memory devices, with write accesses being the most dominant factor. Reduced access frequency not only enhances overall system efficiency but also contributes to sustainability by lowering power requirements and minimizing the environmental impact of frequent data transfers. Towards this direction, we evaluate the examined B+-tree indexes, based on their number of NVM accesses. Fig. 6.21 illustrates the number of read (top) and write (bottom) accesses on the NVM accesses over the TPC-C benchmark. X axis shows the number of Kilo-Transactions, while Y axis indicates the number of clients for a constant number of input warehouses. First, we observe that the number of both read and write operations is pro-

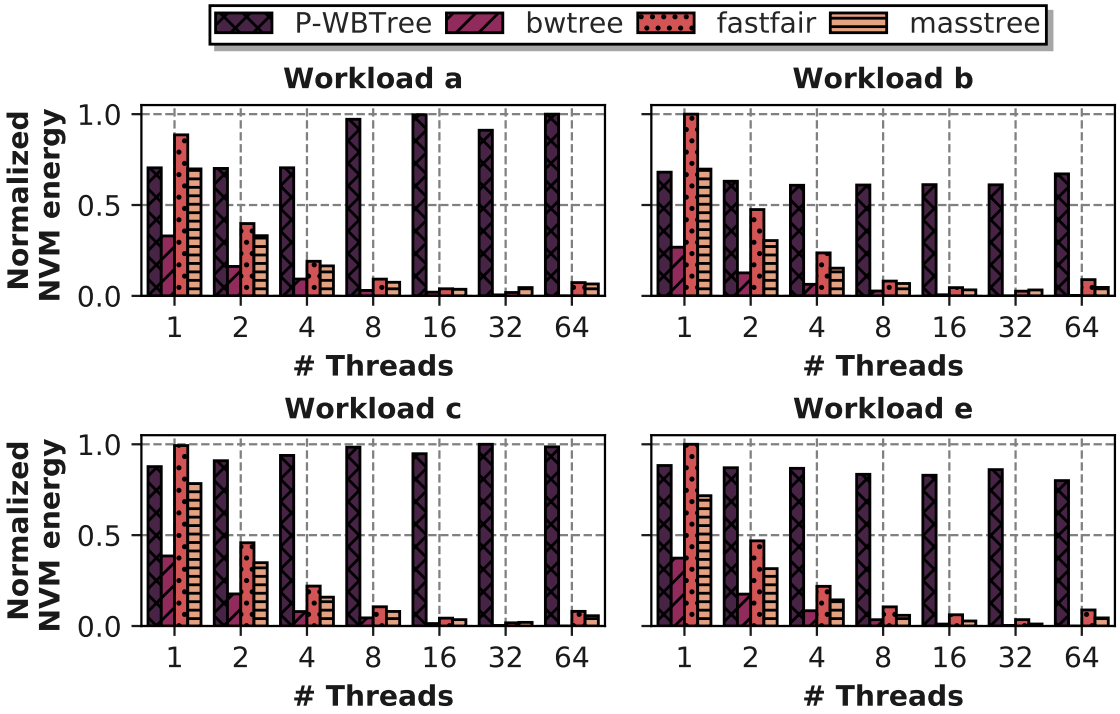


Figure 6.20.: Normalized NVM energy consumption for the YCSB workload for the examined B+-tree indexes deployed on Intel Optane DCPM under different workloads.

portional to the number of the required transactions and the number of clients utilizing the TPC-C database. Moreover, the comparative analysis of write accesses for the different indexes indicates that the `Fast&Fair` requires fewer write accesses compared to the other approaches. For instance, focusing on 1000 Kilo-Transactions the number of write accesses is minimized for different numbers of clients, by requiring up to 51.3%, 70.3% and 41.8% less NVM write accesses, compared to `P-Masstree`, `P-WBTree` and `P-BWTree`, respectively.

Similarly, Fig. 6.22 illustrates the number of read (top) and write (bottom) accesses of the examined B+-tree indexes for the different YCSB workloads and different numbers of threads. In contrast to the TPC-C workload the `P-Masstree` minimizes the number of write accesses for the workloads b,c and e, by requiring 34.8% less write accesses on average, compared to the other indexes. For workload a, the `P-Masstree` and `P-WBTree` provide nearby number of writes. The `P-WBTree` index is not designed to support parallelism, thus, parallel requests in the Intel Optane DCPM due to threading concurrency and locking are avoided, thus leading to a reduced number of accesses, however, performance is traded.

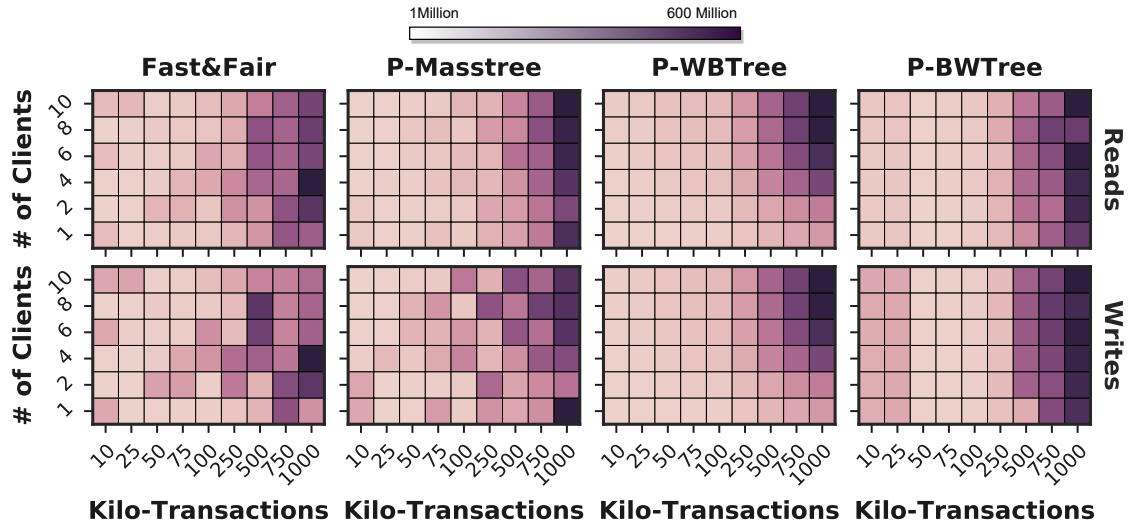


Figure 6.21.: Number of read (top) and write (bottom) access of Intel Optane DCPM DIMMs for the TPC-C workload for the examined B+-tree indexes.

Trade-off and Pareto Optimal Solutions: Following the experiments shown in Sec. 6.2.6, we observe that Pareto optimal solutions can be derived considering i) performance, ii) energy consumption, and iii) NVM write accesses. Deriving trade-offs and Pareto optimal solutions is crucial as it allows us to achieve efficient solutions that not only enhance performance but also minimize energy usage and mitigate the wear associated with frequent NVM write accesses. For instance, for the TPC-C workload, the `Fast&Fair` maximizes the performance as the number of clients rises, the `P-WBTree` minimizes the energy consumption, while the `Fast&Fair` minimizes also the number of write accesses. For the YCSB workloads, in contrast to TPC-C, the `P-BWTree` optimizes performance and energy consumption. However, for the workloads b,c and e the `P-Masstree` minimizes the number of NVM accesses. Thus we observe that different indexes enhance different characteristics based on the examined application and workload characteristics. This observation opens the path for the exploitation of techniques that provide fine-tuned B+-tree index selection based on the target application, the input workload and the target optimization objectives.

Impact on the NVM's lifetime: Estimating the impact of write accesses on the NVM lifetime is essential for ensuring the longevity and reliability of NVM devices. We aim to provide critical insights for designing robust systems with optimized data storage strategies and minimizing the risk of premature wear and degradation. We apply the approach described in [265]. More specifically, we analyze the write accesses associated with the TPC-C and YCSB workloads for the investigated B+-tree indexes, aligning them

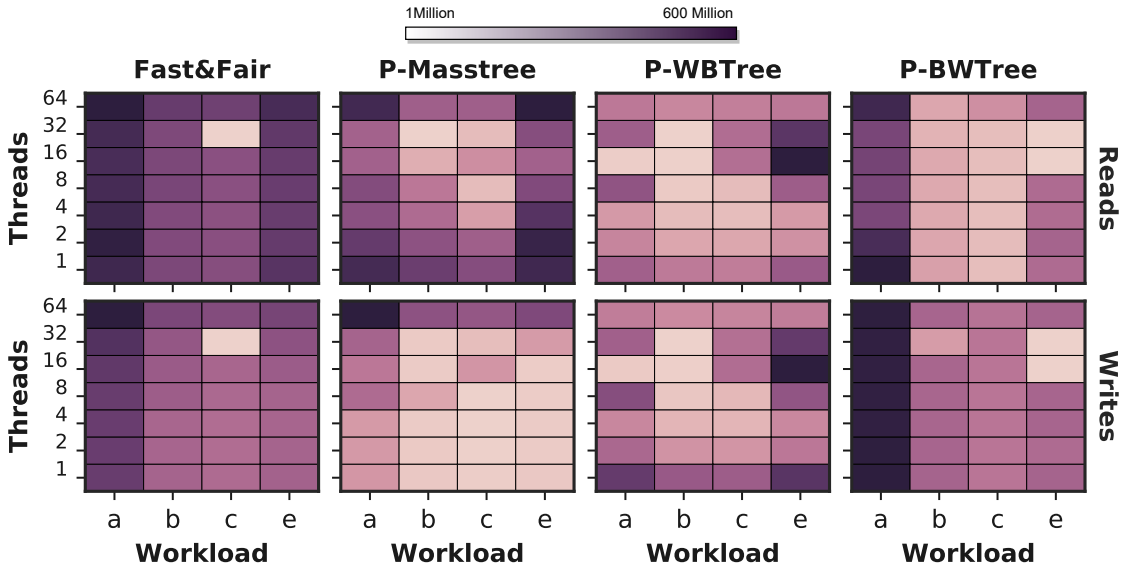


Figure 6.22.: Number of read (top) and write (bottom) access of Intel Optane DCPM DIMMs for the YCSB workload for the examined B+-tree indexes.

with the lifetime-related specifications of the target NVM technology. By evaluating the number of write accesses over the entire runtime and considering the lifetime-related specifications of Optane DCPM (Petabytes Written - PWB) as derived by the manufacturer, we extrapolate a comprehensive estimation of the remaining lifetime of the NVM.

Figure 6.23 illustrates the projected lifetime of Intel Optane DCPM for the examined B+-tree indexes over the different YCSB benchmark workloads. X axis indicates the examined workloads and Y axis shows the normalized Intel Optane DCPM lifetime. Each point corresponds to a different execution configuration of the examined indexes (num. of threads). Our first observation is that each individual index can be selected and mapped to a single workload. For instance, the `P-Masstree` indexes maximizes the lifetime of Intel Optane DCPM when deployed on workload c (read-only). Similarly, the `P-BWTree` maximizes lifetime for the workload b (read-heavy), while `P-WBTree` is more efficient over workload d (scan-heavy), in terms of NVM lifetime. Thus, the *Decision Support* step presented in Section 6.2.4 provides the corresponding optimal B+-tree index, based on the input workload. Furthermore, the effective configuration of the execution strongly affects the lifetime, as it has a significant impact on the effective utilization and accesses on the NVM. For instance, `P-BWTree` lifetime expectancy can range from 0.11 to 1, thus indicating a lifetime expectancy difference up to 89%. Similar outcomes can be derived for the other B+-tree indexes, while similar behavior is expected to be derived for other

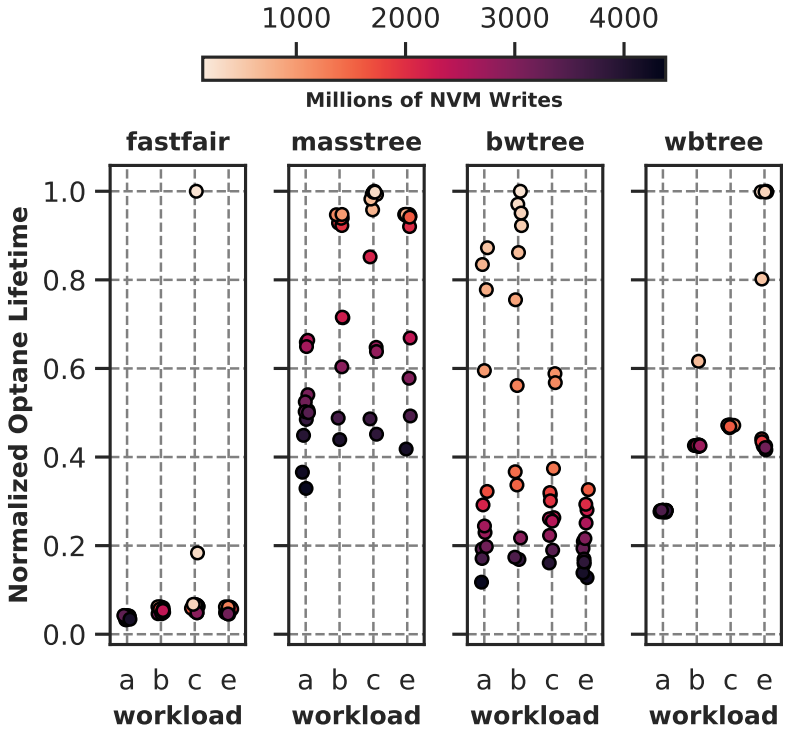
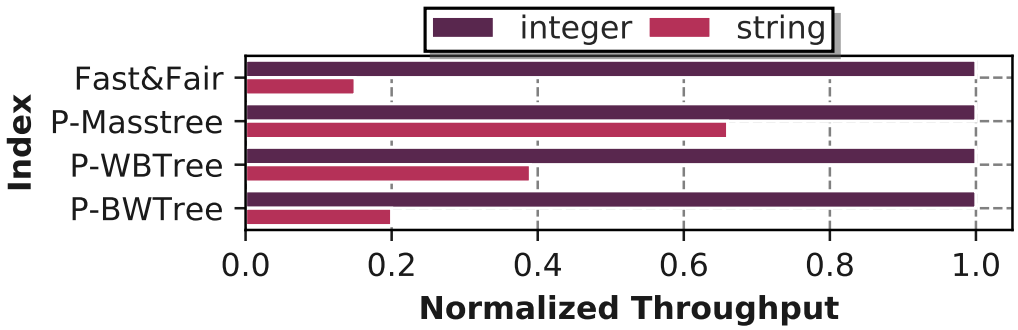


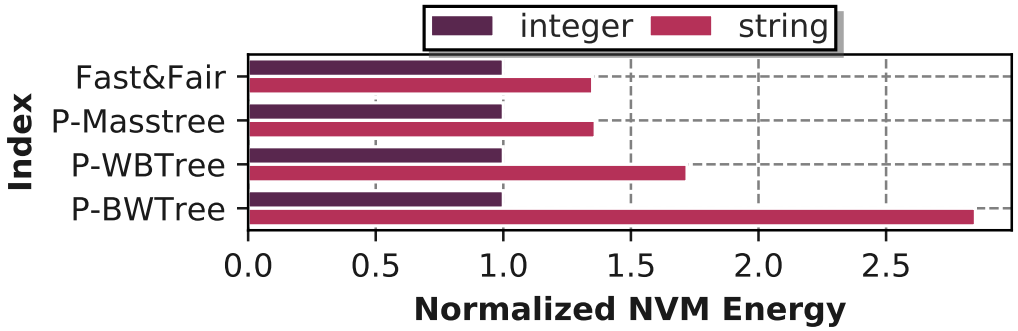
Figure 6.23.: Projected normalized NVM lifetime for the examined B+-tree indexes deployed over the different YCSB workload.

NVM technologies.

Impact of integer and string keys: The selection of integer (8-byte) and string (24-byte) keys in key-value indexes and databases integrates extra trade-offs. Aiming to investigate the impact of key size on performance and energy consumption, we conduct a comparative experiment over the YCSB benchmark. Figure 6.24 shows the impact of the key selection on the execution throughput and the energy consumption on the Intel Optane DCPM. Figure 6.24a illustrates the average normalized comparison of the run throughput when utilizing string keys. Even though string keys offer higher storage capabilities, we observe that this is traded on performance. More specifically, the performance slowdown ranges from 1.51× (`P-Masstree`) to 6.6× (`Fast&Fair`), leading on average to 3.85× slowdown. Similarly, Figure 6.24b highlights the average normalized energy comparison of the YCSB benchmark when deployed with integer and string keys. Being inline with the performance degradation when string keys are utilized, the energy consumption increases, as i) higher time is required for processing the string keys and ii) higher memory resources are required for storing string keys. More specifically, the energy consumption increases from 1.35× (`Fast&Fair`) to 2.85× (`P-BWTree`). Therefore,



(a) String vs Integer keys throughput comparison



(b) String vs Integer keys NVM energy comparison

Figure 6.24.: Normalized run throughput and normalized NVM energy comparison of string vs integer keys for the examined B+-tree indexes over the YCSB benchmark workload.

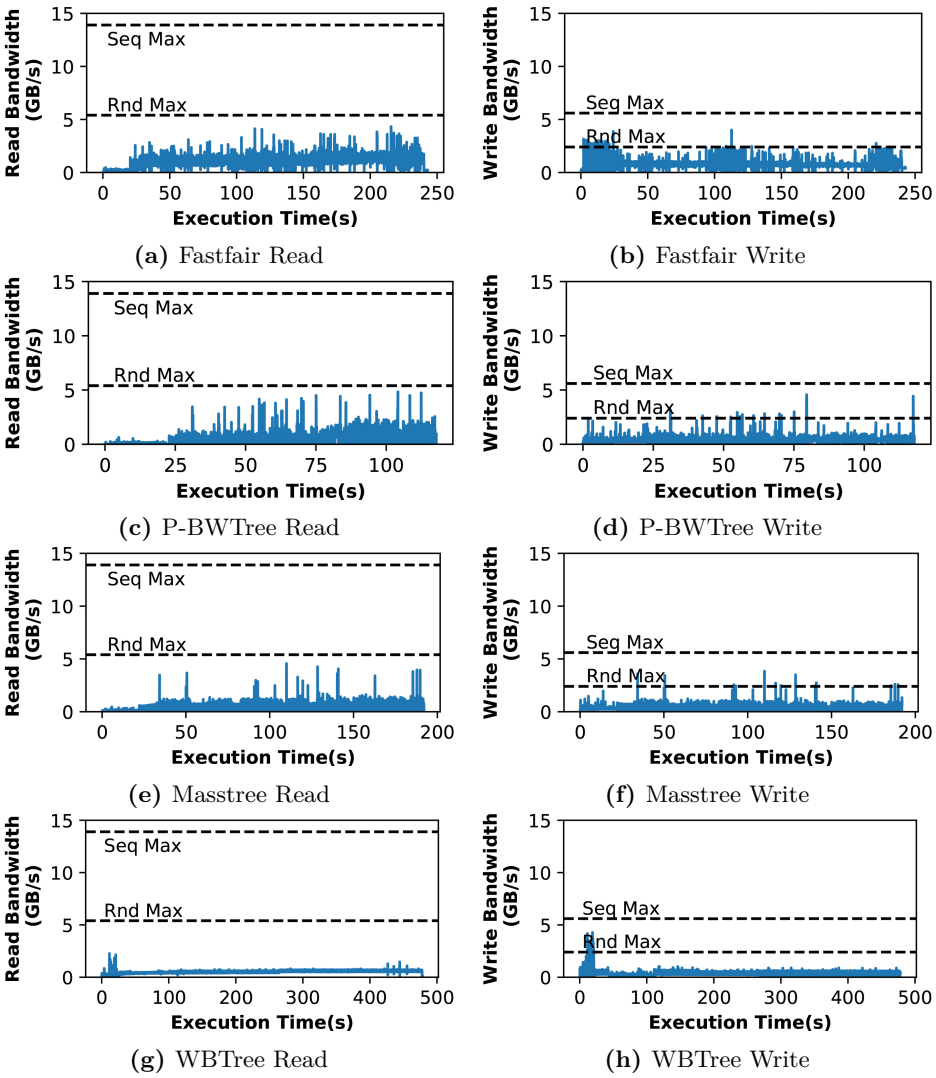


Figure 6.25.: Intel Optane DCPM system's throughput evaluation for the examined B+-tree index implementations over YCSB balanced workload (A) for 64 threads and integer keys.

performance and energy can be traded for higher memory, integrating one more degree of freedom as another aspect of exploration.

Memory Bandwidth Analysis: Moreover, we conduct an experiment on the impact of the discrete indexes on the memory bandwidth utilization, when deployed on 64 threads. The maximum available bandwidth for random and sequential accesses, as shown in each figure, are used as the baseline that can be provided by the underlying system, as measured by the Memory Latency Checker (MLC) tool [272]. The indexes do not saturate Optane DCPM bandwidth. As shown in Fig.6.25a,6.25c,6.25e,6.25g and Fig.6.25b,6.25d,6.25f,6.25h in most of the experiments the memory bandwidth utilized is much lower compared to the available. For instance, the `P-Masstree`, shows 2.8GB/s read bandwidth (Fig. 6.25c), and 1.7GB/s (Fig. 6.25d) write bandwidth on average both for integer and string keys, which is 69.6% lower than the maximum sequential read and write bandwidth and 48% lower than the maximum random read and write bandwidth that the system can provide, respectively. Similar observations can be derived for the rest of the indexes.

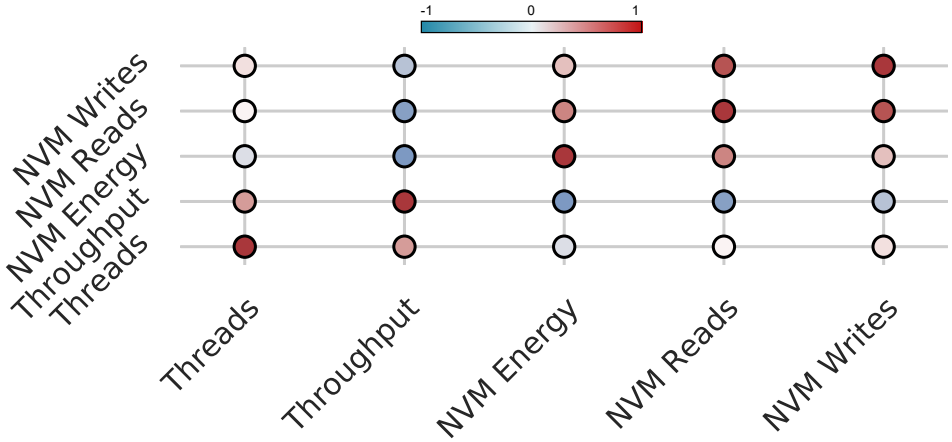


Figure 6.26.: Pearson correlation of high-level metrics for all the examined B+-tree indexes and workloads (-1: perfect negative correlation, 0: No correlation, 1: Perfect positive correlation)

This observation holds true for both read and write bandwidth scenarios. Consequently, even when scaling our workload to utilize the maximum number of threads (i.e., concurrent users), our system demonstrates the capability to run multiple database instances in parallel efficiently. This efficiency is maintained even when interference occurs in the memory bus, suggesting that our system's design effectively handles memory bandwidth demands and prevents bottleneck scenarios. By ensuring that the memory bandwidth is not a limiting factor, we enable our system to accommodate high levels of concurrency

```
// Include LevelDB definitions
#include "leveldb/db.h"
...

enum {OP_INSERT, OP_UPDATE, OP_READ, OP_SCAN, OP_DELETE}; //Operations
enum {DEFAULT, P-WBTREE, BWTREE, FASTFAIR, MASSTREE}; //Trees
enum {INT_KEY, STRING_KEY}; // Key Types
enum {UNIFORM, ZIPFIAN}; // Key Distributions

void main()
{
    initialize_LevelDB(); // Initialize Database
    configure_LevelDB(); // Keys,Distribution,...
    operations = import_YCSB_workload(); // Workload

    for( operation in operations )
    {
        ...
       if (operation = OP_INSERT)
            index_insert(data) // Insert new data
        else if (operation = OP_SCAN)
            index_scan(key, range) // Scan range of keys
        else if (operation = OP_UPDATE)
            index_update(key, data) // Update values
        else if (operation = OP_READ)
            index_read(key) // Read data of key
        else if (operation = op_DELETE)
            index_delete(key, data) // Delete data
        ...
    }

    destroy_LevelDB(); // Destroy the database
}
```

Listing 6.1: YCSB Integration on LevelDB Code Snippet

and maintain optimal performance across various workload intensities. This analysis highlights the robustness and scalability of our system architecture in leveraging Optane DCPM for high-performance database operations.

Correlation of high-level metrics: Last but not least, aiming to provide a thorough investigation of the impact of the B+-tree indexes on the examined metrics, we investigate the correlation among the number of operating threads, performance, energy, read and write accesses for the examined B+-tree indexes. Towards this direction, we examine the Pearson correlation [273]. of the aforementioned metrics, as depicted in Fig 6.26. Our major observations are the following: i) the overall throughput is mostly affected by the number of operating threads, ii) the NVM energy and accesses are not strongly affected by the number of operating threads, iii) the NVM energy consumption is strongly related to the number of write accesses and iv) the energy consumption on the NVM is inversely proportional to the achieved throughput. Thus, the effective throughput optimization can boost the reduction of energy consumption, however the number of accesses should be treated as an independent optimization objective.

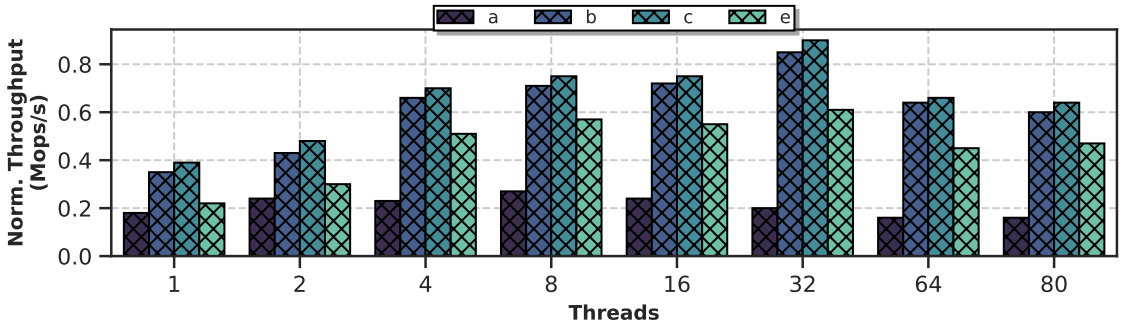


Figure 6.27.: LevelDB throughput for different YCSB workloads

6.2.7. Use-Case Integration of YCSB to LevelDB Database

Last, we integrate the YCSB workload to a real database system, namely LevelDB key-value store Through our prior experiments have illustrated that the `BW-Tree` provides the optimal run throughput, as analyzed previously in Fig. 6.17b. In this analysis, we utilize the default LevelDB's log-structured merge-tree (LSM tree) for storing data, in order to evaluate also the impact of the LSM tree. The major advantage of the LSM trees is that different parts of the tree can be configured differently for the platform they reside. Listing 6.1 presents a code snippet for integrating YCSB workloads to the LevelDB database. First, the LevelDB database is initialized and configured, regarding the deployed index, the key type and the distribution of them. Then, the YCSB workload imports of operations, and then performs each operation (insert, scan, update, read, delete) as specified by the workload. Figure 6.27 illustrates the normalized achieved throughput for the discrete YCSB workloads deployed on Optane DCPM. Workload c, which is read-only achieves the highest throughput, as degradation due to slow write operations on NVM technologies, is minimized.

6.2.8. Conclusion

This section presents a novel methodology for persistent B+-tree indexes exploration for database workloads deployed over heterogeneous DRAM/NVM systems. It investigates the impact of B+-tree indexes deployed on NVM technologies in terms of execution time, energy consumption and impact on the NVM lifetime. Our methodology can be utilized by database developers and vendors as an approach for co-design between database application indexing structures and heterogeneous memory systems.

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

6.3.1. 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. For instance, several application domains across the computing continuum are enabled by Machine Learning (ML) algorithms [274] and rely on databases for performing transactions based on complex queries, or process streams of data generated in real-time at the edges of IoT networks [275]. 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 real-time performance requirements and address energy consumption constraints, by putting severe pressure on the main memory subsystem.

Traditional systems consist of DRAM-based main memories. However, DRAM technologies have hit their physical limits over the last years [261], as they exhibit limitations both in terms of scalability and sustainability. The integration of multiple DRAM DIMMs to enable higher performance and satisfy increased memory 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. Existing literature indicates that a typical DRAM main memory subsystem consumes up to 50% of the energy of a typical HPC system [253].

Aiming to tackle these limitations, emerging memory technologies are investigated, including Non-Volatile Memories (NVM). NVM technologies, such as the 3D-XPoint [24], which is a subclass of Phase-Change Memory [276], Spin-Transfer Torque RAM [277] and Resistive RAM [25] are products of an active research area, involving both academic and industrial communities. The Intel Optane DPCM (Data Center Persistent Memory) is the 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 [135] and the Alibaba Cloud [262], 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 [278].

NVM technologies provide higher density compared to DRAM, thus leading to increased memory capacities, at a reduced cost. Moreover, the integration of NVM technologies has a significant impact on energy consumption and sustainability [116], while providing data persistence features, enabling the development of algorithms for fault tolerance support. Despite the undoubted benefits of the rapidly growing NVM technologies, their constraints impose significant challenges for application developers. NVMs provide increased access latency, low bandwidth and reduced endurance compared to DRAM technologies. Moreover, NVMs provide asymmetry on the latency and energy of read/write operations [154], while directly replacing DRAM with NVMs, such as the Intel Optane DCPM significantly reduces performance, due to effects like the write throttling and concurrency contention [160]. Therefore, NVM DIMMs are typically paired with DRAM modules, composing a heterogeneous DRAM/NVM system.

Towards the efficient exploitation of the DRAM/NVM heterogeneous memory systems, several data placement algorithms have been proposed over the years [189, 279]. These algorithms typically recommend or perform an efficient placement and/or runtime migration of data objects, memory pages, or application data structures over a heterogeneous memory system 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. In particular, developers do not normally consider any data optimization 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 [154, 280]. Also, even though application-level data optimization methodologies have been proposed several years ago [144, 281], their interplay and their impact on the data placement over heterogeneous memories has not been investigated, yet. However, improving the data organization at the application-level can result in significant improvements in performance and energy consumption.

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 combines both a dynamic data structure refinement step, as well as an algorithm for placement over heterogeneous memory systems. The data structure refinement methodology (DDTR)

optimizes 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 [144]. In the context of this work, the DDTR methodology is extended with an expanded design space, including a set of tree-based data structures that, due to their efficient insertion and searching methods, are strong candidates for storing and retrieving data to/from increased-capacity heterogeneous memory systems. Furthermore, an optimized in-memory profiling approach avoids the increased profiling overhead and reduces the design space exploration time. Subsequently, a data placement algorithm provides a set of Pareto optimal data structure placement solutions for heterogeneous DRAM/NVM systems in terms of performance, energy consumption and impact on the NVM's lifetime. 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 that automates many steps of the methodology. It targets developers, who aim to deploy applications on computing systems that integrate Optane DCPM (or other types of NVM DIMMs), or they aim to investigate 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 section is organized as follows. Subsection 6.3.2 illustrates the fundamental background and the motivation. In Subsection 6.3.3 the proposed methodology and implementation are described in detail. Subsection 6.3.7 shows the experimental evaluation and the discussion of the main observations. Finally, Subsection 6.3.8 concludes this work.

6.3.2. Background & Motivation

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

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 [154]. 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, thus it can be directly accessed by the application under execution.

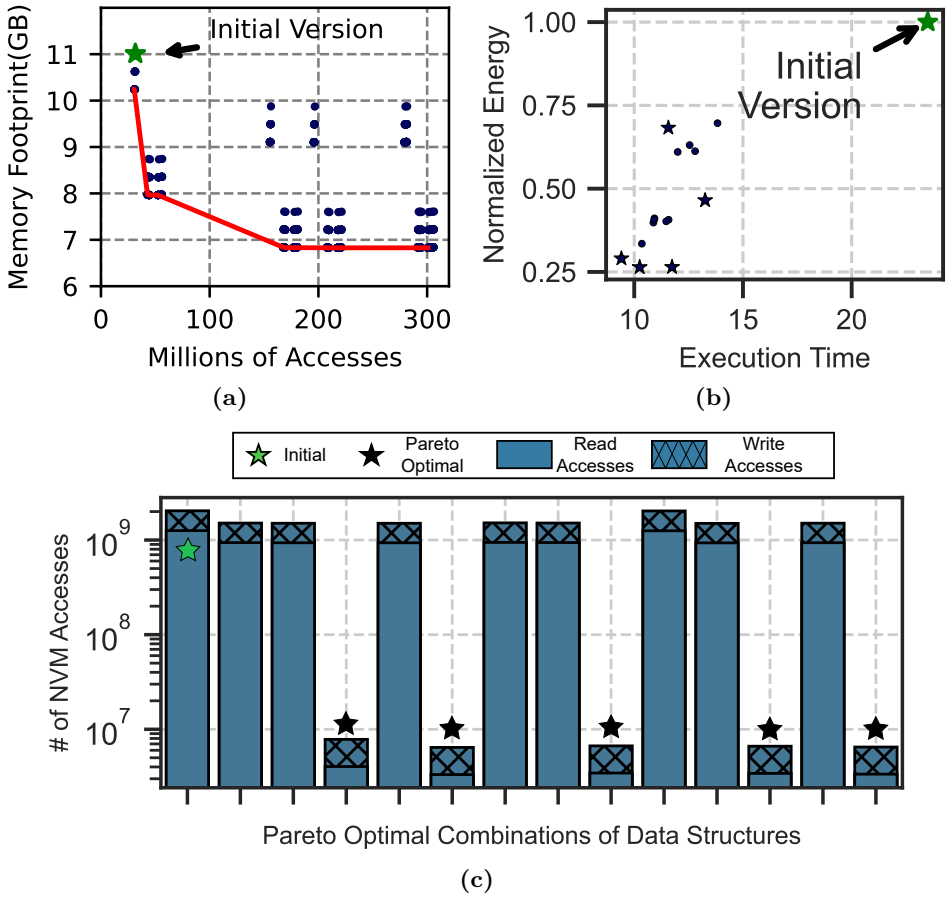


Figure 6.28.: Investigating the impact of an application-level optimization for improved data organization (DDTR) on the results of data placement, in terms of execution time, energy consumption and impact on NVM lifetime. (a) The output of DDTR: application versions, each one having a different data organization (b) The execution time and energy consumption after placement and execution on a DRAM/Optane DCPM system for each version (c) Number of physical read/write accesses for each application version.

Motivation Aiming to investigate the impact of application-level methodologies for data organization improvement on the results of the data placement algorithms on heterogeneous DRAM/NVM systems, we conducted the following experiment: We developed a synthetic workload consisting of four data structures. Each data structure is accessed either in a sequential or random data access pattern, thus leading to different memory access behavior. The data structures were initially implemented as Vectors (VEC). We applied the Dynamic Data Type Refinement methodology (DDTR), which explores the design

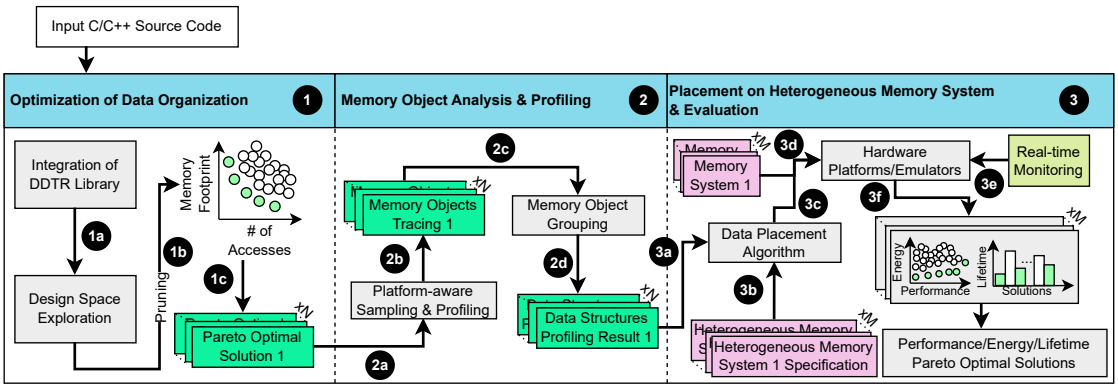


Figure 6.29.: Overview of the proposed methodology

space of data structure implementations and changes the original implementations with ones that provide better results in terms of data accesses and memory footprint. In particular, the output of the methodology is a set of Pareto optimal data structure implementations in terms of data accesses and memory footprint, in which each point corresponds to a different combination of data structure implementations. A version of the DDTR methodology targeting DRAM-only systems can be found in [144].

Fig. 6.28a shows the output of the DDTR methodology for the synthetic application. Each point of the Pareto space corresponds to a version of the application with the same functionality but with different combination of data structure implementations. It can be noticed that developers can trade the required memory capacity for lower number of data accesses (and vice versa), by changing the data structure implementations of the application under optimization.

As a next step, we deployed the Pareto optimal versions of the application on a heterogeneous DRAM/NVM system, using a state-of-the-art data placement algorithm [189]. The performance and energy consumption results are shown in Fig. 6.28b. In other words, Fig. 6.28b illustrates the mapping of the Pareto optimal points of Fig. 6.28a on the performance vs. energy space, after the placement and execution on a DRAM/Optane DCPM system. The X axis is the execution time and the Y axis is the normalized energy consumed by the DRAM and the Intel Optane DCPM DIMMs. We notice (i) that the DDTR step enabled improved performance and energy consumption compared to the initial version of the application (up to 66.3% lower execution time and up to 74.1% lower energy) and (ii) that trade-offs can be obtained between the execution time and the energy consumption by selecting different application versions.

The placement of data on heterogeneous memory systems can have a significant impact

on the NVM's wear. The more frequently a particular block of memory is written to, the faster it will degrade and wear out, thus leading to data loss and reduced device lifespan. Therefore, the number of accesses on the NVDIMMs can be considered a high-level indicator for estimating the impact of the overlying application in the NVM's lifetime [265]. Fig. 6.28c depicts the number of read and write memory accesses on the Intel Optane DCPM DIMMs for the Pareto optimal solutions identified by the DDTR step. It can be observed that the data organization of the application significantly affects the number of physical read and write accesses. Through the effective selection of a combination of data structure implementations, the number of NVM write operations can be reduced up to 2.23× orders of magnitude compared to the initial version of the application. The data structure combinations represented with a star (∗) are the Pareto optimal solutions in terms of execution time, energy consumption and number of write accesses in NVM (five solutions in total).

Based on the aforementioned observations, it is shown that an application optimization and placement methodology which will integrate application-level optimizations of data organization will enable improved data placement results. In the context of this work, the placement results are evaluated based on a multi-dimensional design space, including: (i) application execution time, (ii) energy consumed at the DRAM and NVM DIMMs and (iii) impact on the lifetime of the NVM DIMMs.

6.3.3. Proposed Methodology

The proposed methodology has been designed to meet three main objectives: (i) demonstrate improved results for data placement algorithms, enabled by application-level optimizations. (ii) evaluate placement solutions based on multiple objectives (performance, energy consumption, impact on NVM lifetime). (iii) 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 or emulated NVM technologies).

Fig. 6.29 shows an overview of the proposed methodology. The input of the methodology is the source code of the target application. It consists of three steps:

1. The *optimization of data organization* ❶, which leverages design space exploration for performing source-to-source data-flow transformations at application-level that improve the data organization. The main objective of this step is the efficient selection of the application's data structure implementations, for minimizing the number of data accesses and the memory footprint. This step generates a set

Table 6.6.: The design space of data structure implementations in the DDTR 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

of Pareto optimal solutions, each one corresponding to a different combination of data structure implementations for the application under optimization. A thorough analysis of this step is provided in Section 6.3.4

2. The *memory object analysis and profiling* ❷, which, based on memory-trace and profiling tools, analyzes the Pareto optimal versions of the application provided by the previous step. Profiling information is generated, processed and forwarded to the data placement algorithms for heterogeneous memories. This step is detailed in Section 6.3.5.

3. The *placement on heterogeneous memory systems and evaluation* ❸, in which data placement decisions for each application version are made, based on the profiling information generated. The application versions are placed and executed on a real or emulated heterogeneous memory system. Based on design space exploration, trade-offs are identified, between performance, energy consumption and NVM read/write accesses. A detailed description of this step is provided in Section 6.3.6.

The output of the methodology is a set of Pareto optimal solutions: Each point of the Pareto curve corresponds to a version of the application with a specific data organization, placed on a specific DRAM/NVM system. By selecting different points, developers can perform trade-offs between performance, energy consumption and impact on the NVM's lifetime.

The library for the optimization of the data organization of the application is implemented in the C++ programming language. The framework is developed in Python and in bash scripting. The entire framework is open-source and available online[4].

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

```
#include <Vector>
#include <map>
//Include our library of data implementations
#include <ddtr>

using namespace std;
class SamplePoint {...};

//Native data structure implementation
//vector<SamplePoint> v1;
//Replacing with library data structure
ddtr_vector<SamplePoint> v1;

//Native data structure implementation
//map<int, SamplePoint> mp1;
//Replacing with library data structure
ddtr_map<int, SamplePoint> mp1;

//Defining iteratator
map<int,SamplePoint>::iterator itr;

//Inserting element
mp1.insert(pair<int, SamplePoint>(1,new SamplePoint));
...
```

Listing 6.2: Example code snippet of the integration of the DDTR library to an application

6.3.4. Optimization of Data Organization

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 through frictional contacts	16 VECTORS & 1 MAP

Table 6.7.: Overview of the applications used for the evaluation of the methodology

The first step of the methodology is inspired by approaches such as Brainy [153] and Dynamic Data Type Refinement (DDTR) [144]. These methodologies explore a design space of different data structure implementations, each one having different characteristics in terms of memory footprint and number of data accesses. Lower memory footprint results in lower leakage power in DRAM technology, while reduced memory accesses typically correspond to increased performance and improved NVM life-

time.

In this work, we used the main principles of the Dynamic Data Type Refinement (DDTR) methodology as a starting point and extended it significantly. The initial DDTR was designed for optimizing the data organization for homogeneous DRAM-only systems. Thus, the design space was limited to C++ Standard Template Library (STL) compatible variations of list-based data structure implementations. However, in the context of this work, a set of tree-based implementations were integrated into the library of available data structure implementations. Tree data structures combine structural relationships in data, represent hierarchies and provide efficient insertion and searching with a reduced number of read and write operations, thus making them strong candidates for NVM technologies [160]. In particular, we integrated into the library a Red-Black tree and AVL tree, namely MAP-RB and MAP-AVL, respectively. The complete list of data structure implementations that formulate the design space is shown in Table 6.6.

From a technical perspective, the interface of the library is STL-compatible, to be directly integrated to applications source code which use STL-based data structures. In Listing 6.2 an example code snippet of the integration of the library is illustrated. The original data structures of the application source code are replaced with the ones of the library. Thus, all operations (e.g. add, search, remove) are performed by the DDTR library data structures.

Algorithm 3: Heuristic for pruning of neighbor Pareto Points

```
1  for pX in Pareto Space do
2      for pY in Pareto Space do
3          /*Count Euclidean Distance(ED)*/
4          ED=(pX.footprint - pY.footprint)^2 + (pX.accesses - pY.accesses)^2
5          if ED < Threshold then
6              /*Remove neighbor points*/
7              delete(pY)
```

After inserting the interface of the library in the application's source code, the design space exploration step is performed (1a): the application is executed once for each different combination of the data structure implementations shown in Table 6.6. 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 which 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 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 profiling overhead.

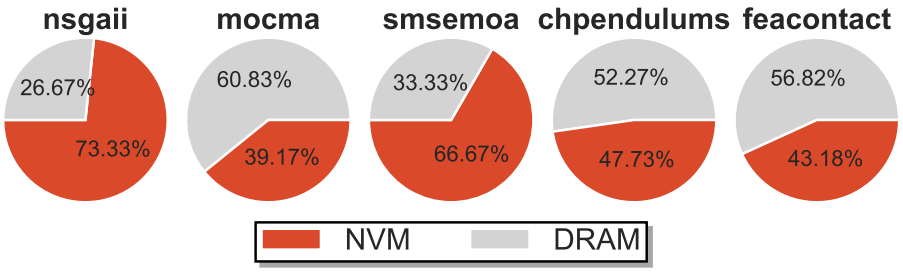
The output of the first step is the Pareto curve of data accesses vs. memory footprint (1b). Each point corresponds to an application version with the same functionality but different combination of data structure implementations. Aiming to further reduce the size of the design space, we implemented a pruning algorithm that removes Pareto points that are too close to each other, thus flagged as neighbor points. 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 h, 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. The pruning heuristic algorithm is shown in Algorithm 3. The final set of Pareto optimal solutions (1c) is provided as input to the *Memory Object Analysis & Profiling* step (2a).

6.3.5. Memory Object Analysis & Profiling

The purpose of the second step of the methodology is to analyze each Pareto point identified by the first step and generate the profiling results required as input by the data placement algorithms.

The profiling results are obtained at the memory object-level, similar to other works in the literature [189, 279]. Through platform-aware sampling and profiling (2b) we collect

Figure 6.30.: Average distribution of the data structures of the Pareto optimal solutions over heterogeneous memory systems for each application.



memory traces per object, including: load and store operations, object size and LLC misses. Other metrics can also be monitored depending on the input requirements of the selected data placement algorithm.

During the DDTR step, each object is assigned an *id*, which indicates the data structure to which it belongs. The *ids* are used to group the memory objects belonging to the same data structure (2c) and accumulate the corresponding profiling results (2d). Objects belonging to the same data structure are grouped and mapped to the data structure to which they belong. Intel's Performance Counter Monitor (PCM) [26] and VTune Profiler [191] are utilized for profiling. An indicative sampling rate is 0.1s. The accumulated profiling results are provided as input to the data placement algorithms of the third step of the proposed methodology (3a).

6.3.6. Placement on Heterogeneous Memory Systems & Evaluation

The data placement algorithms considered by the proposed methodology are the ones that rely on analytical models for decision-making [189, 279]. 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 (3b). This information can be either obtained through manufacturer's reports or found in the existing literature [279]. The memory systems can be either real hardware platforms, such as DRAM/Optane DCPM or emulated for emerging NVM technologies (3d).

Each application version is deployed on the heterogeneous memory system, based on the selected data placement algorithm, and it is executed. To monitor the execution time, the energy consumption and the number of NVM writes, real-time monitoring is required (3e). 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).

Concerning the impact on the NVM's lifetime, we apply the approach described in [265]. More specifically, we map the write accesses for each application's version to the lifetime-related specs of the target NVM technology. 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.

Thus, developers can identify trade-offs between performance, energy consumption and

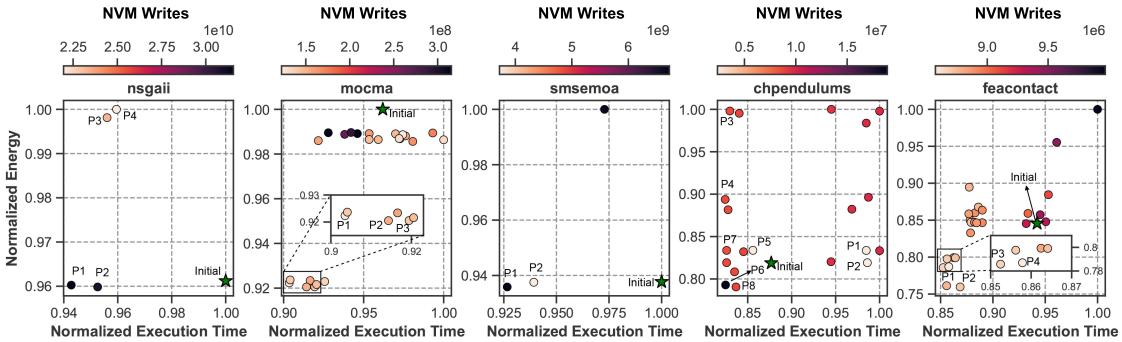


Figure 6.31.: Design space and performance/memory size/number of NVM writes Pareto optimal solutions for each application deployed on a DRAM/Optane DCPM system

the number of NVM write accesses and select the Pareto optimal solutions that meet the design constraints.

6.3.7. Experimental Evaluation

Experimental Setup The proposed methodology was evaluated based on the following criteria:

- The impact of the high-level data organization optimization methodologies, such as DDTR, on the results of data placement, in terms of performance, energy consumption and impact on NVM lifetime.
- 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).
- 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 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) [272] 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) [140] 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 [282] to emulate the PCM, RRAM

	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.				

Table 6.8.: Pareto optimal solutions for each application deployed on a DRAM/Optane DCPM system.

and STT-MRAM technologies. The latency and power specifications of each memory technology are reported in relevant literature [279], while the lifetime-related values for the Optane DCPM were obtained from the corresponding manufacturer's reports [29].

The methodology was evaluated based on applications from two different domains: machine learning and physics-based simulations. Five representative applications were selected from the Shark ML Library [283] and the Chrono Physics engine [284]. Table 6.7 summarizes and briefly describes each application. Concerning the data placement, we implemented two algorithms that operate at memory object granularity and focus on the optimization of performance and energy consumption [189, 279]. The first algorithm relies on a performance and energy consumption model. 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 [279]. Similarly, the second data placement algorithm 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 [189]. This algorithm supports the runtime migration of objects. However, in the context of this work, we apply the initial object placement phase only.

Evaluation We applied the methodology to all applications using the data placement algorithm described in [189]. We first demonstrate the average distribution of the data structures over the heterogeneous memory system (both DRAM/Optane DCPM and emulated) for each application. 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 and trade-offs between performance, energy consumption and impact on NVM lifetime can be identified.

Fig. 6.31 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 to 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 6.8 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 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,

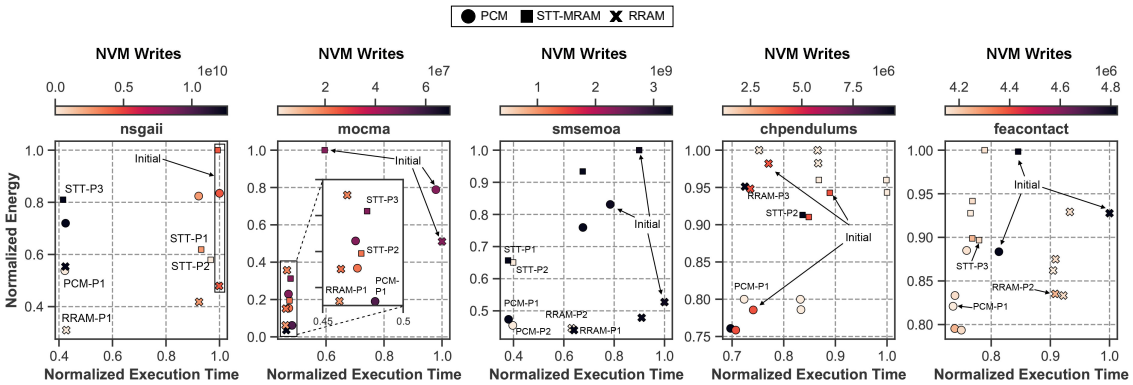


Figure 6.32.: Design space and performance/memory size/number of NVM writes Pareto optimal solutions for each application deployed on a DRAM/PCM, DRAM/RRAM and DRAM/STT-MRAM system.

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 to 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`).

Fig. 6.32 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: PCM, RRAM and STT-MRAM. As in Fig. 6.31, 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 to 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 to the initial version. These results demonstrate the applicability and effectiveness of the methodology on various emerging NVM technologies.

Table 6.9 provides more insights into the impact of the data structure optimization on

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)	

Table 6.9.: Subset of Pareto optimal solutions for each application deployed on a DRAM/PCM, DRAM/RRAM and DRAM/STT-MRAM system.

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 interesting observation is that the same Pareto Optimal combination of data structure implementations may optimize different aspects depending on the underlying memory configuration. 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 6.8 shows that the same combination of data structures for NSGA-II deployed on a DRAM/Optane DCPM system provides optimal results in terms of en-

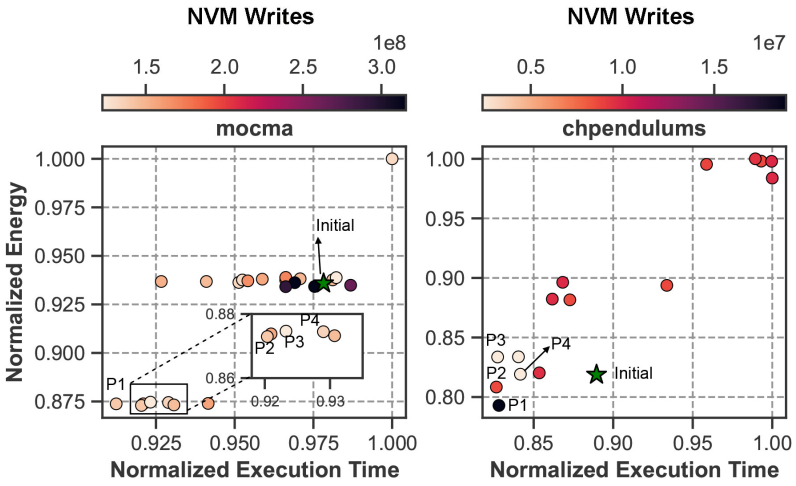


Figure 6.33.: Design space and performance/memory size/number of NVM writes Pareto optimal solutions for the MOCMA and CH-PENDULUMS applications for an alternative data placement algorithm.

ergy 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 6.9) and P1 on DRAM/Optane DCPM (Table 6.8) correspond to the same combination of data structures, with the same placement configuration and provide optimal results in terms of performance in all the aforementioned memory technologies.

To demonstrate the effectiveness of the proposed methodology using alternative data placement algorithms, Fig. 6.33 shows the output of the methodology for the MOCMA and CH-PENDULUMS on DRAM/Optane DCPM, when using the data placement algorithm of [279]. Although this algorithm uses different performance 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.

We investigated the impact of the data organization and placement policies on the underlying NVM's lifetime. We followed a similar approach to [265], where the physical write accesses are projected to the underlying NVM technology, to evaluate the expected lifetime of the Optane DCPM. As stated earlier, the NVM lifetime-related values for the Optane DCPM are obtained from the manufacturer's reported specifications [29],

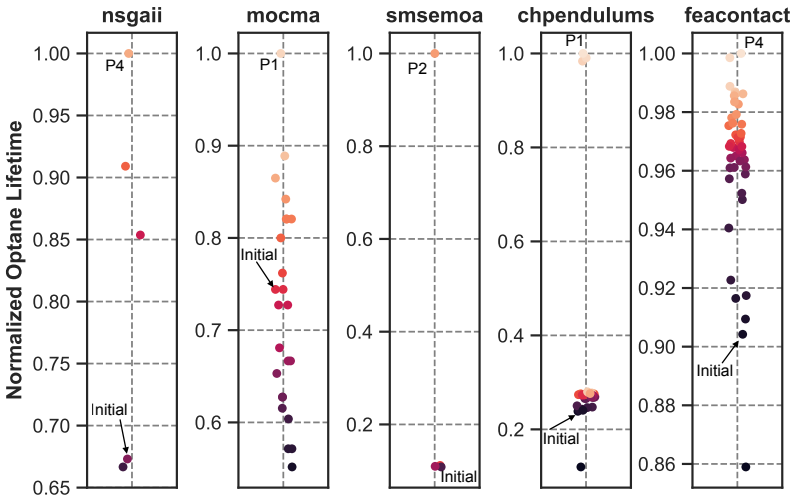


Figure 6.34.: Impact of data organization optimization and placement on the lifetime of the Optane DCPM DIMMs

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 Fig. 6.31. As shown also in Table 6.8, 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.

Fig. 6.35 shows the average memory footprint reduction for the Pareto optimal solutions for each application, respectively, compared to 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 exploration 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. Fig. 6.36 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 ranges from ≈2 hours (SMS-EMOA) to ≈6:30 hours (CH-PENDULUMS). The exploration time depends on the execution time

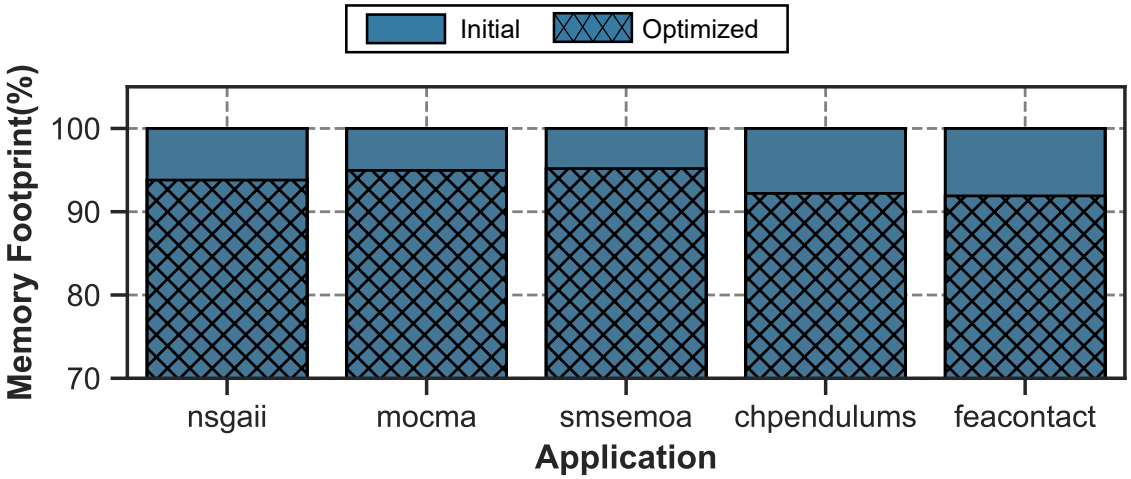


Figure 6.35.: Impact of the methodology on the application memory footprint: The Pareto optimal application versions have lower memory requirements compared to the initial versions.

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 data accesses on other data structures are optimized separately and the pruning of Pareto optimal neighboring solutions, as explained in Sec 6.3.4. 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 Fig 6.37. 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 [273] of the execution time and the number of data structures (close to 1 indicates a more linear correlation). For steps 1, 2 and 3, the Pearson correlation is equal to 0.95, 0.85 and 0.9, respectively. Therefore, we provide near-linear exploration time to the number of data structures over an exponentially increasing design space. This indicates that the exploration is scalable to a higher number of data structures. Finally, we clearly observe that the *Memory object Analysis & Profiling* step (Step ❷) is the most time-consuming, due to the instrumentation for memory object analysis and profiling. On average, the *Optimization of Data Organization* (Step ❶) requires 54min., the *Memory object Analysis & Profiling* (Step ❷) requires 118min. and the *Placement on Heterogeneous Memory Systems & Evaluation* (Step ❸) requires 69min., respectively.

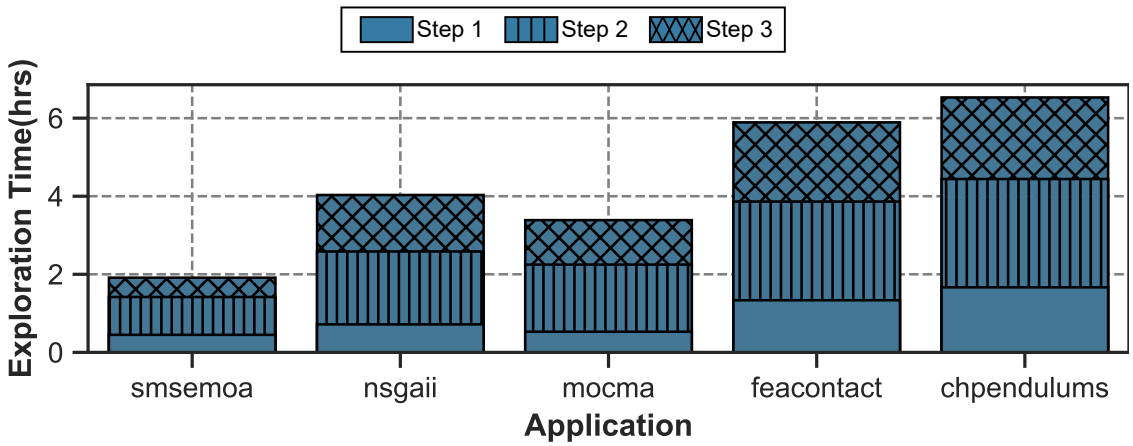


Figure 6.36.: Total exploration time (hours) of each step of the methodology for each application.

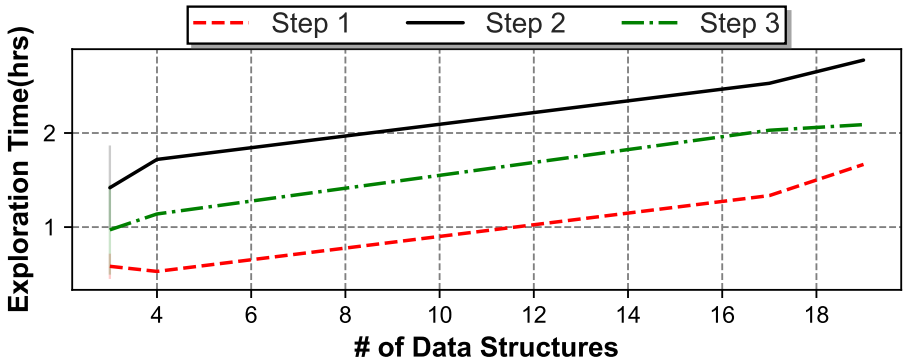


Figure 6.37.: Exploration time over increasing number of target data structures.

6.3.8. Conclusion

This work proposes a methodology for improving the data placement results of applications deployed on DRAM/NVM heterogeneous systems, by integrating a data reorganization step at the application-level. It investigates the impact of high-level optimizations on the results of placement, in terms of execution time, energy consumption and impact on the NVM lifetime. The methodology is scalable and can integrate various data placement algorithms and NVM technologies. Developers can use it as a tool for co-design between applications with increased memory requirements and heterogeneous memory systems. As a future work, we will integrate in the methodology endurance-related models for accurate estimation of the impact of high-level optimizations on the NVM hardware wear.

6.4. On the Implications of Heterogeneous Memory Tiering on Spark In-Memory Analytics

6.4.1. Introduction

From the application developers' perspective, the perpetual rise of data produced has led to the adoption of novel frameworks that enable the parallel processing of huge data volumes in a distributed manner [285–287]. Apache Spark [287] is one of the most widely used processing engines for large scale data analytics, offering a new way of computing with its resilient distributed dataset (RDD), which is stored in memory, while being computed on the latter, thus, avoiding expensive intermediate disk writes found in prior big data frameworks, such as Hadoop [286].

On the other hand, from the providers' standpoint, the increasing memory requirements of modern applications along with the rapid adoption of in-memory analytics frameworks have highlighted the need for novel system architectures, which are able to bypass the fixed resource proportionality of conventional servers, while maintaining performance and maximizing energy and cost efficiency. To this end, multi-tier [288–290] and disaggregated [291–293] memory architectures have been proposed as new design paradigms, where different memory technologies can be leveraged and/or combined according to the needs of running applications. Such architectures can include conventional DRAM technology and Non Volatile Memory(NVM) technologies, such as 3D-XPoint, PCM, 3D DRAM [294] and FeFET [28], which provide trade-offs between access latency, bandwidth, energy efficiency, data persistence and other [161]. In fact, such kind of solutions have been already deployed in various commercial platforms, such as the SAP HANA [295] database and the Aurora exascale supercomputer, which employs the DAOS architecture for storage [30], both taking advantage of the only commercial NVM technology up to now, i.e., Intel Optane DC Persistent Memory(DCPM). Moreover, upcoming technologies, such as Samsung Memory Expander [132] and Compute Express Link (CXL) [32] aim to further bridge existing performance gaps and exploit the limits of multi-tier memory disaggregated platforms, however with the cost of more complex hierarchies.

This simultaneous evolution of the software and hardware "worlds" imposes new challenges on how to efficiently manage the deployment of such applications on heterogeneous memory systems. The co-existence of heterogeneous multi-tier memory architectures with the in-memory application alternatives composes an extended set of exploration for efficient deployment in terms of performance, energy consumption and other aspects. Thus, an efficient exploration of the alternative configurations is required, in order to efficiently

deploy application workloads over multi-tier hybrid memory systems [296]. In-depth analysis for the sufficient memory tier and technology selection is crucial, while the Spark internal configuration is also dominant for the efficacy of the application. The efficient co-selection of hardware and software parameters can fully exploit performance potential, minimize overhead and provide trade-off among hardware and software metrics. However, such kind of selection and efficient correlation among these characteristics is not staightforward.

Although research has illuminated the potential impact of in-memory analytics and the integration of various memory systems, no study to date, to the best of our knowledge, has provided an in-depth characterization and guidelines for Apache Spark in-memory analytics across multi-tier heterogeneous memory systems. In contrast to existing literature, we provide a thorough investigation on the underlying factors and aspects that drastically affect the overall behavior of Spark applications over multi-tier heterogeneous memory systems.

In this section, we present an extensive exploration and characterization of various in-memory Spark applications and workloads over heterogeneous multi-tier memory systems. We present an extended analysis in terms of performance, energy consumption and scalability for in-memory Spark analytics derived from different application domains, i.e, micro-operations, machine learning and websearch. Through our experimental analysis: i) we indicate key aspects that are crucial for the overall performance/energy of the execution, ii) we provide a set of guidelines and key takeaways for efficient deployment over and iii) we indicate current limitations of recent multi-tier heterogeneous memory systems.

6.4.2. Hardware and Software Testbed

Multi-tier Disaggregated Memory System Since the integration and implementation of disaggregated memory systems decoupled from the processor in not trivial, due to absence or existence of complex prototypes [291, 297], we emulate our disaggregated memory system over a multi-socket server with heterogeneous DRAM/NVM memory system. Specifically, our experiments were conducted on a single node with a 2x20 core Intel Xeon Gold 5218R CPU @2.10GHz with 4x32GB DDR4 DIMMs and 6×256GB Optane DC NVDIMMs. Aiming to provide alternative access latency and bandwidth across nodes, we provide asymmetry between Optane DIMMs over the physical sockets, i.e., we deploy 2 DIMMs on socket 1 and 4 DIMMs on socket 2. Last, we configure the Intel Optane DC memory to *App Direct* mode with ext4-DAX file system, which allows the PMEM capacity to be used as byte-addressable persistent memory that is

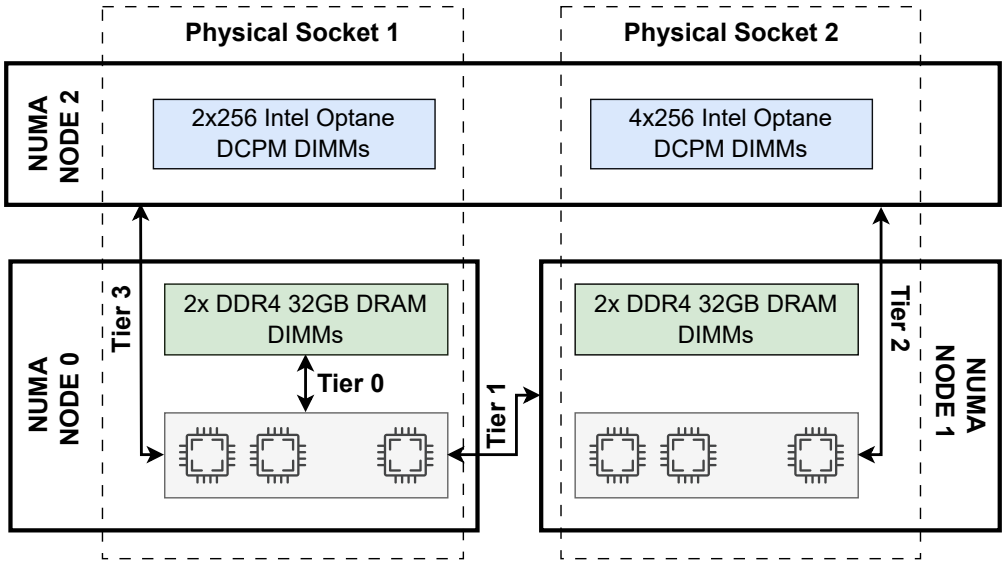


Figure 6.38.: Overview of the target multi-tier system architecture and testbed

mapped into the system's physical address space (SPA) and directly accessible by applications.

Based on the above hardware configuration, our target disaggregated multi-tier architecture is illustrated in Figure 6.38. From an operating system's perspective, the available memory is organized as three distinct, asymmetric NUMA nodes, where NUMA 0 and NUMA 1 are symmetric to each other and include the DRAM memory capacity of socket 0 and socket 1 respectively, whereas NUMA 2 is assymetric and includes the memory capacity of the NVM memory. We define four memory access scenarios (henceforth referred to as different Tiers), that emulate different local (intra-NUMA) and remote (inter-NUMA) allocation modes, as shown in Figure 6.38. Specifically, Tier 0 forms a local allocation mode, where memory is obtained directly from the same physical socket as the cores, whereas Tiers 1-3 are considered as remote modes with heterogeneous memory types each. Traditionally, remote memory accesses reveal higher latency and lower bandwidth per operation [298], due to the overhead of network and data exchange, which is implicitly entailed through our physical hardware configuration described previously. Moreover, the imbalance of NVM DIMMs on each socket allows us to exploit different latency and bandwidth. The idle latency and bandwidth characteristics of the local and remote memory access for our experimental setup are shown in Table 6.10.

Table 6.11.: Examined Spark applications and dataset sizes

Application	Abbr.	Data size range (tiny,small,large)
Sorting of text input data	sort	32KB, 320MB, 3.2GB
Performs shuffle operations	repartition	3.2KB, 3.2MB, 32MB
Alternating Least Squares	als	100, 1.000, 10.000 (users) 100, 1.000, 10.000 (products) 200, 2.000, 20.000 (ratings)
Naive Bayes classification	bayes	25.000, 30.000, 100.000 (pages) 10, 100, 100 (classes)
Random forest	rf	10, 100, 1.000 (examples) 100, 500, 1.000 (features)
Latent Dirichlet Allocation	lda	2.000, 5.000, 10.000 (docs) 1.000, 2.000, 3.000 (vocabulary) 10, 20, 30 (topics)
PageRank	pagerank	50, 5.000, 500.000 (pages)

Spark Engine Configuration We configure the Spark engine to run in a pseudo-distributed, standalone mode, where both Spark Master and Spark Executors reside on the same physical machine. Each computing unit acts independently and is bound with a set of CPU resources either on NUMA node 0 or NUMA node 1 and memory resources from all the available NUMA regions. To achieve this, we force Spark executors to be deployed and

Table 6.10.: Idle access latency and memory bandwidth per tier

		Idle Latency (ns)	Bandwidth (GB/s)
Tier	**Tier 0**	77.8	39.3
	Tier 1	130.9	31.6
	Tier 2	172.1	10.7
	Tier 3	231.3	0.47

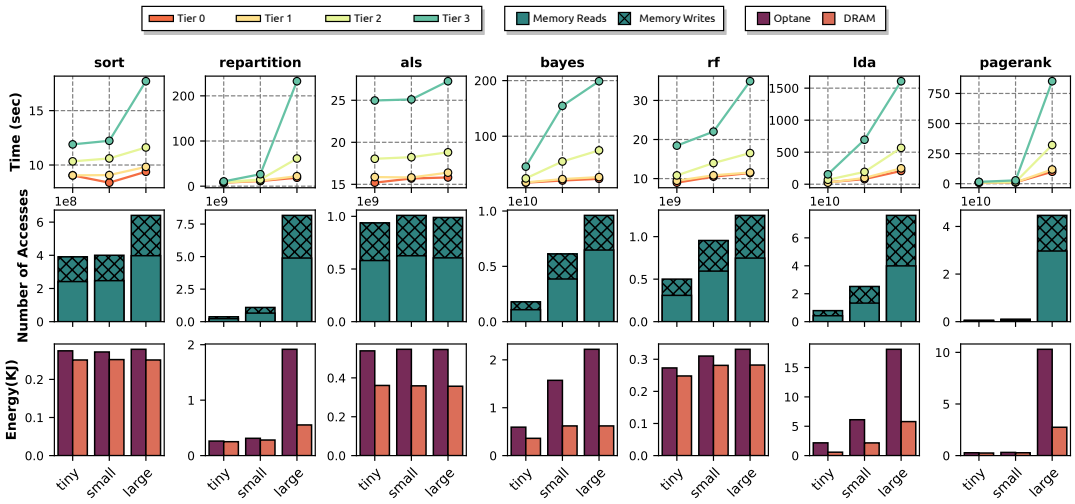


Figure 6.39.: Execution time for all the different memory tiers (top), number of accesses (reads and writes) to memory (middle) and DRAM/NVM energy comparison (bottom) for all the different dataset sizes and benchmarks examined.

access specific compute/memory tiers, by specifying the `cpunodebind` and `membind` flags of Linux's `numactl` tool. By default, in standalone mode, Spark deploys one executor instance that utilizes all the available cores of the bound node (40 hyperthreads in our case). Last, we use Hadoop's Distributed File System (HDFS) [286] instead of the local file system to store input and output data of Spark. Both HDFS and Spark are configured with their default parameters.

Examined Spark applications We utilize a subset of benchmarks derived from HiBench suite [299]. We study 7 Spark applications derived from the HiBench benchmark suite [299], which is widely used to evaluate Spark applications, listed in Table 6.11. These applications form representative examples out of three different workload categories, i.e., micro-operations, machine learning and web searching. Moreover, we examine a diverse set of input dataset sizes per application, i.e. tiny, small and large.

6.4.3. Characterization & Analysis

In this section, we provide an extensive profiling and characterization of our examined examined Spark applications over the different memory configurations considered, as

described in Sec. 6.4.2. Following a Q&A approach, we debate over different aspects that form interesting study factors and, based on our experimental analyses, we provide a set of valuable insights and outcomes to consider when deploying Spark analytics over heterogeneous memory configurations.

How do the different memory tiers affect the performance of our examined applications? First, we examine how allocating memory from different tiers affects the execution time of applications, by using the default Spark configuration (1 executor - 40 cores). Figure 6.39 (top) illustrates the execution time of all the alternative benchmarks, where the Y axis indicates the execution time and X axis the corresponding input dataset size, respectively. We observe that for the majority of the workloads, local execution provides optimized performance compared to remote execution, due to the inherent inferior performance of the remote memory Tiers compared to the local one. Overall, the local execution (Tier 0) achieves 44.2%, 66.4% and 90.1% better execution time on average, compared to Tier 1, Tier 2 and 3 remote execution, respectively. However, there exist certain cases where allocating memory from local and remote nodes lead to similar execution time (e.g., repartition-tiny, pagerank-tiny, pagerank-small), which reveals the potential of certain applications for exploiting remote memory, without sacrificing performance. Moreover, while most of the applications exhibit linear or superlinear execution time increment across the different dataset sizes, `als` shows an almost constant execution time regardless of the input workload and the Tier considered.

⋆ ***Takeaway 1:*** *Performance degradation due to remote memory highly depends on the nature of each application and its workload size, with certain combinations revealing tolerance to the inherent inferior efficiency of remote Tiers.*

Next, we also investigate the impact of each memory technology, i.e., DRAM and Intel Optane DCPM, across the different tiers. Executions bound with DRAM technology (Tiers 1 & 2) exhibit almost similar performance, regardless of the latency and bandwidth overhead due to remote memory fetching. On the other hand, executions that are bound with Intel Optane DCPM DIMMs (Tiers 3 & 4) require 76.7% more execution time, compared to execution bound with DRAM DIMMs, because accessing Optane DCPM requires higher access latency and provides limited bandwidth compared to the latter. Overall, remote memory accesses across different nodes impose an extra communication overhead due to the physical distance of the nodes. Thus, an extra latency penalty is added to the already high access latency of persistent memory, leading to further performance degradation. Applications with higher execution requirements, such as `repartition`, `bayes`, `lda` and `pagerank` are more sensitive to performance degradation, where up to 96.7% more execution time is observed on average for all workloads when deployed on Tier 2 and as

the input workload increases, in contrast to `sort`, `als` and `rf`, where we observe 31.1% degradation on average respectively.

⋆ ***Takeaway 2:*** *The combination of remote memory access penalty along with the inherent latency/bandwidth inefficiency of Optane DCPM compared to DRAM, leads to a disproportional increment on the performance gap between the two technologies as the time of execution increases.*

What are the core bottleneck factors for the observed performance degradation? NVDIMMs are slower compared to DRAM, thus we focus on the insights of the degradation of applications bound with Intel Optane DC. To this end, we monitor the number of read and write memory accesses over the NVDIMMs by utilizing Intel's `ipmctl` tool. Figure 6.39 (middle row) illustrates the respective results. Similar to execution time experiments, as the number of read/write accesses increases, the performance drops significantly. In applications such as `bayes`, `lda` and `pagerank`, where the number of total accesses is an order of magnitude higher compared to the other benchmarks, we notice severe degradation both as the input workload increases and as the application is bound to more distant tiers.

Moreover, in cases where the ratio of write to read accesses increases, we observe a non-linear degradation on the performance of applications. This is due to the asymmetry of read and write operations on the Intel Optane DCPM in terms of access latency, which is known to perform worse when write accesses increase [264]. This is clearly observed for the `lda` benchmark under the `large` workload, whose execution time skyrockets proportionally to the number of write operations performed in the Intel Optane DCPM. Similar observations can be derived for the other benchmarks. Last but not least, apart from the impact of the high number of accesses on the target application, increased number of write operations reduces the lifetime of persistent memory [300], thus in the long-term further performance degradation may occur due to potential hardware failures.

⋆ ***Takeaway 3:*** *Application's performance is highly affected by the number of read and write operations on the persistent memory, with the latter having even more impact by design.*

Does bandwidth or latency dominate performance? In addition to the undeniable overhead added by increased accesses on the NVDIMMs, we aim to investigate further overheads regarding the utilization of persistent memory as a remote tier. For this scope,

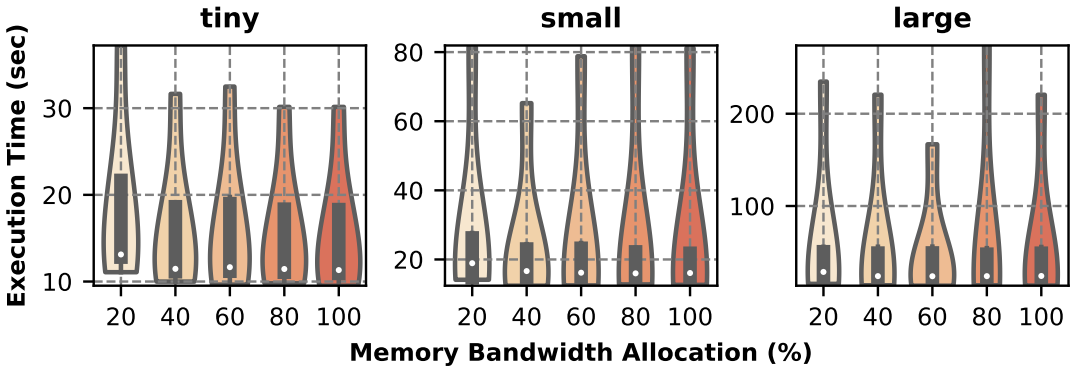


Figure 6.40.: Execution time over various memory bandwidth levels for all benchmarks and tiny,small and large input workloads

we take advantage of Intel's Memory Bandwidth Allocation(MBA) tool [301] and we limit the maximum possible bandwidth utilization to different thresholds. Our experiments were conducted for 1 single executor with 40 cores. Violin plots in Figure 6.40 illustrate the impact of alternative maximum memory bandwidth allocation percentages(X-axis) on the execution time of the target applications(Y-axis) for all input workloads, on average. Both the average execution time and the variance of the distributions does not change as the bandwidth allocation percentage varies for all the input workloads, respectively. Thus, our target applications do not saturate the memory bandwidth, thus it is not a bottleneck for the application's performance. We conclude that the increased access latency to NVDIMMs and the overhead of remote access overhead is the most dominant factor for performance degradation.

⋆ ***Takeaway 4:*** *Performance is highly affected by the access latency of memory technologies, while memory bandwidth is not saturated, thus memory access latency and remote communication are the major bottlenecks.*

Despite its performance degradation, is persistent remote memory energy efficient? Figure 6.39 (bottom) illustrates a comparison of the average energy consumption per DRAM(Tier 0) and Intel Optane DCPM DIMM(Tier 2), over the alternative benchmarks and workloads, respectively. Even though NVMs provide less power consumption per access compared to traditional DRAM technologies, we observe that Optane DIMMs consume higher energy consumption in total compared to the DRAM DIMMs. More specifically, DRAM execution provides 63.9% less energy consumption on average compared to Intel Optane DCPM execution. This is due to the fact that applications deployed on the NVDIMMs are utilized for higher amount of time, thus leading to increased accumulated energy consumption. Furthermore, we observe that the energy consumption,

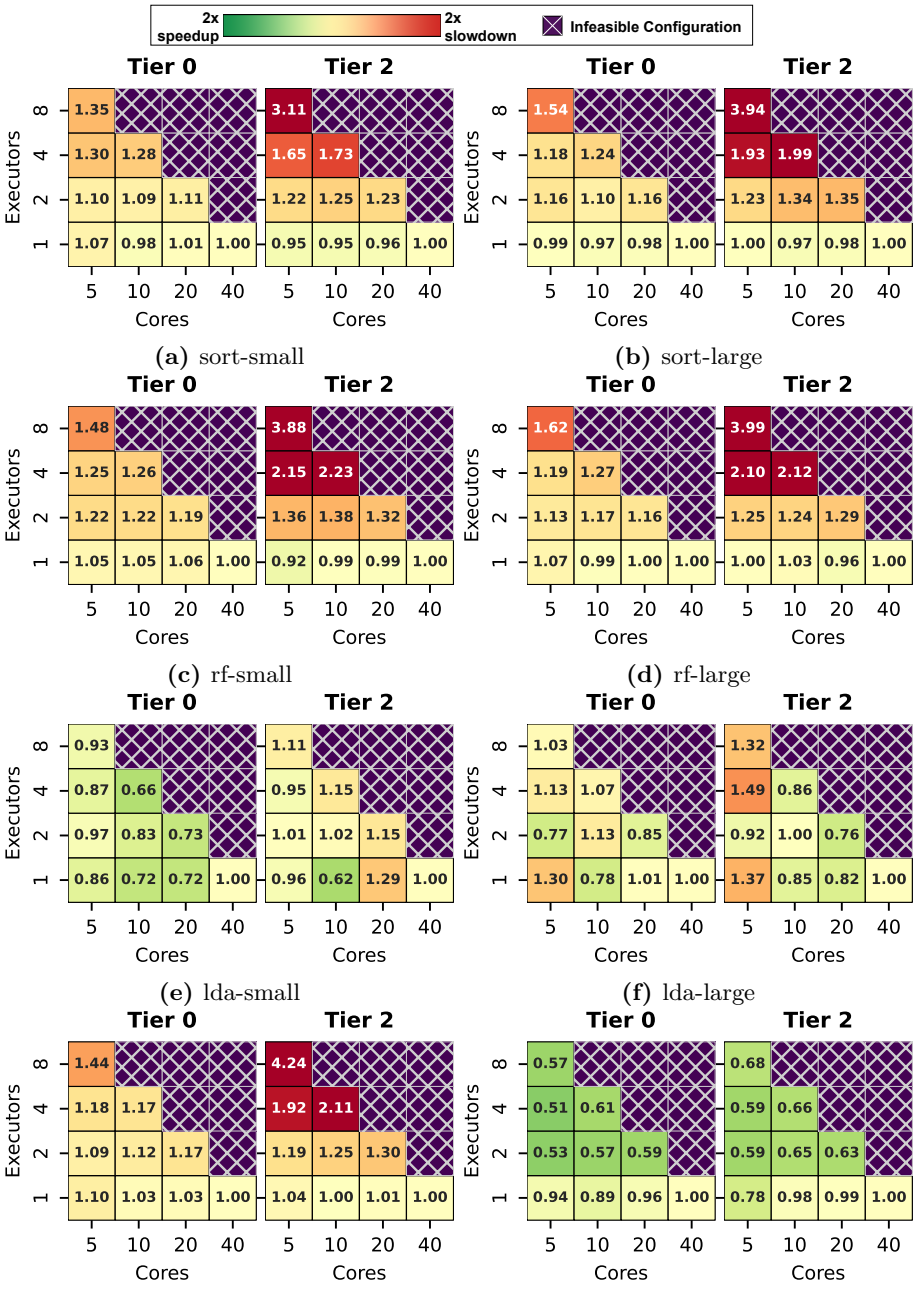


Figure 6.41.: Speedup/Slowdown of `sort`, `rf`, `lda` and `pagerank` applications for all workloads for varying number of cores and executors. The default configuration is 1 executor with 40 cores.

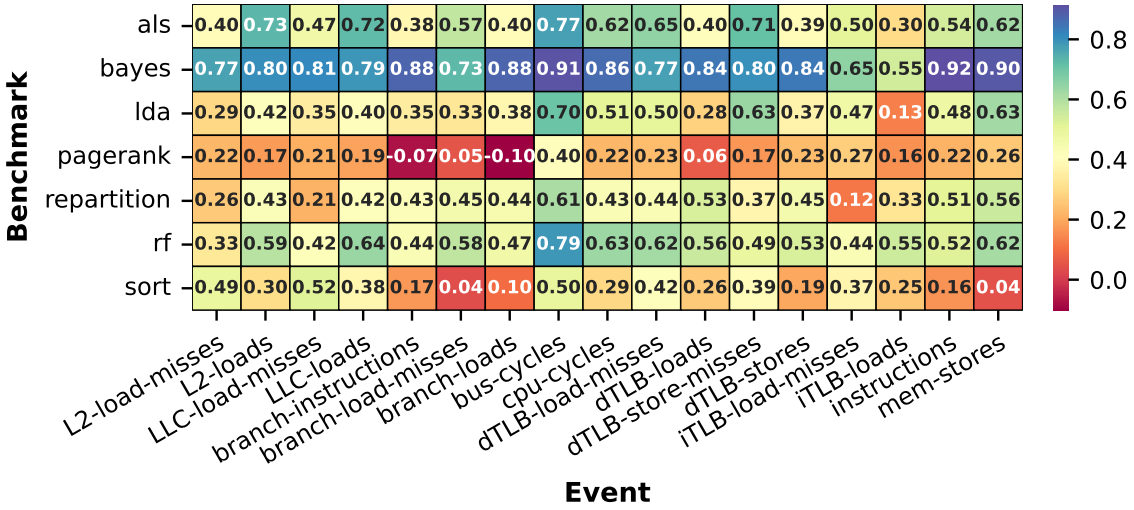


Figure 6.42.: Correlation of system-level metrics with execution time

both on the DRAM and Intel Optane DCPM DIMMs is inline to the execution time scaling, as the input workload size increases. However, there exist workloads, such as `sort` and `als` can scale to larger workloads without significant energy overhead. Thus they can be considered as candidates for remote deployment without significant energy penalty.

⋆ **Takeaway 5:** *Energy consumption is inline with the execution time, however energy over-consumption can be avoided in some cases.*

How does software tunable Spark configuration parameters relate to and affect performance? Next, we investigate how alternating the execution behavior of the Spark engine affects the performance of applications, by examining the performance for different number of executors and cores per executor. With this analysis, we revisit the long-standing debate between “fat" and “skinny" executors for disaggregated memory architectures, which have either isolated access to the memory or compete for shared resources (e.g., executors requesting memory on a rack-scale memory disaggregated system [292]). Figure 6.41 illustrates the relative speedup/slowdown for a subset of the representative benchmark under small and large workloads. X-axis indicates the number of cores per NUMA node and Y-axis shows the number of corresponding executors. As baseline, we consider a single executor over 40 cores (bottom-right square). We observe that `sort`(Fig. 6.41a), `rf` (Fig. 6.41c) and `pagerank` (Fig. 6.41g) benchmarks provide significant slowdown when they are bound to the NVM memory tier by getting down to

3.11× slowdown. The increase of allocated CPU cores per executor does not necessarily provide performance optimization, due to contention in the shared resources of the system and especially the memory bus. Moreover, increased number of executors leads to extra number of accesses on the persistent memory DIMMs for executors co-operation, thus leading to performance degradation. In contrast to the examined benchmarks, the `lda` (Fig. 6.41e) benchmark is not significantly affected as the number of cores/executors ranges.

⋆ **Takeaway 6:** *Increased number of executors that compete over shared memory resources leads to further performance degradation, with persistent memory being even more susceptible to resource contention.*

Moreover, we investigate how alternative combinations of cores and executors behave over larger amounts of workload. Figures 6.41b- 6.41h illustrate the corresponding speedup/slowdown for the examined applications under the large workload. For the `sort`, `rf` and `lda` the qualitative comparison under small and large workloads is quite similar. However, regarding the `pagerank`(Fig. 6.41h) benchmark, speedup is observed as the number of executors increases, in contrast to small workload(Fig. 6.41g), where significant slowdown is observed as the number of executors rises. This is due to the fact that for the large workload, efficient data partitioning across the executors and sufficient resource utilization is occurred and executors are not underutilized. As the input dataset size increases, the benefits of parallel processing and distribution across multiple executors become more pronounced, leading to a speedup in the overall processing time. Similar observations can be derived for the other benchmarks.

⋆ **Takeaway 7:** *Workload size affects the performance, however there exist benchmarks who handle better high workloads.*

Can we somehow obtain a rough estimation of performance degradation on remote memory tiers? Last, we investigate the potential of exploiting different metrics for estimating the performance of applications across the several tiers. Specifically, we examine two directions *i)* how system-level events correlate with execution time when deployed on local memory and *ii)* how performance correlates with the hardware specifications (Communication Latency/Memory Bandwidth) of each tier. Inspired by prior research [302], we investigate the relationship between system-level metrics to higher-level metrics of interest, i.e., execution time. Towards this direction we evaluate the system-level event Pearson correlation [273] with execution time, as depicted in Figure 6.42. We observe that `bayes` benchmark has the highest correlation(near-linear) with almost all system-level metrics, thus linear prediction models are expected to perform efficiently in execution time pre-

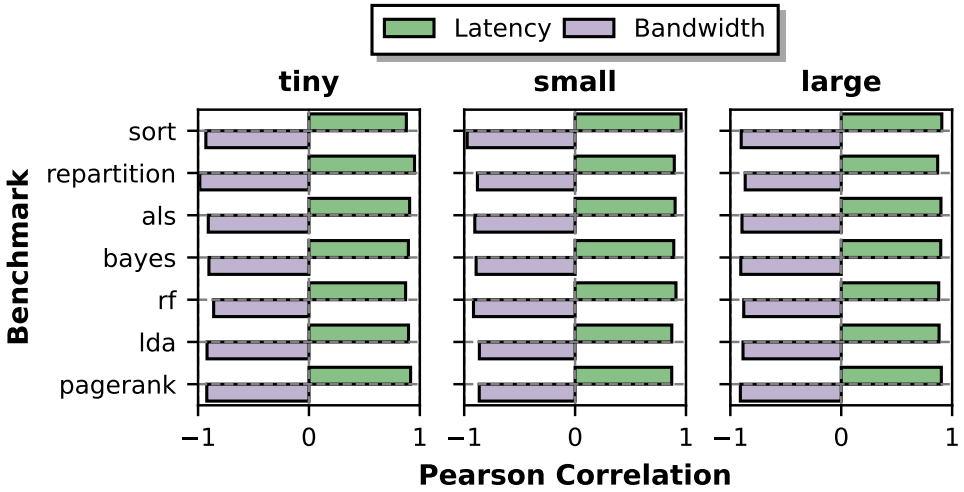


Figure 6.43.: Correlation of hardware specs (latency and bandwidth) with execution time for all applications and workloads

diction over new tiers deployed. On the contrary, benchmarks such as `pagerank`, have low correlation to system-level metrics, thus more complex models are required for providing efficient time prediction on alternative tiers. Moreover, Figure 6.43 indicates the Pearson correlation of execution time with bandwidth and latency for each individual application and workload, across all tiers. We observe that the execution time converges to near perfect positive(1) and negative(-1) correlation for both latency and bandwidth, respectively. Thus, execution time is highly correlated with both latency and bandwidth, therefore linear prediction models are expected to perform efficiently. Overall, we expect that by combining the hardware-related specifications along with system-level metrics, we can create accurate predictions of performance degradation across the different tiers, by using analytical models and/or Machine Learning techniques.

⋆ **Takeaway 8:** *The inherent latency/bandwidth specifications and system-level events reveal high correlation with performance over different tiers, thus, they can be utilized for prediction of execution time on heterogeneous memory tiers.*

Discussion and future perspectives The outcomes of this section reveal that Spark applications are highly affected by disaggregated memory architectures, mainly due to the high latency imposed by remote accesses. However, there are cases where remote memory can be employed without any discount in performance, whereas there is also plenty room for exploration w.r.t. determining the optimal memory tier per access type, especially in the case of persistent memory. On top of that, since Spark is designed to

work over distributed nodes, further optimizations can be performed on the engine itself, to leverage a unified disaggregated memory architecture thus avoiding shuffling operations and minimize the overhead of remote memory access and scale across memory tiers. Last, the high correlation among system-level metrics and execution latency gives promising signs for employing Machine Learning techniques to effectively predict the performance of applications across different memory tiers.

6.4.4. Conclusion

In this section, we conduct an extensive characterization of Spark in-memory analytics over heterogeneous DRAM/NVM memory tiering. We investigate the impact of various factors that affect high-level characteristics, such as performance and energy consumption. We provide deployment guidelines and explore effective system and application configurations, while we denote potential future aspects. The major outcomes of this work can be exploited by developers who target Spark analytics over multi-tier heterogeneous memory systems.

Chapter 7.

Hardware-Level Memory Management and Configuration

The rapid growth of heterogeneous DRAM/NVM systems impose new challenges in terms of page placement and migration among the alternative technologies of the heterogeneous memory system. In this chapter, we present a novel approach for efficient page placement on heterogeneous DRAM/NVM systems. We design an adjacent LSTM-based approach for page placement, which strongly relies on page accesses prediction, while sharing knowledge among pages with behavioral similarity. The proposed approach leads up to 65.5% optimized performance compared to existing approaches, while achieving near-optimal results and saving 20.2% energy consumption on average. Moreover, we propose a new page replacement policy, namely clustered-LRU, achieving up to 8.1% optimized performance, compared to the default Least Recently Used (LRU) policy.

Moreover, advances in applications that are highly dependent on efficient cache utilization, in addition to the rapid growth of Edge computing systems deployed with RISC-V processors, generate a complex paradigm across the hardware/software continuum. We propose a systematic exploration methodology for effective cache configuration on RISC-V architectures. Our exploration relies on fine-tuning parameters as cache size, associativity and replacement policy, aiming to provide performance optimization in terms of execution time. This approach acknowledges the evolving landscape, addressing intricate challenges in the hardware/software continuum and offering valuable insights for advancing computing systems in dynamic environments. Our proposed methodology is evaluated on a set of benchmarks deployed over GEM5 RISC-V simulated system, showing that i) the effective exploration of alternative cache configurations can lead up to 43.8% execution time optimization, ii) the integration of sufficient replacement policy and coherence protocol can further affect the overall system efficiency and iii) we provide insights on the impact of application's organization to the cache configuration.

This chapter's text strongly relies on the following publications:

- *Katsaragakis Manolis, Christos Lamprakos, Peter Kourzanov, Manu Komalan Perumkunnil, Lazaros Papadopoulos, Francky Catthoor & Dimitrios Soudris. Optimizing Cache Configurations via Dynamic Memory Management in Edge Computing Systems, IEEE Embedded Systems Letter (ESL), 2024 (Under review).*
- *Katsaragakis Manolis, Stavrakakis Konstantinos, Masouros Dimosthenis, Papadopoulos Lazaros, & Soudris Dimitrios (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 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik. [190]*

7.1. Adjacent LSTM-based Page Scheduling for Hybrid DRAM/NVM Memory Systems

7.1.1. Introduction

Modern commercial state-of-the-art platforms integrate Intel Optane DC Persistent Memory (DCPM) in the same memory layer with DRAM, providing a hybrid memory system, which consists of a large, energy efficient, but high latency (and/or low bandwidth) NVM and a fast DRAM of limited capacity, increased energy requirements and higher cost per byte. Aiming to identify trade-offs between performance and energy consumption over hybrid DRAM/NVM systems, the clustering of pages is a promising solution [74]. Traditional page scheduling approaches tend to store frequently accessed (hot) data on the fast DRAM, in order to optimize performance and least accessed (cold) data on the slow, but energy efficient NVM, in order to provide energy gains [76]. Nevertheless, providing an efficient page clustering and placement scheme still remains non-trivial. Several prior state-of-the-art researches have investigated the problem of page placement [75, 76, 172]. However, existing approaches do not take advantage of spatio-temporal characteristics of the application's pages, but their decision making is limited on the behavior of each individual page behavior, thus restricting the potential performance and energy gains that can be derived.

In order to fully exploit the hybrid memory systems characteristics, the Machine Learning paradigm is widely integrated in operating systems [74], as it can combine near-

optimal results, low-overhead and real-time decision making. In this work, we introduce a page scheduling approach, which integrates a *Critical Page Selector* mechanism, a *History-based Predictor* and an *Adjacent Long Short Term Memory (LSTM)-based Predictor*. Page placement algorithms can strongly take advantage of spatio-temporal locality among pages [303]. Thus, we use LSTM networks to enable fast and accurate prediction of the memory accesses per page, respectively, in order to take advantage of temporal locality. Moreover, we enhance our predictor with knowledge of neighboring pages, aiming to take advantage of spatial locality and the behavioral correlation of neighboring pages [303]. The essential tuning knob of the system is the designation of the pages to be placed on a DRAM/NVM system, aiming to optimize performance and/or reduce energy consumption. **The novel contributions of this chapter are the following:**

- **We propose a *Critical Page Selector***, for efficiently identifying and clustering critical pages based on read and write access operations over the hybrid memory system.

- **We propose an *Adjacent LSTM-based Predictor* mechanism** for accurately predicting the number of page memory accesses. We integrate knowledge sharing among application pages with behavioral similarity based on k-nearest neighbors clustering and we evaluate our approach against state-of-the-art solutions, showing that our framework achieves up to 65.5% optimized performance and 20.2% less energy consumption on average.

- **We integrate a novel page eviction policy** on main memory, namely *clustered-LRU*, which extends the default, widely used, Least Recently Used (LRU) policy, based on the K-means clustering algorithm.

7.1.2. Proposed Methodology

The key idea of our proposed approach is to provide a run-time decision making scheduler for efficiently placing pages over heterogeneous DRAM/NVM memory systems, in order to optimize performance and energy consumption. More specifically, the main objective is to maximize the number of application's memory requests that are served from DRAM, thus to maximize performance, while reducing the energy consumption by placing data on low-power NVM when they are not frequently processed. Our proposed scheduler consists of three major components: (i) the *Critical Page Selector*, (ii) the *Adjacent LSTM-based Predictor* and (iii) the *History-based Predictor*. *Critical Page Selector* is responsible for efficiently identifying the critical pages that have significant impact on the overall

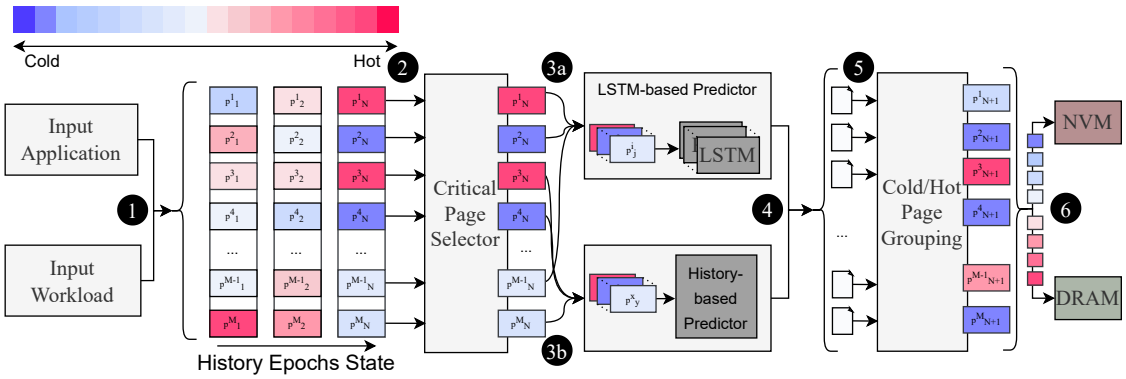


Figure 7.1.: Overview of the proposed page scheduler over heterogeneous DRAM/NVM systems

performance, based on past memory accesses. The *Adjacent LSTM-based Predictor* is responsible for providing memory accesses predictions based on LSTM networks, aiming to co-exist and optimize *History-based Predictor*, which provides memory accesses predictions, however its effectiveness in providing applications with fast (i.e., in-DRAM) data accesses inherently depends on the application data access behavior [76]. Typical page placement approaches mostly rely on history-based approaches, resulting to sub-optimal placement decisions. Our proposed scheduler is illustrated in Fig. 7.1. As input to our framework, we provide the target application with the corresponding workload(❶). The application consists of M memory pages (p). Page scheduling occurs on every scheduling epoch. As scheduling epoch, we define the interval between two consecutive placement decisions.

7.1.3. Critical Page Selector

The core functionality of the *Critical Page Selector* component is the identification of performance-critical pages of the input workload, based on their actual memory accesses over the past N scheduling epochs. Ideally, if computational resources and real-time requirements were not a bottleneck, the maximum prediction accuracy could be achieved by deploying a single LSTM for each application page, respectively. However, this would lead to increased demands in terms of required resources, while the decision latency would skyrocket. Thus, the *Critical Page Selector* is designed to classify the pages on two major categories: a) *Critical*, which require sophisticated decision making through LSTM-based solution and a miss-placement can be significant for the application's performance and b) *Non-Critical*, which represent those pages, whose influence is not that significant for the application's performance, thus they are tolerant

to miss-placements and a typical lightweight History-based Predictor can handle this decision.

The classification criteria of pages to *Critical* and *Non-Critical* is strongly related to the number of past read and write accesses of each page, respectively. Due to the inherent asymmetry of read and write access latency on NVMs and their limited write endurance, the impact of write operations on the application's performance and energy consumption is more dominant. Thus, we consider weighted accesses of page p for the scheduling epoch i, as shown in Eq. 7.1. Constants α and β are set to 0.75 and 0.25, respectively.

$$Accesses_i(p) = \alpha \times writes_i(p) + \beta \times reads_i(p) \tag{7.1}$$

For the scheduling epoch i, we define a binary function $MP_i(p)$ (Eq. 7.2), based on the placement decision, which indicates whether the page p was misplaced on scheduling epoch i. A page is defined as misplaced by the *Critical Page Selector*, when the page was not placed in the optimal memory technology, due to page hotness inaccurate prediction in the past. A page is characterized as hot when the number of past accesses exceeds a user-defined threshold h and thus is expected to maintain this behavior in the next scheduling epoch.

$$MP_i(p) = \begin{cases} 1 & \text{if } p \text{ was misplaced on epoch } i \\ 0 & \text{if } p \text{ was properly placed on epoch } i \end{cases} \tag{7.2}$$

For every individual page p of the input application and for N past scheduling epochs, we define profit function (Eq. 7.3), which is the essential factor for the final classification of the misplaced pages and indicates impact of the corresponding misplacement, measured on the summary of accesses of N previous epochs.

$$\text{Profit}(p) = \sum_{i=0}^{\text{N}} Accesses_i(p) * MP_i(p) \tag{7.3}$$

In order to classify the pages, we define λ, a user-defined constant. If the profit function of page p is higher than λ, then the page p is considered as *Critical*, therefore it is fed as input to the *LSTM-based Predictor*(3a), otherwise it is considered as *Non-Critical* and given as input to *History-based Predictor*(3b).

Adjacent LSTM-based Predictor This component is responsible for accurately predicting the number of *Critical* page accesses(as derived from *Critical Page Selector*) for the

next scheduling epoch. We utilize a single LSTM for every individual page, consisting of two layers with 256 neurons per layer, followed by one Dense layer. Furthermore, we deploy k-nearest neighbors algorithm, in order to encapsulate the spatio-temporal locality of our application without adding significant computation overhead [303]. We cluster the M application pages on groups of k pages, according to behavioral similarity on previous scheduling epochs. Constant k is set to 8. The input of the LSTM for page p on scheduling epoch i is a 2D-feature vector of size $k \times l$, where l denotes the history length, which is set to 20 through experimentation.

The output of the per-page LSTM is the predicted number of main memory accesses for the next scheduling epoch. The input sequence is normalized between 0 and 1 using *softmax()* function, to stabilize the gradient descent step and achieve faster convergence. In contrast to existing approaches [74, 76], the only information required from the model are the relative accesses of the pages, as the absolute prediction of accesses would lead to extra computational overhead and potential fault predictions. Significant role for the efficiency of our proposed scheduler has the selection of the loss function. In contrast to existing approaches, such as [76] which utilizes the *mean squared error* loss function, we design and implement a custom *weighted loss function*, aiming to weight each prediction's contribution to the cost function. The custom loss function for page p on scheduling epoch i is defined in 7.4.

$$\text{Loss}_i(p) = real_i(p) * log(predicted_i(p)) * w_i(p) \tag{7.4}$$

The variables $real_i(p)$ and $predicted_i(p)$ denote the normalized real and predicted value of page p on epoch i, respectively, while $w_i(p)$ indicates the corresponding weight. The neural network aims to minimize the custom loss value between the predicted and actual values, using the *Adam* optimizer and a learning rate equal to 0.01. The training stops if the loss for the validation dataset is not reduced for s consecutive training epochs. We set s equal to 30 epochs.

Regarding the training phase of our proposed scheduler, the training time of an LSTM for every individual page can skyrocket due to huge overhead. Thus, inspired by [74], we detect the most dominant pages over the c most performance-critical pages. Variable c is a user-defined threshold derived through extensive experiments. 75% of the dataset is utilized for training and 25% for validation.

History-based Predictor The *Non-Critical* pages, as derived by the *Critical Page Selector* are fed as input to the *History-based Predictor*, which strongly relies on existing approaches, such as [304]. The core functionality of this component is the selection of page accesses based on past traffic patterns and strongly relies on the assumption that

both cold and hot pages will maintain their pattern based on the latest scheduling epochs, e.g. a highly-accessed page will remain highly-accessed. This kind of scheduler is vulnerable to mis-predictions. However, due to the *Critical Page Selector* strategy, such faults will not significantly affect the performance of a page misplacement on DRAM/NVM system.

Page Grouping, Placement and Migration The predictions of the *LSTM-based Predictor* and the *History-based Predictor* are finally accumulated and sorted in ascending order based on the number of the predicted memory accesses. Our grouping policy aims to split the pages into two distinct groups, i.e. hot pages and cold pages, based on a user-defined threshold $t\epsilon[0,1]$. The $t\%$ is selected to be placed on the DRAM, while the $(1-t)\%$ is placed on the NVM. Through this classification the hot pages can benefit from the fast DRAM, while cold pages that are not accessed frequently take advantage of the low-power storage of persistent memory.

However, the state of each individual page can be modified in consecutive scheduling epochs, thus creating the necessity for page reorganization between the DRAM and NVM memory technologies. Based on the prediction of the upcoming epoch, each page is either maintained on the same memory technology or migrated to the other technology. We target engines that allow seamless page migration, which is overlapped with the computation, thus overhead of the migration between DRAM and NVM is negligible [76, 305].

Clustered-LRU Replacement Policy Aiming to further optimize the performance of our proposed framework, we integrate a page eviction policy from the main memory, namely *clustered-LRU*. Our proposed replacement policy extends the LRU replacement policy, which is the state-of-the-art replacement policy on main memories. Inspired by the clustering approach implemented in [74], we integrate a k-means clustering approach, in order to cluster the pages.

The number of clusters is chosen through extensive evaluation based on inspecting the distribution of addresses of the whole address space. Each page is clustered with respect to the number of read and write accesses on the main memory (DRAM or NVM). Through this clustering, the sparsity of the address space of an application is avoided. Every individual page is assigned with a unique cluster ID, which represents the group of pages that the corresponding page belongs to.

During every scheduling epoch, we maintain information about the most active address

Benchmark Suite	Benchmark	ID	Application Domain	Page Write Accesses	Page Read Accesses
PARSEC [306]	Blackscholes	PS1	Finance	4881	66216
	Bodytrack	PS2	Computer Vision	32411	122371
	Streamcluster	PS3	Data Mining	165495	31915
Rodinia 3.1 [307]	Backprop	RD1	Machine Learning	286869	302419
	Bplustree	RD2	Graph Theory	108274	346723
	Hotspot	RD3	Physics Simulation	5186	197473
	K-means	RD4	Data Mining	187364	356603
	Lud	RD5	Linear Algebra	370481	329342

Table 7.1.: Overview of Benchmarks

clusters, i.e. the clusters that have the highest number of page read and write accesses on the main memory. Clusters are then sorted in descending order based on the number of accesses, thus pages that belong to the most active clusters will be prioritized to retain their position in DRAM during the eviction process. Through this policy, we take advantage of the spatial and temporal locality of the data, based on the observation that a cluster that is highly active during a scheduling epoch will probably remain active within the next scheduling epoch as well. Moreover, by maintaining the most highly-accessed clusters on the main memory, the number of page accesses on the slow disk are avoided, thus performance degradation is avoided. subsectionExperimental Setup and Evaluation

7.1.4. Experimental Setup

Our experiments were conducted on a single node with a 2x20 core Intel Xeon Gold 5218R CPU @2.10GHz with 4x32GB DDR4 DIMMs. In order to derive the required tracing data, we utilize the Intel Pintool [308], which is the standard tool for dynamic binary instrumentation framework for x86-64 instruction-set architecture that enables the creation of dynamic program analysis custom tools. We consider 4 KB virtual page ID, that corresponds to the virtual memory address and we group memory accesses into scheduling epochs interval similar to [76].

Similar to existing approaches [76] and in order to evaluate the impact of our proposed scheduler, we design an in-house lightweight heterogeneous DRAM/NVM memory system simulator. We simulate a memory system that consists of a fast memory (i.e., DRAM) and one with high access latency (i.e., NVM). The specific memory characteristics regarding the access latency per memory technologies are derived by [76]. We integrate the *Adjacent LSTM-based Scheduler* on our simulator, which is responsible for placing, migrating and monitoring the behavior of pages. Last but not least, for our evaluation we utilize a set of benchmarks from alternative application domains, derived by Rodinia 3.1 [307] and PARSEC [306] benchmark suites. Table 7.1 summarizes the benchmarks used for our

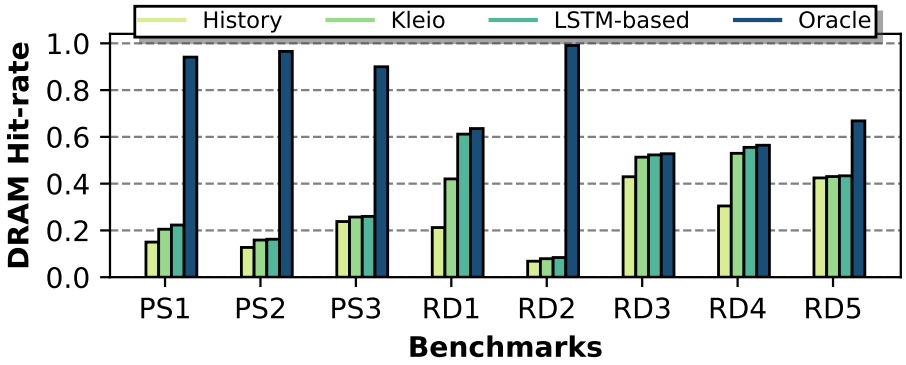


Figure 7.2.: Normalized performance comparison of alternative placement scheduling policies. Results are normalized based on Oracle prediction.

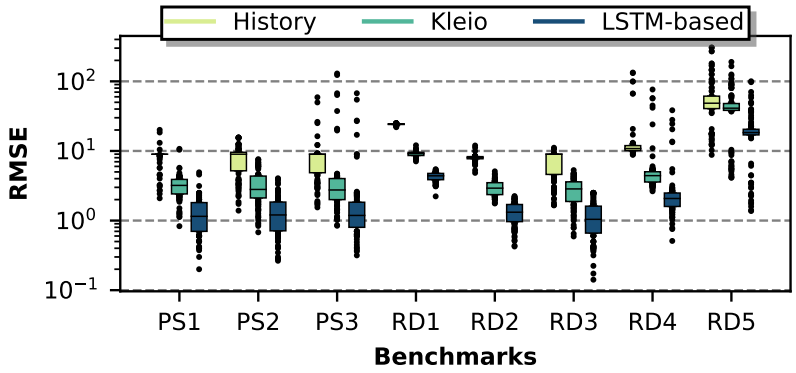


Figure 7.3.: Root Mean Square Error(RMSE) comparison of alternative placement policies for all benchmarks.Y-axis is in log scale.

experimental evaluation and technical characteristics regarding the page read and write page accesses.

Comparative Performance Analysis: We evaluate our *Adjacent LSTM-based page scheduler* against existing scheduling algorithms. Initially, in order to quantify the performance of our approach against the optimal solution, we augment the comparison by adding an Oracle prediction mechanism, which possesses all the page information a priori and provides the optimal solution based on exhaustive search method. Moreover, we implement from scratch and compare our proposed scheduler against a history-based approach [304], which is a straightforward solution for page placement on heterogeneous memory systems, based on the pattern of previous history epochs. Finally, we implement from scratch and compare against a state-of-the-art page scheduler which utilizes machine intelligence, namely *Kleio* [76]. As evaluation metric we utilize the *DRAM hit-rate*

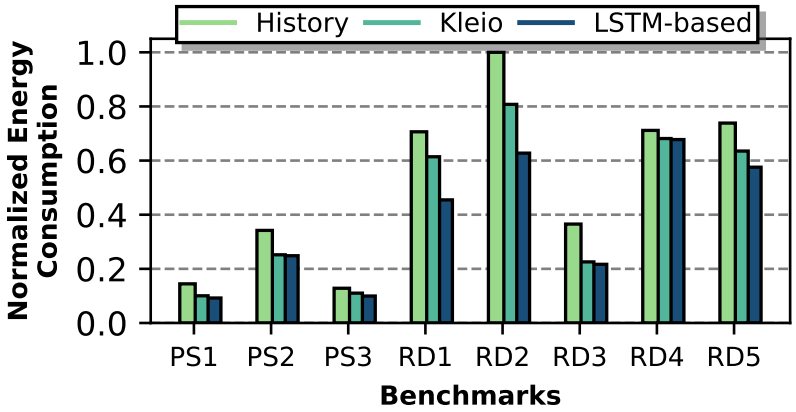


Figure 7.4.: Normalized energy consumption of alternative placement scheduling policies. Results are normalized based on History-based scheduler.

metric, indicating the percentage of requests served from DRAM (higher DRAM hit-rate corresponds to higher performance).

Figure 7.2 illustrates the comparative analysis of the alternative scheduling policies. The X-axis denotes the corresponding benchmark ID, as derived from Table 7.1, while Y-axis represents the DRAM hit-rate. We observe that our approach outperforms both history-based scheduler and *Kleio* in all experiments, by achieving up to 65.5% and 32.7% higher DRAM hit-rate, respectively. As expected, history-based predictor behaves worse compared to other approaches, as it does not have any intelligence for placement. Regarding the comparison with *Kleio*, our *Adjacent LSTM-based scheduler* performs better due to the fact that we achieve optimized placement and migration decisions. Moreover, our proposed scheduler converges to the optimal solution (RD1, RD3, RD4, RD5 derived by the Oracle prediction, by achieving down to 3.9% miss from the optimal solution(RD3). However, in a subset of benchmarks(PS1, PS2, PS3, RD2), there is still space for further optimization in order to approximate near-optimal results. This is due to the fact that automatic memory pattern discovery is not straightforward and cannot be effectively determined by Recurrent Neural Networks(RNNs), as it has already been indicated in [74].

Comparative Accuracy Analysis: The functionality of our scheduler is directly linked to the efficiency of our proposed *Adjacent LSTM-based predictor* for page accesses prediction. Thus, we evaluate the Root Mean Square Error (RMSE) of our predictor and compare with the prediction mechanisms of the history-based approach and *Kleio*, respectively. Figure 7.3 illustrates the RMSE comparison of alternative approaches. The X-axis denotes the corresponding benchmark ID and the Y-axis reports the RMSE in log-

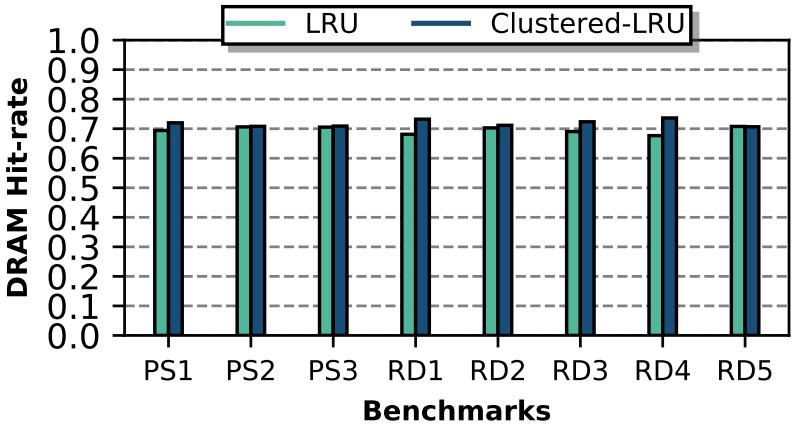


Figure 7.5.: Comparison of LRU vs Clustered-LRU replacement policies over alternative benchmarks.

arithmic scale. As expected, the history-based predictor does not provide high accuracy, as it only relies on past epochs. The predictor integrated on our scheduler outperforms both the history-based scheduler and *Kleio*. More specifically, our proposed scheduler achieves 74.8% less RMSE compared to history-based approach on average and 54.9% less RMSE compared to *Kleio* on average. The former is due to the fact that history-based scheduler does not provide significant machine intelligence, thus there exists high randomness, while the latter is due to the fact that our *Adjacent LSTM-based predictor* takes advantage of the knowledge of pages with behavioral similarity in terms of access pattern, in contrast to *Kleio*, which does not exchange any knowledge among pages.

Comparative Energy Consumption Analysis: Efficient page scheduling policies for heterogeneous memory systems should be able to provide sustainable behavior. Thus, we perform experiments for evaluating the overall energy consumption of both DRAM and NVM. We utilize the energy models presented in [309]. Figure 7.4 depicts the normalized energy consumption of the alternative schedulers over the benchmark workloads. *Adjacent LSTM-based* approach performs significantly better in the energy consumed. More specifically, our proposed scheduler achieves up to 38.7% and 24.3% less energy consumption compared to history-based and *Kleio* scheduler, respectively.

The key insight for the reduced energy consumption of the *Adjacent LSTM-based scheduler* is the efficient page migration between DRAM and NVM. Through the efficient prediction mechanism, pages remain in the DRAM as long as it is predicted to have increased number of accesses, i.e. characterized as hot, while for the rest of the predictions are migrated and stored in the low-power NVM. Once they become hot again they mi-

grate again to DRAM, for the period of the increased number of accesses, thus combining increased performance and low energy consumption.

Aiming to further exploit the behavior of energy consumption, we first observe that there is a correlation between performance and energy. For instance, the RD2 benchmark provides increased energy consumption, while it provides comparatively low DRAM Hit-rate. This is due to the fact that both DRAM and NVM are utilized for higher period of time. Moreover, we observe that there exist benchmarks which even though they provide low DRAM Hit-rate they also provide reduced energy consumption (e.g. PS3). This is due to the fact that the PS3 benchmark is read-heavy. Through the memory traces collected, the write/read ratio is measured to 0.19. Thus energy-expensive write operations are avoided. Similar observations can be derived for the other benchmarks.

Replacement Policy Evaluation: Last but not least, we evaluate the impact of the underlying page replacement policy on main memory over out LSTM-based framework. We compare our proposed replacement policy, namely clustered-LRU, against the default replacement LRU policy. Figure 7.5 indicates the DRAM Hit-rate optimization derived by our proposed replacement policy. We observe that the DRAM hit-rate is optimized up to 8.1% compared to the default LRU policy. This is due to the fact that highly-accessed pages are mainly accessed directly from the main memory and not from the low-latency disk. Moreover, through our proposed approach, the replacement of clusters of pages instead of single pages reduces the number of replacement requests both on DRAM and NVM.

7.1.5. Discussion

In this subsection, we indicate key findings derived from our experimental analysis and evaluation. **Machine intelligence methods can be effectively utilized and combined for page placement.** Throughout our observations on the experimental analysis we observe that the combination of alternative machine learning models, such as LSTMs and k-nearest neighbors in our case, can boost the performance(Fig. 7.2), as the most crucial factor for optimized performance is the accuracy of the corresponding predictor. Thus, the exploration and integration of alternative machine learning approaches is of major research interest, as the page placement can provide further optimizations, especially in application workloads that have still performance gap compared to the Oracle scheduler.

Furthermore, **poor replacement policy cannot allow an heterogeneous memory**

system to fully benefit from an efficient page scheduler approach. As indicated in Figure 7.5 widely utilized approaches, such as the LRU policy have still significant space for optimization. Therefore, the investigation of alternative replacement policies can further optimize the performance of existing placement algorithms.

Efficient data placement and migration can exploit the advantages of the sustainable nature of recent NVM technologies. As indicated in 7.4 an efficient page placement scheduler provides energy optimizations, despite the fact that the main objective of our *Adjacent LSTM-based scheduler* is performance-oriented. Further energy optimizations can be potentially achieved through the design of page placement policies aiming to optimize energy/power consumption.

7.1.6. Conclusion

In this section, we present a novel approach for efficient page placement on heterogeneous DRAM/NVM systems. We design an *Adjacent LSTM-based approach* for placing pages based on page accesses prediction through knowledge of pages with behavioral similarity. Our proposed approach leads up to 65.5% optimized performance compared to existing approaches, while achieving near-optimal results in some cases and saving 20.2% energy consumption on average. Moreover, we propose a new page replacement policy, namely *clustered-LRU*, achieving up to 8.1% optimized performance, compared to the default LRU policy.

7.2. Optimizing Cache Configurations via Dynamic Memory Management in Edge Computing Systems

7.2.1. Introduction

Applications with unusually large working sets, characterized by intensive data access patterns and high cache usage are becoming increasingly susceptible to cache configuration, as it strongly influences their overall performance. Examples include data analytics, biomedical applications [212], and AI workloads, often requiring specialized optimization techniques for efficient execution [310]. These applications often exhibit temporal and spatial locality, making them highly dependent on efficient cache utilization. Deploying such applications in resource-constrained nodes and in order to take fully advantage of

the underlying memory and cache system, developers need to avoid the limitations imposed by static memory management (e.g. inefficient memory allocation, limited cache utilization and increased memory footprint) and support effective dynamic memory management throughout the execution of the examined application [143, 212]. Thus, applications characterized by highly static memory allocation can not meet the constraints of typical embedded devices with limited resources.

Additionally, the effective selection of cache configuration options, i.e. cache size, associativity, and replacement policies across the cache hierarchy continuum significantly impacts the speed and efficiency of computation. More specifically effective cache configuration: i) significantly reduces memory access latency by ensuring that frequently accessed data, ii) ensures that computing systems operate at their highest potential efficiency. In essence, effective cache configuration and the ongoing exploration of its design space are essential for achieving the delicate balance between performance and responsiveness in contemporary computing systems. Furthermore, novel architectures, such as RISC-V processors impose a set of special characteristics, in contrast to conventional architectures, e.g. they are usually deployed with weak memory consistency models [311]. A weak memory consistency model, as seen in RISC-V, allows for potential reordering of memory operations between processors, departing from strict program order. Even though this flexibility facilitates customization and optimization, it requires programmers to manage memory ordering explicitly using memory barriers when necessary.

Recent research has focused either only on providing dynamic memory management for Edge computing applications, such as [143,147,212] or on the architectural exploration on computing systems [77–81]. Authors of [77] investigate the impact of hardware technology configurations for three-level hybrid cache systems. In addition, authors of [78] introduce an exploration framework focusing on the microarchitecture configuration of the RISC-V BOOM processor, while in [79] a reinforcement learning-based approach is proposed for exploration. Moreover, authors of [80, 81], examine the effective coherence configuration for RISC-V.

Even though prior research has investigated the architectural exploration of various components, no work for the moment, according to our knowledge has explored the effective co-design and optimization of cache configurations via dynamic memory management for the dynamic data types in the application code. In this work, we propose a novel systematic exploration methodology for providing cache configuration exploration and optimization enhanced with dynamic memory management techniques over emerging architectures to provide architects with decision support. Our methodology envisions performance optimization while minimizing hardware overhead and cost. We integrate our approach into GEM5 [312] and Structural Simulator Toolkit (SST) [313], providing an end-to-end full system simulation that allows computer architects to explore discrete

configurations. Through our experiments, we highlight that: i) the co-impact of dynamic memory management and effective cache configuration leads to 61.1% execution time optimization, ii) the effective application's memory management leads to reduced hardware complexity and iii) we provide insights on the impact of application's organization to the cache configuration.

7.2.2. Exploration Methodology & Decision Support

Our proposed methodology provides decision support for optimizing systems performance through effectively configuring the cache hierarchy and applying dynamic memory management techniques on the examined application. The methodology envisions exploring the cache configuration over emerging architectures, aiming to support application-specific optimization. As input to our approach, we provide the target input source code, alongside the optimization objective(s). As input, we provide the target ISA configuration, i.e. focusing on the specific characteristics of the target system. This work envisions the adaptability of our proposed methodology to different computing platforms, thus ensuring its relevance and applicability across a spectrum of diverse computing scenarios.

First, at the application-level, we aim to provide source code level optimization for transforming static memory management to dynamic memory management 1a. This step strongly relies on our previous research conducted in [212]. In this work we extend the static to dynamic memory management by leveraging iterative exploration for transforming the application-level organization, based on the underlying cache configuration options. Thus, instead of deriving a single application version, we derive discrete application versions that will optimize performance for discrete cache configurations. Through this extension, the application level optimizations can be co-explored with the underlying cache configurations. Once the static to dynamic memory management is performed, the source-to-source transformations 1b are applied, thus leading to the optimized application versions. Furthermore, we integrate an application-level monitoring mechanism, which relies on in-code profiling, thus deriving the optimal application configurations 1c to be propagated as inputs to the next steps.

In addition to the input workload and the optimization objective(s), the target architecture to be configured is additionally provided. Based on the target architecture, the cache configuration options are defined, which will be utilized for exploration later on. We focus on providing exploration on the following cache configuration options: i) `cache size`, ii) `associativity`, and iii) `writeback policy` 2a. The aforementioned options are selected due to their high criticality in optimizing cache configuration and boosting

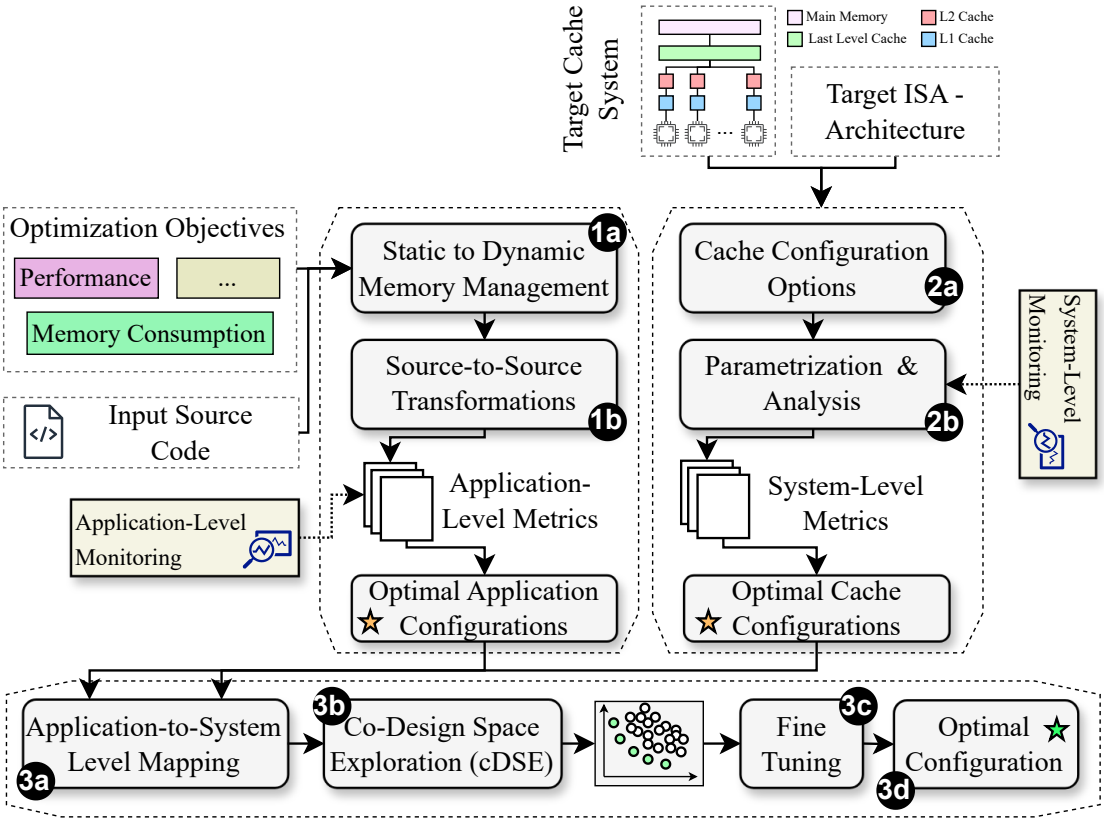


Figure 7.6.: Overview of the Proposed Methodology

performance [314]. By systematically conducting effective exploration on parametrization and analysis 2b of cache size, associativity, and writeback policy, the optimal configurations that enhance overall system performance can be pinpointed. Our methodology is enhanced with sophisticated run-time profiling and monitoring support, strategically gathering application-level and system-level metrics. These metrics, including Instructions Per Cycle (IPC) and cache hits/misses across the cache hierarchy, serve as crucial indicators for the exploration process. The diverse configurations undergo execution, and the profiling results are meticulously collected and propagated for further post-processing and in-depth analysis for deriving the optimal cache configurations 2c to be co-explored in the following steps alongside with the application-level parameters.

Next, the Design Space Exploration (DSE) phase follows 3b, where optimal cache configurations are systematically probed. Moreover, through leveraging fine-tuning procedures,

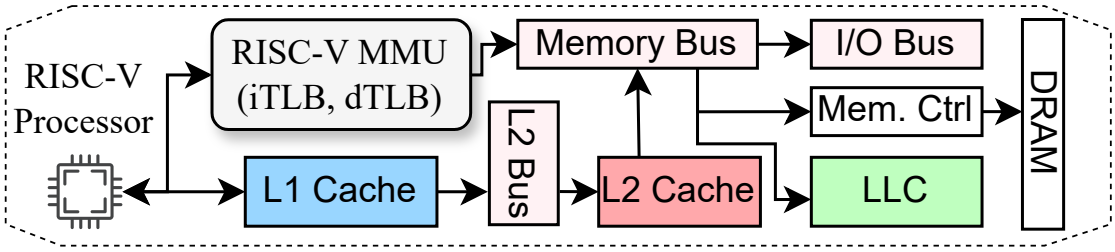


Figure 7.7.: Target RISC-V Architecture

we aim to iteratively refine these configurations, working towards deriving optimal solutions. This iterative refinement process ensures that the cache settings are precisely tailored, focusing on maximizing performance 3c. Throughout the methodology, careful consideration is given to potential trade-offs, balancing the pursuit of enhanced performance with mindful assessments of hardware complexity. The outcome of our methodology results in an optimal and fine-tuned architecture, precisely configured with the most effective cache settings for enhanced system performance 3d.

7.2.3. Experimental Evaluation

Technical Implementation and Experimental Setup The architecture of the target system is shown in Fig. 7.7. Our methodology is instantiated using the Gem5 simulator v22.1.0.0 [312] and the Structural Simulation Toolkit (SST) v12.0.1 [313] for simulating the architecture. These tools are effectively and accurately emulating complex architectures with system-level interactions and scalability. Python language and bash scripting are employed for the exploration. To minimize Operating System (OS) and driver-related overhead, our emulation relies on the Zephyr Real-Time OS (RTOS) [315] with POSIX support, ensuring a streamlined environment for precise exploration and analysis of cache configurations. Figure 7.7 illustrates the target architecture, while Table 7.3 summarizes

Table 7.2.: Overview of the design space specifications

Component	Exploration Specifications
L1 Cache	Size:{16,32,64,128}kB, Associativity:{1,4,8,16,64}
L2 Cache	Size:{8,16,32}MB, Associativity:{1,4,8,16,64}
Last-Level Cache (LLC)	Size:{16,32,48,64}MB, Associativity:{1,4,8,16,64}, Replacement Policy:{MRU, LRU, Random, LFU}

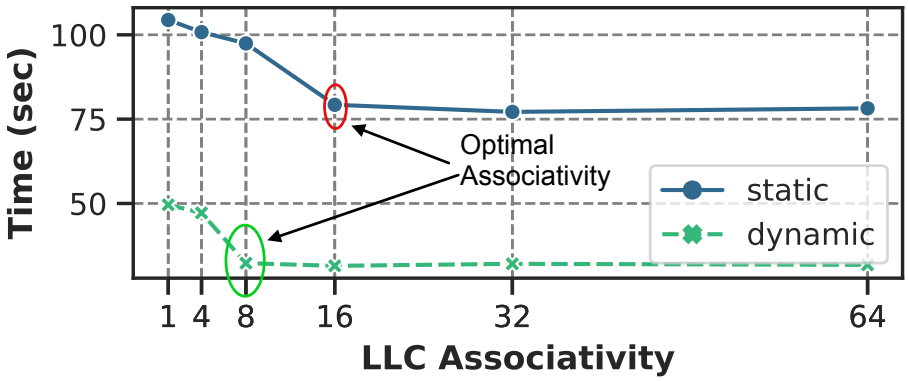


Figure 7.8.: Impact of static vs dynamic memory management on LLC cache configuration for the IoT biomedical application.

the cache configuration alternatives exploration space. For the LLC, apart from the size and associativity exploration, we integrate four replacement policy alternatives, namely i) Most Recently Used (`MRU`), ii) Least Recently Used (`LRU`), iii) `Random` and iv) Least Frequently Used (`LFU`). We need to state here that the replacement policy is strongly affected by the weak memory consistency model of RISC-V processors, thus special exploration is needed.

The simulated RISC-V system was deployed on a high-end server featuring an Intel Xeon(R) CPU E5-2699v3@2.30GHz and 16x16GB (256GB) DDR4 DRAM DIMMs, running on Linux 5.4.0-150-generic. The evaluation of our approach was conducted using a real-life IoT biomedical application for patient activity prediction, derived by IMEC-NL and detailed in Chapter 5 [212]. The specific application was incorporated into the experimentation process due to its initial static version and the transformed version of the application with highly dynamic memory management nature, thus opening the road for this research to investigate the impact of static and dynamic memory management to cache configurations.

Static vs. Dynamic Memory Management Conducting a comprehensive examination of cache configurations based on the unique characteristics of the application, we undertake a comparative analysis of two variants of the IoT biomedical application, delineated as i) employing static memory management and ii) utilizing dynamic memory management, as presented in [212]. Both iterations retain identical I/O functionalities, differing primarily in the upfront static allocation of data types versus the dynamic memory (de)allocations during runtime. Figure 7.8 showcases the execution time of the IoT biomedical application for various LLC associativity options. Our focus lies on in-

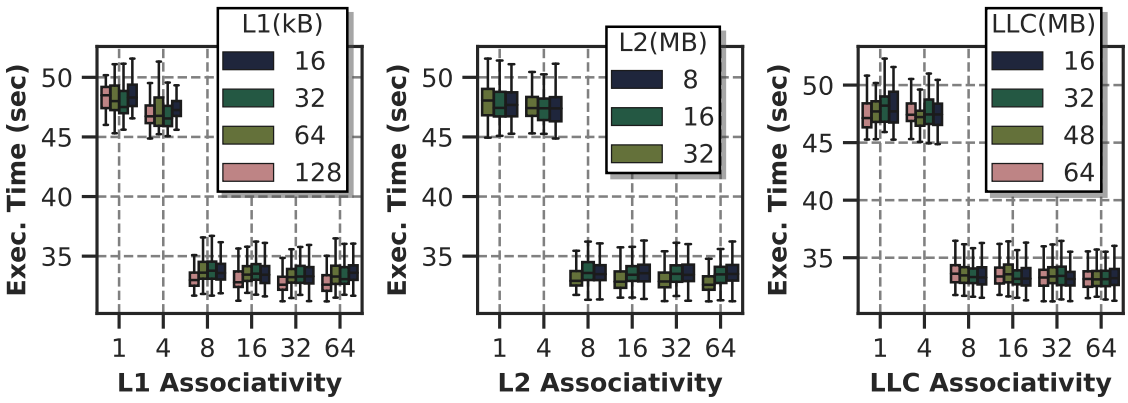
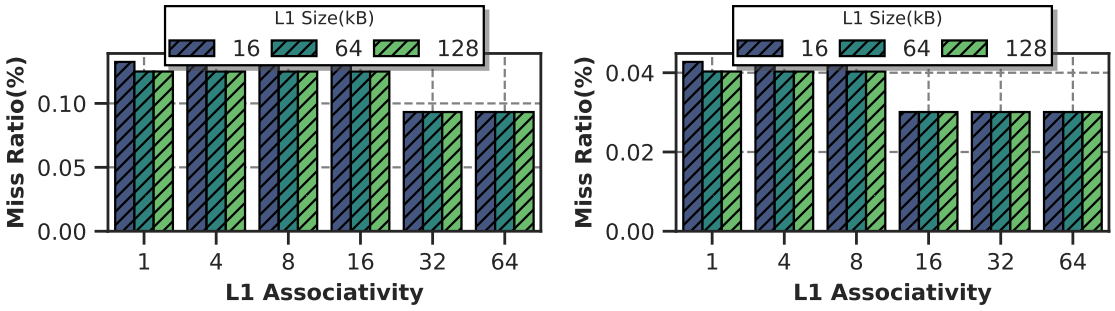


Figure 7.9.: Impact of the L1,L2 and LLC size and associativity configurations on the execution time for the dynamic version for the IoT biomedical application.

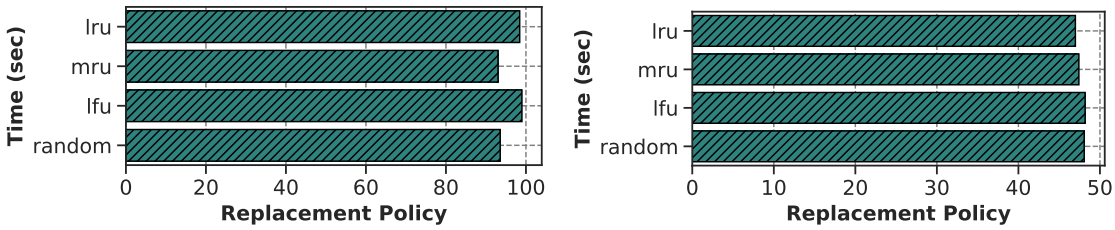
vestigating the influence of static versus dynamic memory management on associativity configurations, revealing that performance thresholds are met at associativity 16 and 8, respectively. Consequently, the higher-level code organization incurs lower hardware costs. Furthermore, the dynamic version exhibits a steeper curve compared to the smoother trajectory of the static one, underscoring that optimal configurations fail to significantly enhance performance. Through the dynamic approach, we achieve execution time optimizations of up to 48.7%, while the static approach yields optimizations of up to 22.1%, compared to the baseline. Thus, efficient application of dynamic memory management paves the way for further optimizations through hardware configuration and hardware/-software co-design.



(a) Static Mem. Management

(b) Dynamic Mem. Management

Figure 7.10.: Impact of Application-level static vs dynamic memory management and cache configurations to the L1 cache misses.



(a) Static Mem. Management **(b)** Dynamic Mem. Management

Figure 7.11.: Evaluation of cache replacement policies for static and dynamic memory management application versions.

Analysis and Evaluation of Cache Exploration First, we evaluate the impact of our methodology to provide effective performance optimization based on the exploration of various configuration options. Our experiments are conducted on the dynamic version of the IoT biomedical application. Figure 7.9 illustrates the execution time over different L1 (left), L2 (middle) and LLC (right) cache associativity and size configurations for the IoT biomedical application, respectively. The examined cache configurations vary from direct-mapped to fully associative cache. Our initial observation relies on the variation of the execution time based on the different configurations. We observe that the execution time can be further optimized by up to 31.1%, thus leading to improved performance. Furthermore, increasing cache associativity leads to increased cost and hardware complexity, thus a threshold should be defined where performance is optimized without skyrocketing the cost/complexity. The examination of cache associativity highlights a delicate balance between performance, cost, and hardware complexity. More specifically, we observe that for the examined benchmarks, a performance plateau is visible when cache associativity is equal to 8. Last, the cache size does not strongly affect the performance.

Moreover, a lower cache miss ratio corresponds to a reduction in the necessity to fetch data from slower levels of the memory hierarchy, thereby improving throughput and enhancing the overall responsiveness of the system. Thus, we delve deeper into how various configu-

Table 7.3.: Optimal Configurations

Component	Config. for Static Memory Management	Config. for Dynamic Memory Management
L1 Cache	16kB, **associativity 16**	16kB, **associativity 8**
L2 Cache	8MB, **associativity 16**	8MB, **associativity 8**
LLC	16MB, **associativity 16**, **MRU**	16MB, **associativity 8**, **LRU**

rations affect the L1 cache miss ratio percentage for both the static memory management (Fig.7.10a) and dynamic memory management (Fig.7.10b) versions of the application, as illustrated in Fig.7.10. Initially, we observe a consistent decrease in the cache miss ratio as the associativity increases for the IoT biomedical application, achieving reductions of up to 32.3% compared to the baseline configuration. Furthermore, in comparing the static and dynamic versions of the application, we find that the miss ratio percentage is, on average, 53.4% lower in the dynamic version. The optimal associativity for the static version is 16, while for the dynamic version, it is 8, aligning with the execution time observations in Fig.7.8. Similar trends and insights are derived for the L2 and LLC cache levels, underscoring the importance of associativity and memory management techniques in optimizing cache performance across various cache levels.

Furthermore, we investigate the impact of different replacement policies. Figure 7.11 illustrates the effect of the replacement policies both for the static and dynamic version of the IoT biomedical application. Our first observation relies on the fact that different replacement policy provides the optimal execution time for the static and dynamic memory management application versions. For the static memory management, the `mru` replacement leads to 13.3% execution time reduction compared to the other policies. For the dynamic version of the application, the selection of replacement policies contributes to performance enhancements ranging from 1.2% to 5.1% with the `lru` providing the optimal performance. The examined application is characterized by a higher prevalence of dynamic memory allocations. The dynamic nature of memory usage in this application introduces a higher probability of cache line evictions, thereby offering substantial overall system performance gains. Specifically, the frequent allocation and deallocation of memory objects can significantly influence cache behavior, highlighting the importance of efficient replacement policies. The expansion of our proposed methodology to multi-core systems and the investigation of different coherence protocols is left as future work. This underscores the importance of carefully considering cache coherence protocols alongside replacement policies to optimize performance in dynamic memory allocation scenarios. Exploring these aspects could unveil additional opportunities for improving system efficiency and scalability in complex, memory-intensive IoT applications. Overall, the effective co-design of the application's memory management and the cache hierarchy's configuration, as shown in Table 7.3, leads to 61.1% execution time optimization.

7.2.4. Conclusion

In this work, we propose a systematic exploration methodology for cache configuration over RISC-V architectures, aiming to provide decision support for performance optimiza-

tion. We show that the integration of dynamic memory management and different cache configurations can lead up to 61.1% performance optimization and we provide insights based on the application characteristics, while we investigate the impact of different replacement policies. Last, we provide insights on the impact of application's organization to the cache configuration. The outcomes of this work can be exploited by system architects who target effective co-design of application memory management and cache configuration.

Chapter 8.

Conclusions

In this chapter, we present the conclusions of this Ph.D. thesis. We summarize how the frameworks presented in this thesis advance beyond the state-of-the-art, we propose future extensions of the dissertation and we discuss the memory sub-systems of the future.

8.1. Thesis Summary

Today, we witness the rapid technological advancements towards the creation and transformation of a digital world, where data are stored, processed and analyzed. The effectiveness of recent systems strongly relies on the ability of the overall memory system to operate efficiently. The scope of this thesis is to examine the applicability of dynamic memory management techniques and propose solutions over distinct levels across the hardware-software continuum, aiming to provide performance, energy, and memory resources optimization. However, given the complexity as well as the huge number of available optimization knobs of modern memory systems, naive dynamic memory management policies are not able to provide optimal decisions. Moreover, aiming to follow the recent advances in heterogeneous memory technologies that are rapidly changing and the state-of-the-art computing platforms on the Edge-HPC continuum, in this dissertation we focus on optimizing memory management over different optimization layers.

In this thesis we examined the application of memory management solutions on the following optimization levels: i) application-level optimizations, ii) OS support, libraries and placement optimizations, and iii) hardware-level mapping optimizations. Given the inseparable and strong relationship between hardware and software memory components,

we enhanced the interoperability of different optimization layers and provided an end-to-end workflow and methodology.

Henceforth, we present a brief summary of our work and how the proposed frameworks tackle open-problems in the field.

- **Extensive Analysis, Classification and Guidelines of State-of-the-Art Data Placement over Heterogeneous Memory Systems**: We present an extensive survey for application-specific software data placement approaches that target heterogeneous main memory systems. Throughout the existing literature, we present the current state-of-the-art trends in software data placement, we highlight key design choices and features that affect decision making over various application domains. Through this extensive literature review, we indicate existing gaps in the literature, we present a qualitative comparison of different approaches and provide future guidelines and directions for extension of recent research and adaptation of the existing works in the upcoming main memory technologies,[ACM-CSUR under review].

- **Application-Level Memory Management and Source-to-Source Transformations for Python Workloads**: A novel solution for applying source-to-source transformations for Python applications aiming to support memory footprint optimization. This approach leverages a set of static (offline) and dynamic (online) analysis optimizations. Through this research, our ambition is to support developers and establish memory footprint-friendly solutions as the default in application development design and extend the range of target Edge computing devices that Python applications can be deployed on [143, 212].

- **Data Structure Refinement, Exploration and Placement over Heterogeneous DRAM/NVM Memory Systems**: In this level, we present a set of solutions for dynamic data type and indexes refinement, design place exploration and placement over heterogeneous DRAM/NVM memory system. We target multi-objective optimization, aiming to optimize and derive trade-offs among performance, energy consumption, and hardware wear. In this thesis a set of different application domains is targeted, varying from database workloads to ML-driven applications and in-memory Spark analytics. The scientific burden of this research aims to establish an overall rethinking of the data structure organization and placement over DRAM/NVM memory systems, derive key outcomes for different application domains, and boost fine-grained and coarse-grained data placement policies [160, 161, 240], [ACM TACO, IEEE TSuSC under review].

- **Hardware-Level Page Placement and Design**: An LSTM-based framework for page placement over heterogeneous DRAM/NVM memory systems and memory management boost through effective cache configuration. In the scope of this thesis, the LSTMs are utilized for memory accesses prediction and hot/cold pages classification, while taking advantage of historical behavior of neighboring pages in the memory. Through this research, the objective is to re-design existing page placement policies of recent systems, aiming to provide performance improvements and energy savings. Last, the impact of cache configurations is exploited over memory management, aiming to open the research path for full end-to-end memory-cache management co-design [190], [IEEE ESL under review].

8.2. Memory Sub-systems of the Future

As mentioned in Section 2.1.2 the advancements of memory technologies are composing novel memory architectures, which are expected to further transform over the next years. Aiming to surpass the limited bandwidth and capacity imposed by existing memory technologies and interfaces, new technologies are expected to come forward. Among the examined alternatives, the Compute eXpress Link (CXL) [316] has emerged as one of the most promising memory interface technologies. The CXL forms an open standard, built on the standard PCIe, which is a serial interface. The CXL offers higher bit-transfer rates per pin and consumes less energy per bit-transfer but with higher link latency. Compared to the PCIe, the CXL provides extra features, enabling the communication of the CPU with devices and their connected memory, using load/store operations in a cache-coherent manner. Furthermore, the CXL enables the decoupling of memory technologies from a specific memory interface, thus giving the memory manufacturers further flexibility on the design of memory technologies. The aforementioned advantages have led major hardware vendors and manufacturers to announce CXL support on their product roadmaps [317–319]. Both the latest 4th-generation Intel Xeon Scalable Processor (Sapphire Rapids) and the latest 4th-generation AMD EPYC Processor (Genoa) are among the first server-class commodity CPUs to support the CXL 1.1 standard [320]. Figure 8.1 illustrates an abstract view of novel CXL memory module architectures [320].

However, Moore's Law has been slowing over the last few years [321], while the amount of data being processed is increasing significantly and, even though the next generation of memory is likely to help accelerate data movement, it may simply shift the bottleneck to a different place within a system [322]. Such kind of limitations lead to an overall rethinking of the state-of-the-art systems. Thus, system architects need to rethink

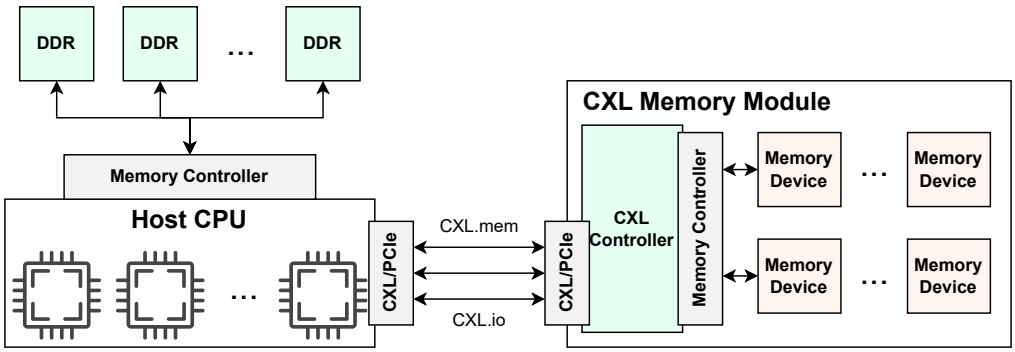


Figure 8.1.: CXL Memory Module Architecture

from the beginning system architectures to address major bottlenecks that are imposing limitations on performance and power efficiency. Traditional data center designs address scalability issues by increasing the number of resources, having up to thousands of servers, and storing data across multiple racks. Such kind of approach leads to low CPU utilization, high access latency, reduced power efficiency, and increased Total Cost Ownership (TCO) [323, 324]. Aiming to surpass such limitations, Rack Scale Architectures have been introduced [325]. In contrast to legacy architectures, they offer the ability to group processing, memory and storage resources, allowing them to be scaled independently as needed. Through this approach a single rack can contain a mix of resources, in terms of compute, memory, and storage, while others can provide a different mix, therefore resulting in reduced bottlenecks for varying workloads with improved performance, power efficiency, and TCO.

Moreover future computing systems are anticipated to heavily leverage 3D technologies [326, 327] and chiplets [328]. By employing 3D stacking techniques, memory components can be vertically integrated, leading to increased memory density within a smaller physical footprint. Additionally, chiplet-based memory solutions allow for modular designs where memory dies can be developed independently and then seamlessly integrated into the overall memory subsystem. This approach enables greater flexibility in memory configurations, facilitating the creation of customized memory architectures tailored to specific performance and capacity requirements. Furthermore, the use of chiplets promotes scalability and reusability, as standardized memory modules can be easily integrated into diverse computing systems. Overall, the adoption of 3D technologies and chiplets in memory subsystems promises to enhance memory capacity, performance, and efficiency, driving forward the evolution of future memory-centric computing platforms.

Finally, near-memory [329–331] and in-memory [332–334] computing paradigms are ex-

pected to play significant roles in enhancing processing efficiency and reducing data movement bottlenecks. Near-memory computing involves integrating compute units directly adjacent to memory modules, minimizing the distance data needs to travel for processing. This proximity accelerates computation and reduces energy consumption by mitigating the latency and power overheads associated with data movement between memory and processing units. Similarly, in-memory computing leverages the vast bandwidth and low-latency access of main memory to perform computational tasks directly within memory, eliminating the need for data transfer to and from separate processing units. These approaches are poised to revolutionize data-intensive workloads by offering orders of magnitude improvements in performance and energy efficiency. Moreover, future systems are anticipated to embrace disaggregation [335, 336], where compute, storage, and memory resources are physically separated and interconnected through high-speed networks. This disaggregated architecture enables flexible resource allocation and scalability, allowing for dynamic provisioning of computing resources based on workload demands. By decoupling computation from memory and storage, disaggregated systems offer greater agility, resource utilization, and cost-effectiveness, paving the way for the next generation of scalable and efficient computing infrastructure.

8.3. Future Extensions

The presented research work has showed that dynamic memory management forms a very promising solution for optimizing performance, energy and hardware wear over heterogeneous memory systems. We showed that dynamic memory management can be effectively applied in multiple levels of the computing stack, ranging from the application/source code level to the mapping over the real hardware. Moreover, given the rapid growth and expansion of next generation HPC systems consisting of heterogeneous memory systems and the new requirements, both from a technical as well as a societal perspective, that constantly appear over the years, there are still open and novel issues to be addressed in future efforts. All the major HPC providers, European networks of researchers (e.g., HiPEAC), as well as Europe itself, have already forged the path for the design of future memory systems. This path builds upon four major pillars, discussed below.

8.3.1. Sustainable and Green Computing

In the landscape of future computing systems, the adaptation of green and sustainable computing requires innovative memory management solutions. As the demand for com-

putational power continues to escalate, addressing the energy consumption of memory subsystems becomes increasingly critical. While CPUs have made strides in energy efficiency and performance scaling, the memory wall persists as a formidable challenge. Despite advancements in processor technologies, the widening gap between CPU speeds and memory access times underscores the urgency of optimizing memory management strategies. By focusing on green computing principles, future memory management solutions must aim to minimize energy consumption while maximizing performance and scalability. This entails exploring novel approaches such as energy-aware memory allocation, dynamic voltage and frequency scaling, and intelligent memory prefetching techniques. Embracing sustainable memory management practices not only reduces the environmental footprint of computing systems but also enhances their overall efficiency and longevity. As we strive towards more energy-efficient computing architectures, the optimization of memory systems remains paramount in realizing the vision of a greener and more sustainable digital future.

8.3.2. Novel Hybrid Memory Architectures and Technologies

Hybrid Memory Architectures are anticipated to integrate a diverse array of memory technologies beyond traditional DRAM and Non-Volatile Memory (NVM). Emerging memory technologies such as Magnetoresistive RAM (MRAM) [27], Ferroelectric RAM (FeRAM) [28], and Conductive-Bridging RAM (CBRAM) [337] hold promise for inclusion in future memory architectures. These technologies offer unique advantages such as non-volatility, low power consumption, high endurance, and fast access times, making them attractive candidates for hybrid memory solutions. Moreover, photonics-based memory technologies, such as Optane-based memory [29] and Photonic RAM (PRAM) [338], are gaining traction for their potential to enable ultra-high-speed data transfer and reduce power consumption. The integration of these diverse memory technologies presents new opportunities and challenges for dynamic memory management solutions. Dynamic memory management algorithms will need to account for the varying characteristics and access latencies of different memory technologies, dynamically optimizing data placement and movement to maximize system performance and energy efficiency. Additionally, emerging interconnection technologies, such as high-speed optical interconnects [339–341] and silicon bridges [342], will play a crucial role in facilitating efficient communication between different memory tiers within memory architectures, enabling high-bandwidth, low-latency data transfers and minimizing overhead. As hybrid memory architectures continue to evolve and incorporate a wider range of memory technologies, dynamic memory management solutions will play a key role in orchestrating the efficient utilization of these resources to meet the demands of future computing workloads.

8.3.3. Memory Disaggregation and in-memory Computing

In the future, memory disaggregation is expected to become a widely adopted paradigm in data center architectures, offering significant advantages in resource utilization, flexibility, and scalability. To support effective memory disaggregation, dynamic memory management mechanisms should adapt to the unique challenges and requirements of disaggregated memory environments. Dynamic memory management solutions should be designed to optimize memory allocation, migration, and access patterns across disaggregated memory pools, considering factors such as network latency, bandwidth, and memory locality. Robust memory virtualization techniques are essential to ensure seamless abstraction and provisioning of memory resources to individual servers or nodes. Additionally, intelligent memory placement algorithms can help minimize data movement overhead and optimize data locality, improving overall system performance and efficiency in disaggregated memory architectures. As memory disaggregation becomes more dominant, dynamic memory management will play a crucial role in enabling efficient resource utilization and maximizing the benefits of disaggregated memory environments.

Appendices

Appendix A.

DMRM: Distributed Market-Based Resource Management on Edge Computing Systems

Resource management is a key technique for efficiently operating devices in Internet of Things (IoT). In this chapter, we propose DMRM, a new algorithm based on economic and pricing models for dynamic resource management of IoT networks under CPU, memory, bandwidth and latency constraints. We use a supply and demand model, smart data pricing and perceived valued pricing, implementing a marketplace where IoT devices and Gateways buy and sell computing and communication resources necessary for task execution. Our new market-based algorithm is compared to relevant approaches showing that it not only reaches near-optimal results, but also, its scalable, distributed nature leads to three orders of magnitude lower execution requirements compared to centralized approaches.

This appendix text strongly relies on the following publications:

- *Katsaragakis Manolis, Masouros Dimosthenis, Tsoutsouras Vasileios, Samie Farzad, Bauer Lars, Henkel Jorg., & Soudris Dimitrios (2019, March). DMRM: Distributed market-based resource management of edge computing systems. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1391-1396). IEEE. [343]*

A.1. Introduction

Internet of Things (IoT) refers to a self configuring, adaptive and complex network of objects that surround us, extending Internet connectivity beyond traditional devices, such as desktops, laptops, smart-phones and tablets, to any physical devices and everyday objects. The rapid growth of IoT has exploded the number of devices connected to the Internet, a number which keeps increasing in high pace. This has led to an enormous amount of collected data which, in the traditional cloud-centric approach, has to be offloaded to the cloud infrastructure for the information inference [344].

However, cloud servers and datacenters are geographically centralized, situated far from the end devices and as a consequence, real-time and latency-sensitive computation services often endure large round-trip delay, network congestion and service quality degradation [345]. Moreover, in data-sensitive domains such as healthcare, the storage of data on third-party infrastructure has raised privacy and confidentiality concerns [346]. To address the aforementioned issues, the Edge computing paradigm proposes to push the computation and data inference to the Edge of the IoT network, in order to alleviate the dependency on cloud infrastructure and preserve the privacy of sensitive personal data.

The presented work regards the well-established Edge computing architecture of IoT nodes and Gateways, where a portion of the tasks of the IoT nodes has been mapped for execution at the Edge and is/can be offloaded to the IoT Gateway [347]. Resources including available CPU, memory and communication bandwidth of IoT nodes and Gateways are limited and a portion of them is shared. Thus, an efficient resource management mechanism is required to dynamically allocate the shared Gateway resources and to designate the operating configuration of IoT nodes. The essential tuning knob of the system is the designation of the tasks to be offloaded to the Gateway, an NP-complete problem which has already been addressed using centralized decision making on the Gateway [347].

The main goal of the presented work is to take advantage of the inherently distributed nature of Edge computing in order to avoid the pitfalls of centralized decision making, i.e. increased execution latency and lack of performance scalability. **The novel contributions of this work are as follows:**

- We present an analytical model of an Edge computing setup, where local tasks can be offloaded from the IoT nodes to the Gateway. The model captures the properties and constraints of the devices and formulates the system objective as a deadline miss minimization problem.

- We design and implement DMRM, a distributed, market-based algorithm for Edge computing resources management, which employs economic and pricing theory in order to minimize task deadline misses.

- An evaluation of DMRM is presented, highlighting its ability to designate near-optimal solutions, while achieving orders of magnitude lower execution latency and scalable performance compared to centralized approaches.

The rest of the appendix is organized as follows. Section C.2 presents related work, while in Section B.3 the system architecture, the characteristics of the devices and the optimization problem are formulated. The proposed market-based solution is detailed in Section A.4. Section D.4 provides the experimental and comparative evaluation of the proposed resource management scheme, while Section D.5 concludes the appendix.

A.2. Related Work

Several works have been conducted on resource management on Edge computing. Authors of [347] address the problem of QoS management for IoT devices under bandwidth, battery and processing constraints. They propose a centralized, pseudo-polynomial solution based on a dynamic programming approach, which enables reusing pre-computed solutions. In a similar setup, the authors of [348], solve the task assignment problem using stochastic optimization, taking also into account the mobility of devices in the IoT network. In [349] the problem of computation offloading on mobile edge computing is tackled using queuing theory.

With respect to economic theory, authors of [350] propose a distributed power management framework for heterogeneous multi-cores, based on price theory, aiming at minimizing power consumption, while satisfying performance goals under a power budget constraint. Furthermore, in [351] authors present a power budget economy for performing thermal management in many-core architectures, in order to reduce high peak temperatures. In this work, many distributed Market agents are present, serving the bids from multiple Local thermal agents. Additionally, [352] proposes a combinatorial auction-based service provider selection in Mobile Edge Computing Networks (MECs). The authors study the computation offloading, where MEC Service Providers are equipped with limited wireless and computation resources. An auction is utilized and the users' bid strategies are designed based on multi-round priority rule and the winner determination process is formulated as a two-dimensional Knapsack problem.

In [353], authors design a device-to-device offloading framework that integrates a distributed incentive scheme and a distributed reputation mechanism. In [354] Agora is presented, a resource management framework based on market principles, which deals with unavailable resources and distributes the computation burden among the chip's resources. The allocation of jobs to resources is decided by the jobs' offering price. The offering price is based on their waiting time and the price that the resources are willing to accept is based on their utilization under their current state. Additionally, the authors of [355] propose several pricing, game theory and auction-based models that can be applied in IoT challenging issues and authors in [356] discuss the data offloading problem and implement an auction-based mechanism. Finally, Wong et al. [357] studied a variant of resource allocation problem at the edge of network, to control network usage from the devices.

In overview of the literature, economic models have been already used for resource management allocation purposes but have not been applied to a setup of IoT devices competing for limited, shared resources, under deadline restrictions. In addition, we distinguish from other works by introducing a more flexible, per-application awareness where the price of the resources is defined by the willingness of each distributed agent (device) to pay instead of letting the resource seller de facto define the selling price. Using this concept, we are able to develop a distributed market place using fundamental pricing models instead of auction based approaches.

A.3. System and Problem Formulation

Our target architecture envisions a system of n IoT nodes, one Gateway device and a number of tasks per device, which can be executed either on the Gateway or on the IoT node, as shown in Fig. A.1. The IoT nodes are resource constrained as far as their CPU, memory capacity and communication capabilities are concerned, and can offload their tasks to the Gateway device. Considering the communication of the devices, IoT devices can be connected with any wired or wireless communication protocol supported by the Gateway.

Each IoT device is uniquely defined by a value $i \in \{1, ..., n\}$ and is specified by the tuple $D_i = \{C_i, M_i, B_i, Money_i, \mathcal{T}_i\}$. Variables C_i, M_i, B_i denote the CPU, memory and bandwidth resources of device i, respectively. $Money_i$ denotes the total money of the device and $\mathcal{T}_i$ is the set containing all the tasks τ_j^i of device i. For every task, two auxiliary variables characterize its execution. For the task j of device i, o_j^i is set to 1 if the task is executed on the Gateway or set to 0, otherwise. Similarly, l_j^i is set to 1 if the

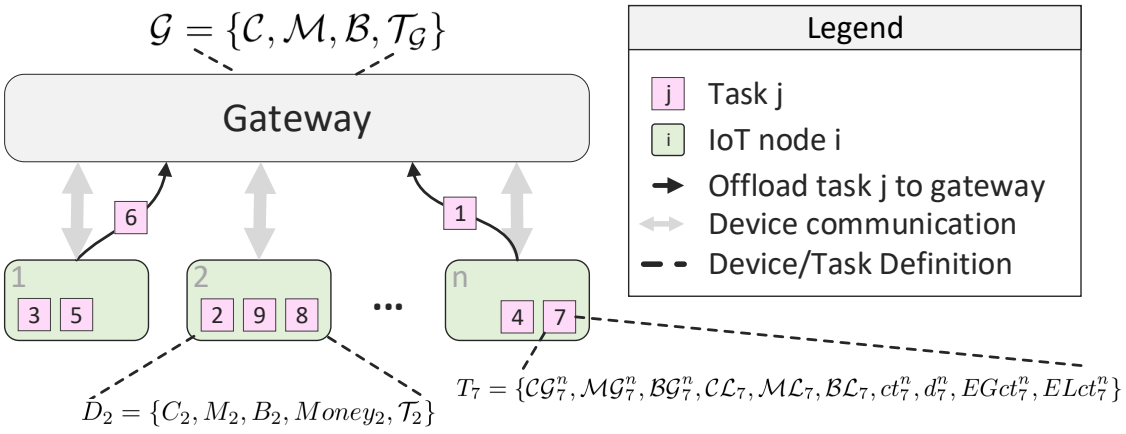


Figure A.1.: Target system architecture

task is executed locally or set to 0, otherwise. The case that both variables are set to zero means that execution of the task has been postponed.

On the Gateway, the IoT devices can acquire resources in order to have their tasks executed. Similar to the IoT devices, the Gateway is defined by a tuple $\mathcal{G} = \{\mathcal{C}, \mathcal{M}, \mathcal{B}, \mathcal{T}_\mathcal{G}\}$. Variables $\mathcal{C}, \mathcal{M}, \mathcal{B}$ denote the CPU, memory and bandwidth resources of the Gateway, respectively. $\mathcal{T}_\mathcal{G}$ is the set of tasks offloaded from the IoT devices to the Gateway, at any moment. Every task of device i is defined by its own unique id j and is specified by the tuple:

$$T_j = \{\mathcal{CG}_j^i, \mathcal{MG}_j^i, \mathcal{BG}_j^i, \mathcal{CL}_j, \mathcal{ML}_j, \mathcal{BL}_j, ct_j^i, d_j^i, EGct_j^i, ELct_j^i\} \tag{A.1}$$

where $\mathcal{CG}_j^i, \mathcal{MG}_j^i, \mathcal{BG}_j^i$ denote the utilization requirements of task j if executed on the Gateway and $\mathcal{CL}_j, \mathcal{ML}_j, \mathcal{BL}_j$ the respective requirements if executed locally. The variable ct_j^i is the time that a task completes its execution, while d_j^i is its associated deadline. Variables $EGct_j^i$, $ELct_j^i$ also indicate the estimated execution time of each task of device j on the Gateway and locally, respectively.

The key parameters of the system model are also summarized in Table A.1. The optimization objective of the system as expressed in Eq. B.2, is to minimize the number of tasks of IoT nodes that exceed their deadline. The task arrival trend is unknown, since IoT nodes can be mobile and dynamically spawn new tasks. The solution is constrained by the available CPU, memory and bandwidth resources both in IoT nodes and Gateway (Eq. B.4 to B.8). In this work, power constraints have not been examined, but the problem formulation and solution allows for their seamless integration. The specified requirement of *Unique Execution* in Eq. B.9 means that a task is prohibited to be

Table A.1.: Key system model parameters

Denotation	Description
τ_j^i	task j of device i
o_j^i	1 if task j of device i is executed (offloaded) to Gateway, 0 otherwise
l_j^i	1 if task j of device i is executed locally, 0 otherwise
$\mathcal{C}$, $\mathcal{M}$, $\mathcal{B}$	CPU, Memory and Bandwidth resources of Gateway
C_k, M_k, B_k	CPU, Memory and Bandwidth resources of device k
$\mathcal{CL}_j$, $\mathcal{ML}_j$, $\mathcal{BL}_j$	CPU, Memory and Bandwidth resources required for task j executed locally
$\mathcal{CG}_j^i$, $\mathcal{MG}_j^i$, $\mathcal{BG}_j^i$	CPU, Memory and Bandwidth resources required for task j of device i executed on Gateway
ct_j^i, d_j^i	Completion time and deadline of task j of device i
$EGct_j^i$, $ELct_j^i$	Estimated completion time of task j of device i on Gateway and locally, respectively

executed both on an IoT node and Gateway, while the postponing of a task i remains a valid choice. Our proposed market-based solution aims at solving the optimization problem by dynamically designating where each task of the IoT nodes will be executed on.

$$\text{minimize} \quad N_{missed} = \sum_{\forall i}(\sum_{\forall j} miss(\tau_j^i)) \tag{A.2}$$

$$\text{where} \quad miss(\tau_j^i) = \begin{cases} 0 & if\ ct_j^i \leq d_j^i \\ 1 & otherwise \end{cases} \quad \text{s.t.:} \tag{A.3}$$

Gateway $$\sum_{\forall i}(\sum_{\forall j} o_j^i \cdot \mathcal{CG}_j^i) \leq \mathcal{C} \qquad \textit{CPU} \tag{A.4}$$

constraints: $$\sum_{\forall i}(\sum_{\forall j} o_j^i \cdot \mathcal{MG}_j^i) \leq \mathcal{M} \qquad \textit{Memory} \tag{A.5}$$

$$\sum_{\forall i}(\sum_{\forall j} o_j^i \cdot \mathcal{BG}_j^i) \leq \mathcal{B} \qquad \textit{Bandwidth} \tag{A.6}$$

Device k $$\sum_{\forall j} l_j^k \cdot \mathcal{CL}_j \leq C_k \qquad \textit{CPU} \tag{A.7}$$

constraints: $$\sum_{\forall j} l_j^k \cdot \mathcal{ML}_j \leq M_k \qquad \textit{Memory} \tag{A.8}$$

$$\sum_{\forall j} l_j^k \cdot \mathcal{BL}_j \leq B_k \qquad \textit{Bandwidth} \tag{A.9}$$

Unique execution $$o_j^k + l_j^k \leq 1, \forall j \in \mathcal{T}_k \tag{A.10}$$

A.4. Distributed Market-Based IoT Management

A.4.1. Background and Main Proposed concepts

The key idea of the proposed algorithm is to create a distributed marketplace, where buyers and sellers interact with each other and compete for shared resources. The IoT devices act as buyers, who demand resources according to the virtual money in their possession. The Edge Gateway is the seller of the market, who supplies its computing, memory and bandwidth resources at a specific price. The advantage of this approach is that (i) decision making burden is distributed to numerous agents and (ii) each IoT node is empowered with the ability to specify the priority and importance of its tasks. The downside of these attributes is that the final decision maker (in our case the Gateway) may have knowledge of only a subset of the IoT tasks. This might lead to sub-optimal scheduling decisions and this explains our meticulous design of the reasoning in the IoT nodes, to avoid this pitfall. The following fundamental pricing and economic models are used.

Supply and Demand model, which postulates that, holding all else equal, in a competitive market, the unit price for a particular good or commodity will vary until it settles at a point where the quantity demanded equals the quantity supplied, resulting in an economic equilibrium.

Consumer Perceived Value Pricing, where in order to maximize long-term profits, a seller needs to set the price by considering the buyer's perceived value from the commodity or the service rather than using traditional costs. The buyer's perceived value is the overall benefit derived at the price that the buyer is willing to pay.

Smart Data Pricing, where buyers are charged according to their access rate on resource usage. If the price is set based on usage, a fair and efficient use of resources is promoted.

Consumer perceived value pricing and smart data pricing are combined to create a pricing mechanism. In particular, the bidding price for a resource is set by the buyers (IoT nodes) according to their estimated resource usage each time. The final decisions are made on the Gateway, where all bids are accumulated. There, the supply and demand model is used to reach an equilibrium between the demanded and supplied resources, while maximizing the Gateway profit and minimizing the deadline misses of the IoT tasks.

To achieve the aforementioned goals, *DMRM* is introduced, composed of four core mechanisms divided among IoT nodes and Gateways. *Money Distribution* and *Task Selection* mechanisms are executed on the IoT Gateway, while the *Offloading* and *Bidding* mechanisms are executed on each IoT node. The combination of these mechanisms via data exchange between IoT nodes and the Gateway completes DMRM leading to its outcome, i.e. the decision of where each task of the IoT nodes will be executed, either on the Gateway or locally.

It is important to note the following advantage of the market-based approach to resource management in Edge computing systems. With the continuous integration and adoption of Edge computing by commercial enterprises, it is foreseen that the provided services will follow a business model similar to the one of cloud resources. According to this model, the services and resources of an IoT Gateway, will be available for sale according to the specific needs of each customer. Consequently, a market-based resource management model can be seamlessly customized to such a business model, since pricing of resources is the principle idea.

A.4.2. Decision making functions of DMRM on the IoT node

The portion of DMRM executed on an IoT node is composed of two critical parts, i.e. the *Task Offloading* and *Bidding* mechanisms. The *Task offloading* mechanism defines whether a task will be executed locally or offloaded to the Gateway, with the aim to minimize the number of the tasks that exceed their deadline. To achieve that, for every single task a metric is defined that estimates the probability of the specific task to miss its deadline. We introduce this metric as **sensitivity (s)** and for the task τ_j^i of device i it is defined as (Section B.3):

$$s_{\tau_j^i} = ((d_j^i - EGct_j^i - r) + (d_j^i - ELct_j^i - r))/2 \tag{A.11}$$

where r is the current execution moment (measured in rounds) of the system. The lower the sensitivity, the more likely it is for the task to outrun its deadline. The tasks are sorted in ascending sensitivity order and for each task the IoT node decides whether to (1) offload the task to the Gateway, (2) execute the task locally or (3) suspend the execution of the task for future execution on the Gateway or on the IoT device.

We assume, that the tasks examined at each round are those that have not been executed yet and assume no knowledge about future tasks. In other words, the decisions made at each round depend only on the past and present condition of the device.

According to the following criteria, each device decides if a task will be offloaded or not.

- If an IoT device does not possess the required resources of the task then it makes an offload request to Gateway.
- If an IoT device has the required resources, but the estimated offloading latency is lower than the estimated local execution latency, then it makes an offloading request.
- Otherwise, the IoT device executes the task locally, or postpones its execution.

For those tasks that are decided to be offloaded a bid should be defined via the *Bidding* mechanism, in order for the IoT node to buy the required resources from the Gateway.

The *Bidding* mechanism is crucial for the efficient operation of the market-based approach as it defines the bidding that each IoT device is willing to pay for the requested resources. By design it cooperates with the *Money distribution* mechanism so that the selling price of the resources is adapted to the total available money of the market and is affordable by the majority of the IoT devices. This assures that resource prices and biddings are not prohibitively high, so that devices are able to pay for those resources, and not extremely low, thus creating a competitive marketplace.

Each IoT node calculates the biddings for its tasks and the final bid and task details are sent to Gateway. The actual bidding for a specific task is calculated according to its required resources on the Gateway, its sensitivity (favoring the tasks that are close to their deadlines) and the current bidding capabilities of the IoT device. The bidding calculation function for task τ_j^i is provided in Eq. A.12.

$$B_{\tau_j^i} = \begin{cases} \mathcal{CG}_j^i + \mathcal{MG}_j^i + \mathcal{BG}_j^i - s_{\tau_j^i} * b, & \text{if } s_{\tau_j^i} \leq 0 \\ \mathcal{CG}_j^i + \mathcal{MG}_j^i + \mathcal{BG}_j^i + \frac{1}{s_{\tau_j^i}} * c, & \text{if } s_{\tau_j^i} > 0 \end{cases} \tag{A.12}$$

Values are normalized in the range of [0-100]. Constants b and c are set to 1000 and 800 and have been defined according to extensive system simulation. The estimated bidding is compared to the total money $Money_i$ of device i and:

1. If $B_{\tau_j^i} \leq Money_i$ then the task is offloaded to Gateway with the estimated bidding.

2. Otherwise, if $B_{\tau_j^i} > Money_i$, the IoT device cannot afford the bidding price that has been calculated Eq. A.12. Then, the task is sent with bidding price equal to $Money_i$, i.e. the device bids all its remaining money.

Algorithm 4: DMRM functionality on IoT nodes

```
   Data: Gateway, IoT device Tuple, Set of Tasks
 1       volatile curRound /* Round updated at background */
 2       RemTasks = checkTasks(Tasks, curRound) /* Remaining Tasks */
 3       while length(RemTasks) > 0 do
 4           /* Invoke Task Offloading Mechanism */
 5           oTasks = offloading(Tasks, DevTuple, curRound)
 6           /* Invoke Bidding Mechanism */
 7           bids = bidding(oTasks, Money)
 8           /* Send offloading proposition to Gateway */
 9           sendTasksGateway(oTasks, bids)
10           /* Wait for answer */
11           answer = waitAnswers(Gateway, offloaded)
12           for t in oTasks do
13               if answer(t) = "Accept" then
14                   payGateway(Money) /* Pay Gateway */
15               task else
16                   RemoveFromList(t, oTasks)
17           RemTasks = checkTasks(Tasks, curRound)
```

Algorithm 4 summarizes the DMRM software functionality on an IoT node, which possess an amount of *Money* and a set of *Tasks* to be executed under specific deadlines. It specifies the interaction of the *Task Offloading* and *Bidding* mechanisms, as well as how bids and responses are transmitted to and from the Gateway and appropriately handled.

A.4.3. Decision making functions of DMRM on the Gateway

The portion of DMRM executed on the Gateway is dominated by two parts, i.e. the *Money Distribution* and *Task Selection* mechanisms. The *Money Distribution* mechanism, ensures that the IoT devices will not run out of money by frequently re-distributing money to them in a fair way, thus ensuring the existence of a competitive market. The amount of extra money distributed to each IoT node is calculated according to its CPU, memory and bandwidth resources, the respective resources of the Gateway and the number N of active IoT nodes.

This mechanism is activated every time the total money of all N devices connected to the

Gateway reach a minimum threshold. This threshold is dynamic in the sense that it takes N into account, i.e. varying number of connected devices. Money redistribution is crucial for a system, where there is no means for the IoT nodes to earn more money. In an actual price-based Edge computing system, this mechanism would not be necessary as each customer would have the obligation to renew its money reserve.

The Gateway receives all biddings from the IoT nodes and must decide which will be accepted via the *Task Selection* mechanism. This mechanism combines the profit maximization of the Gateway with the minimization of the delayed tasks. From Eq. A.12 it follows that the lower the sensitivity is, the higher the bidding price is. Similar to that, the higher the bidding price is, the higher the profit gained by the Gateway is. As a result, if the tasks with higher bids are chosen to run first, the system approaches an equilibrium between the maximization of profit of the Gateway and the minimization of delayed tasks of the IoT nodes.

Based on the bids for each task (higher to lower), the *Task Selection* mechanism examines whether its available resources are enough to satisfy the requirements of the task and, if they are, it consents or rejects the offloading. This process is repeated for all the received tasks and the outcome is transmitted back to the corresponding devices. If a task was chosen to be executed, then the IoT device is charged according to the bid made. Otherwise, if a task is rejected then the IoT device is not charged and it is a responsibility of the IoT node to decide its execution fate.

Algorithm 5 summarizes the DMRM software part of the Gateway. It is executed as long as there are active connected devices to Gateway. The *Money Distribution* is executed conditionally according to the total amount of money of the IoT nodes. The *Task Selection* function is executed when the Gateway has received new bids by the IoT nodes and then the nodes are informed accordingly.

A.5. Experimental Evaluation

A.5.1. Experimental Setup

DMRM is implemented using C programming language and executed on various contemporary embedded devices with increasing computational capabilities, which represent diverse design choices of a Gateway device. Specifically, the utilized devices are Intel Galileo Gen 1 at 400 MHz, 256 MB RAM, Raspberry Pi 3 Model B with 4 Cortex-A53

Algorithm 5: DMRM functionality on the Gateway

```
   Data: Gateway, IoT devices' Tuples
1        volatile curRound /* Round updated at background */
2        totalMoney = 0 /* Initialize Money */
3        /* Determine Money re-distribution threshold */
4        threshold = determineT(NDevs)
5        while ActiveDevs(DevTuples) > 0 do
6            if totalMoney < threshold then
7                /* Invoke Money Distribution Mechanism */
8                totalMoney += moneyD(NDevs, DevTuples)
9            /* Wait for offers */
10           taskBids = waitForBids(NDevs, DevTuples)
11           /* Invoke Task Selection Mechanism */
12           sTasks = taskSelection(taskBids)
13           /* Reply to IoT nodes */
14           sendAnswers(sTasks, NDevs, DevTuples)
```

CPUs at 1.2 GHz, 1 GB of RAM and Nvidia Tegra X1 with 4 ARM Cortex-A57 processors running at 1.9 GHz and 4 GB of RAM.

We consider a variety of scenarios where IoT devices and tasks are randomly selected. The resources of the devices and the required resources of the tasks are randomly sampled from Normal or Poisson distributions in the range of [0-100]. In this way, we can construct diverse scenarios of CPU intensive, memory intensive and bandwidth intensive tasks and devices.

A second important experimental parameter is the arrival time and density of incoming tasks at IoT nodes. We focus our attention on two distinct cases. In the first one, created using the Normal distribution, tasks are arriving almost simultaneously, thus creating a huge demand for resources. The second scenario involves cases, where simultaneous high peaks of demand are avoided and Poisson distribution is selected for this goal. Last, assuming that the arrival time of task τ_j^i is Ar_j^i, we calculate its deadline according to Eq. A.13.

$$d_j^i = C_a * ((EGct_j^i + ELct_j^i + 2 \cdot Ar_j^i))/2 \tag{A.13}$$

Using Eq. A.13, the deadline of a task is correlated to its arrival time and estimated execution latency both on the Gateway and locally. The actual deadline is scaled according to a coefficient C_a, which in the context of this evaluation was set to 1.4.

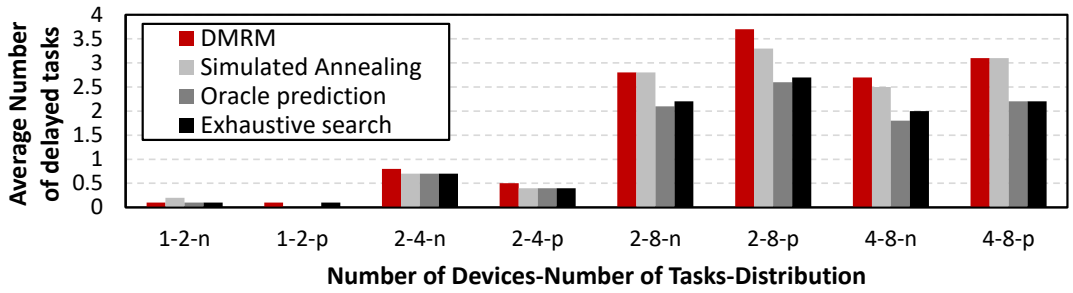


Figure A.2.: #Tasks with missed deadlines (Small number of tasks)

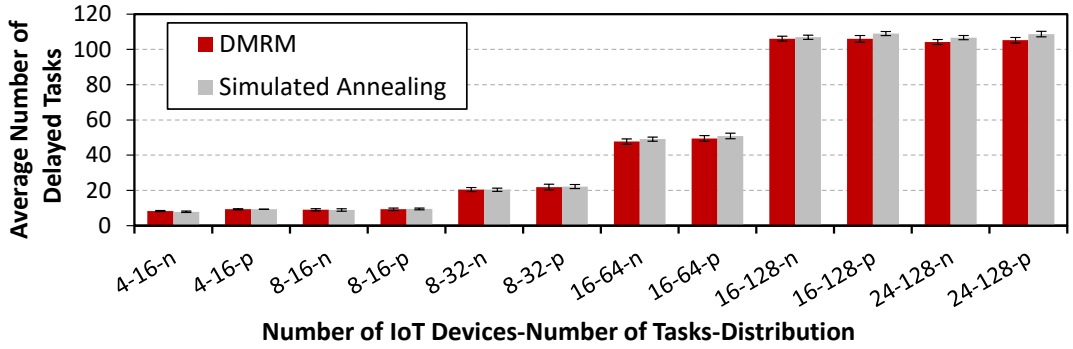


Figure A.3.: #Tasks with missed deadlines (High number of tasks)

A.5.2. Comparative Study

We evaluate DMRM against an offloading decision mechanism based on Simulated annealing (SA), a well-known probabilistic metaheuristic, which has been already used for task scheduling [358, 359] and application mapping [360]. SA is configured to solve at design time the optimization problem presented in Section B.3. We also calculate the optimal solution of the problem using an exhaustive, brute-force approach, to quantify the quality of the solutions provided by the heuristics. However, this exhaustive solution is executed only when new tasks arrive. Consequently, we augment the comparison by adding an Oracle prediction mechanism, which knows a priori all the tasks that will arrive at the system. Therefore, the Oracle scheduler can lead to less deadline misses by postponing the execution of a task in order to save resources for a task arriving in later rounds and has stricter deadline.

In the first comparative experiment, we evaluate DMRM against all other decision making alternatives. However, due to the exhaustive approaches, the experiments are conducted only for inputs with limited number of IoT tasks and devices, to avoid exploding the available scheduling combinations. The results of the comparison are summarized in Fig. A.2, where the values of the X-axis include triplets of the number of IoT devices of the examined scenario, the total number of tasks and the probability distribution function of tasks' arrival times, denoted as 'n' for Normal and 'p' for Poisson. The Y axis represents the number of delayed tasks. As shown, both DMRM and SA algorithms achieve similar results, which are very close to the misses of the exhaustive experiments.

The first evaluation, shows that DMRM provides results close to the optimal ones, but further more demanding experiments with scaled number of IoT devices and tasks are

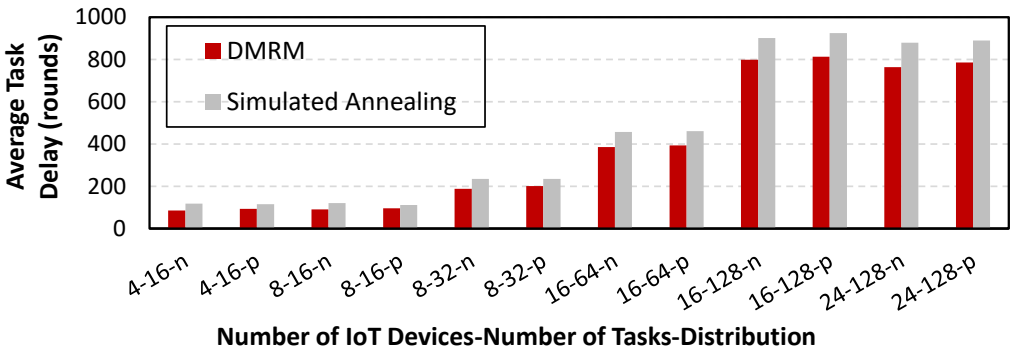
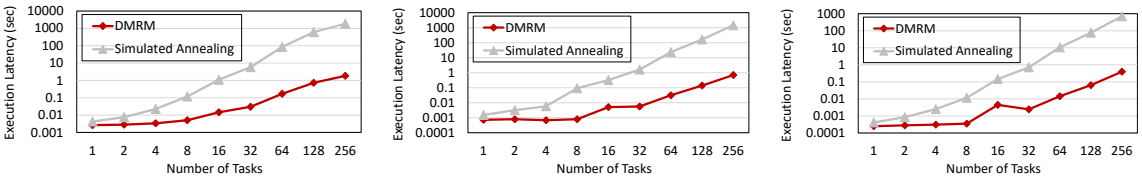


Figure A.4.: Total rounds of exceeded deadlines (Demanding input)



(a) Execution latency on Intel Galileo

(b) Execution latency on Raspberry Pi 3

(c) Execution latency on Nvidia Tegra

Figure A.5.: Comparative evaluation of DMRM and SA based resource management on Gateway alternatives (log. scale)

necessary. Fig. A.3 presents such an evaluation, maintaining the same format for X and Y axes. We observe that as the input workload of the system increases (both in number of devices and tasks), DMRM outperforms the SA approach, achieving an average reduction of 3.22% in the number of delayed tasks.

Since both DMRM and SA are approximate solutions, we further evaluate them in terms of the severity of the deadline misses. Specifically, the experiment presented in Fig. A.4 shows the total number of rounds that the delayed tasks exceeded their deadline, using the same input as the experiments presented in Fig. A.3. The results show that DMRM also decreases this value up to 12.35% compared to the SA approach.

Last, a successful dynamic resource management scheme should also be able to make decisions in an online manner. Therefore, we perform a comparison of the execution latency of the two heuristic approaches on the embedded devices presented in Section A.5.1. Fig. A.5 illustrates the measured execution latency in logarithmic scale, showing that for low number of IoT tasks, both solutions are characterized by similar performance. However, as the number of tasks rises, the execution latency of the SA approach skyrockets, while DMRM is insubstantially affected. This gap in the execution latency requirements is highlighted in the case of 256 tasks, where DMRM concludes 2000× faster. This behaviour is attributed to the inherent differences of the centralized nature of the SA approach as opposed to the scalable approach of the proposed market based solution, where the computational burden is distributed to the all involved devices of the system. In total, the execution overhead of DMRM is less than 1 second in the average case, while it only surpasses this in the least computationally powerful device (Intel Galileo), where at the worst case it reaches an overhead of 4 seconds.

Appendix B.

SGRM: Stackelberg Game-Based Resource Management on Edge Computing Systems

The incessant technological advancements of recent Internet of Things (IoT) networks have led to a rapidly increasing number of connected devices and workloads. Resource management is a key technique for such systems to operate efficiently. In this chapter, we present SGRM, a game theory-based framework for dynamic resource management of IoT networks under CPU, memory, bandwidth and latency constraints. SGRM combines a novel execution time prediction mechanism along with Stackelberg games and Vickrey auctions in order to tackle the multi-objective problem of task offloading in a competitive Edge Computing system. We design, implement and evaluate our novel game theory-based framework over a real IoT system for a diverse set of interference scenarios and varying devices, showing that i) the proposed prediction mechanism can provide accurate predictions, achieving 2.3% absolute percentage error on average, ii) SGRM achieves near-optimal results and outperforms alternative solutions by up to 66.6% and iii) SGRM provides scalable, real-time and lightweight performance characteristics.

This appendix text strongly relies on the following publications:

- *Karteris Antonis, Katsaragakis Manolis, Masouros Dimosthenis, & Soudris Dimitrios (2022, March). SGRM: stackelberg game-based resource management for edge computing systems. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1203-1208). IEEE. [361]*

B.1. Introduction

Over the last years, the connection and networking of alternative heterogeneous devices that need to interact and cooperate with each other in order to offer more efficient and advanced control and monitoring services to everyday users composes a multidisciplinary and self-configuring network, namely Internet of Things (IoT). Traditionally, a significant subset of computation, storage and processing has been offloaded to the cloud infrastructure for the information inference [344].

However, the explosive growth of IoT devices and their increasing computing power have resulted in unprecedented volumes of data, computational, storage and energy resources [362]. Moreover, machine learning-enabled applications deployed at the edge of IoT networks and the rise of time-sensitive and Quality of Service(QoS)-critical requirements [363] impose numerous challenges for efficiently operating IoT networks.

Geographically centralized Cloud servers and datacenters cannot guarantee real-time and latency-sensitive computation services due to large round-trip delays, network congestion and service quality degradation [345]. In order to overcome the aforementioned limitations, Edge Computing provides an alternative and promising platform paradigm to process tasks by pushing data inference to the Edge of the IoT network and alleviating the dependency on Cloud infrastructure.

In this work we target the well-established Edge Computing architecture, consisting of interconnected IoT nodes and Gateways. Both nodes and Gateways possess limited computational resources (CPU), limited memory capabilities and bounded network bandwidths. Substantial data workloads produced on the IoT nodes require processing, either locally or offloaded to the Gateways under throughput constraints. Thus, an efficient resource management to dynamically allocate the shared Gateway resources and define the operating behavior of IoT nodes is essential. This composes an NP-complete problem, which corresponds to the Multidimensional Multiple-Choice Knapsack Problem (MMKP), thus computationally intractable [347].

The main goal of this research is to take advantage of the inherently competitive and distributed nature of edge computing architectures, in order to tackle the multi-objective problem of task offloading, under resource and time constraints. **The contributions of this work are:**

- **We present an Edge computing system architecture,** where IoT nodes can either execute their tasks locally or offload them to Edge Gateways. Our model captures the limited resources and constraints of the system and formulates the

problem optimization objective as a deadline miss minimization problem.

- **We formulate and evaluate a novel Execution Time Prediction Mechanism** for heterogeneous applications and devices by identifying the special characteristics of the application, the available resources of the corresponding device and alternative interference levels.

- **We design and implement SGRM**, a distributed, scalable game-theoretic resource management framework that models the resource management problem as a Stackelberg game, utilizing Vickrey auctions.

- **We conduct an extensive experimental evaluation of the SGRM framework** in order to investigate the validity of our approach. We compare SGRM to a set of baseline algorithms and state-of-the-art resource management approaches over real hardware and network setups, showing that our proposed framework outperforms the compared approaches by up to 66.6%, while highlighting its distributed, scalable and lightweight nature.

The remainder of this appendix is organized as follows: Section C.2 presents related work. In section B.3 we present the system model and the formulation of the target optimization problem. The execution time prediction mechanism and the proposed Stackelberg-based framework are detailed in Section B.4. Section D.4 provides the experimental and comparative evaluation of the proposed resource management scheme, while Section D.5 concludes this work.

B.2. Related Work

Several works have been conducted in order to address the resource management challenges in Edge Computing systems. Authors of [347] address the problem of QoS management for IoT devices under bandwidth, battery and processing constraints, by proposing a centralized, pseudo-polynomial solution based on a dynamic programming approach, which enables reusing pre-computed solutions. In a similar setup, authors of [348], solve the task assignment problem using stochastic optimization, taking also into account the mobility of devices in the IoT network. Even though these approaches provide near-optimal solutions, they are not lightweight and lack in terms of scalability.

In order to overcome the inherent limitations of centralized approaches, market and game

theory based approaches have been employed. Authors of [343] present a distributed market-based approach for resource management in Edge Computing systems, however they target only single-Gateway systems. In [364], authors design a game-theoretic task allocation algorithm combined with reinforcement learning, however this approach avoids the critical parameter of real-time requirements.

Regarding Stackelberg games (SG), these have also been widely utilized to tackle the resource management problem in Edge computing systems. Authors of [365] tackle a resource management problem utilizing a reverse Stackelberg model, aiming to achieve maximum revenue of the target system. Authors in [366] combine an SG approach with a many-to-many matching game to design a framework for resource allocation in three-tier Edge networks. Both solutions cannot be applied in real-time systems as they exhibit increased latency overhead.

Although research has illuminated the potential impact of efficient resource management in Edge Computing systems, no study to date, according to our knowledge, has incorporated Stackelberg games and auctions in order to produce an efficient, decentralized solution for allocating latency-sensitive tasks and managing the limited resources of Edge Computing systems.

B.3. System and Problem Formulation

Our target system architecture consists of X IoT nodes (also called Edge devices) and Y Edge Gateways. The former are resource-constrained as far as CPU, memory and network capabilities are concerned and can process locally or offload tasks for execution to the latter. Regarding the communication of the devices, IoT nodes can be connected with any wired or wireless communication protocol supported by the Gateways.

Each IoT node is assigned a unique identifying value $x \in \{1, ..., X\}$, and described by a tuple $D_x = \{C_x, M_x, N_x, T_x\}$. Variables C_x, M_x and N_x denote the available processor, memory and network resources of node x respectively, while T_x indicates the set containing all the tasks τ_z^x generated on IoT node x. Similarly, every Gateway is defined by an identifying value $y \in \{1, ..., Y\}$, and described by a tuple $G_y = \{C_y, M_y, N_y, T_y\}$, where C_y, M_y and N_y denote the available CPU, memory and network resources of Gateway y respectively. T_y indicates the set containing all the tasks τ_z^y offloaded on Gateway y.

Table B.1.: Key System Model Parameters

Denotation	Description
C_x,M_x,N_x	Available CPU, Memory and Bandwidth of device x
C_y, M_y, N_y	Available CPU, Memory and Bandwidth of Gateway y
T_x, T_y	Set of tasks on device x and Gateway y
τ_z^x	Task z of device x
CL_z, ML_z, NL_z	CPU, Memory and Bandwidth required to execute task z locally
CG_z^y, MG_z^y, NG_z^y	CPU, Memory and Bandwidth required to execute task z on Gateway y
ct_z, et_z, Ω_z	Completion time, Estimated time & Deadline of task z of device x
y_z	Id of Gateway y that received task z, 0 otherwise
o_j^i	1 if task j of device i is offloaded, 0 otherwise
l_j^i	1 if task j of device i is executed locally, 0 otherwise

Every task of device x is uniquely defined by an identifying value z, and described by a tuple :

$$\tau_z^x = \{CL_z, ML_z, NL_z, CG_z^y, MG_z^y, NG_z^y, \Omega_z, ct_z, et_z, y_z\} \tag{B.1}$$

where CL_z, ML_z and NL_z signify the amount of CPU, memory and network bandwidth required for the task z to be executed locally. Similarly, CG_z^y, MG_z^y and NG_z^y signify the amount of CPU, memory and network bandwidth required for the task z to be executed on Gateway y. Ω_z signifies the deadline of the task z, while ct_z denotes the termination time of task z. Variable et_z denotes the estimated execution time of task z, as derived by the *Execution Time Prediction Mechanism* described in Sec. B.4.2. Moreover, we define o_x^z, a binary variable indicating whether the task z was offloaded or not, and variable l_x^z, which is its complement. Finally, variable y_z denotes the corresponding Gateway where the task was offloaded to. For tasks that are not offloaded, y_z is set to zero. All key parameters are summarized in table I.

$$\text{minimize} \quad N_{missed} = \Sigma_{\forall x}(\Sigma_{\forall z} miss(\Omega_z^x)) \tag{B.2}$$

$$\text{where} \quad miss(\Omega_z^x) = \begin{cases} 0 & if\ ct_z^x \leq \Omega_z^x \\ 1 & otherwise \end{cases} \tag{B.3}$$

$$\textit{Gateway y:} \quad \Sigma_{\forall x}(\Sigma_{\forall z} o_z^x \cdot \mathcal{CG}_z^x) \leq \mathcal{C}_y \qquad \textit{CPU}$$

$$\Sigma_{\forall x}(\Sigma_{\forall z} o_z^x \cdot \mathcal{MG}_z^x) \leq M_y \qquad \textit{Memory} \tag{B.4}$$

$$\Sigma_{\forall x}(\Sigma_{\forall z} o_z^x \cdot \mathcal{NG}_z^x) \leq N_y \qquad \textit{Network} \tag{B.5}$$

$$\textit{IoT Node x:} \quad \Sigma_{\forall z} l_z^x \cdot \mathcal{CL}_z^x \leq \mathcal{C}_x \qquad \textit{CPU} \tag{B.6}$$

$$\Sigma_{\forall z} l_z^x \cdot \mathcal{ML}_z^x \leq M_x \qquad \textit{Memory} \tag{B.7}$$

$$\Sigma_{\forall z} l_z^x \cdot \mathcal{NL}_z^x \leq N_x \qquad \textit{Network} \tag{B.8}$$

$$\textit{Unique execution} \quad o_j^k + l_j^k \leq 1, \forall j \in \mathcal{T}_k \tag{B.9}$$

The optimization objective of the target problem as expressed in Eq. B.2 is to minimize the number of tasks of IoT nodes that exceed their deadline. The task arrival trend is unknown, since IoT nodes can dynamically spawn new tasks. The solution is constrained by the available CPU, memory and network resources both on IoT Nodes and Gateways (Eq. B.4-B.8). Moreover, each individual task has the requirement of *Unique Execution* as shown in Eq. B.9, i.e., a task is prohibited to be executed both on an IoT node and a Gateway. Our proposed solution aims at solving the optimization problem by dynamically designating where each task of the IoT nodes will be executed.

B.4. SGRM: A Stackelberg Game-Based Resource Management Framework

B.4.1. Background and Main Proposed Concepts

The key idea of the proposed approach is to take advantage of the inherently competitive nature of IoT networks and implement a non-cooperative strategic game among IoT Nodes and Gateways. In order to model our approach, we utilize the following fundamental game-theory concepts:

- **Stackelberg games** [367], which are non-cooperative games where the participants are segregated into two groups and the game is played sequentially.

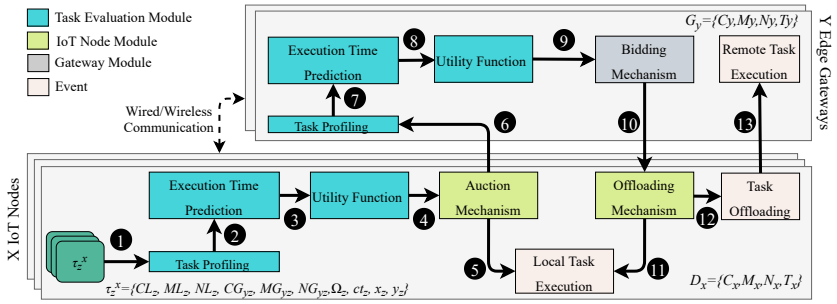


Figure B.1.: Overview of SGRM Operation Flow among IoT Nodes and Edge Gateways

- **Vickrey auctions** [368] or sealed-bid, second-price auctions, where the highest bidder wins the item in question. The second-price property forces the agents participating in the auction to bid truthfully on the value of the auctioned item.

In our approach, we model the task allocation decision making process of a node after a Stackelberg game played between the corresponding node and the Edge Gateways. The IoT node acts as a *leader* in the Stackelberg game, evaluating the task and deciding whether it should be offloaded to one of the available Gateways. Afterwards, the Gateways participating in the Edge computing system act as *followers* in the game and evaluate the incoming task, informing the corresponding IoT node. Finally, the IoT node conducts a Vickrey auction utilizing the task evaluations as bids, and offloads the task to the highest bidder.

Towards this direction, we present SGRM, a framework composed of three discrete modules divided among IoT nodes and Edge Gateways: (a) a *Task Evaluation Module*, which is executed by every individual device (IoT nodes and Gateways) on the network and consists of the *Task Profiling Mechanism*, the *Execution Time Prediction Mechanism* and the *Utility Function*, (b) a *IoT Node Module*, which is executed on IoT Nodes and consists of the *Auction* and *Offloading mechanisms* and (c) a *Gateway Module*, which is executed on the Gateways, containing the *Bidding Mechanism*. The efficient combination of these mechanisms via data exchange between nodes and Gateways completes SGRM, leading to its outcome, i.e. the decision of where each individual task of the IoT nodes will be executed, either on the Gateway or locally. Figure B.1 illustrates the operational overview of the SGRM framework steps.

B.4.2. Task Evaluation Module

Both IoT nodes and Gateways are equipped with a *Task Evaluation Module*, consisting of the *Task Profiling Mechanism*, the *Execution Time Prediction Mechanism* and the *Utility Function*. This module is responsible for a device to appraise an incoming task, estimate its execution time and quantify the computational and temporal cost of carrying it out.

Prior to a device's joining our target Edge Computing system, a pre-processing step of profiling the different types of tasks present in the system needs to be carried out on the device, and the effects of multiple tasks being executed in tandem need to be analyzed and quantified. To that end, the *Task Profiling Mechanism* (❶, ❻) executes a set of dedicated combinations of tasks to define alternative interference levels on the target device and records their execution times. The goal is to define a unique function for each individual device that will be utilized for predicting the execution time.

The process for evaluating a given task z on a device x is as follows: (1) *Chunk Gauging*: Task z is executed on the device, and its execution time is logged and quantized by a factor q, i.e. split into entities, namely $chunks_{z,x}$. Variable q is defined throughout experimentation. (2) *Execution Time Estimation*: A bundle of combinations of the different tasks present in the system is carried out in tandem, and the execution time $et_{z,x}$ for task z on device x is estimated as shown in Eq. B.10 each time.

$$et_{z,x} = chunks_{z,x} \times n_x \tag{B.10}$$

where n_x denotes the number of tasks being executed on device x. (3) *Error compensation*: In order to reduce the error produced by the *Execution Time Prediction* (❷,❼) steps, we generate a logarithmic trendline of the estimation error percentage as a function of the current number of tasks being executed on the device x (Eq. B.11).

$$compf_{z,x}(r_x) = \alpha_{z,x} \times \ln n_x + \beta_{z,x} \tag{B.11}$$

Constants $\alpha_{z,x}$ and $\beta_{z,x}$ are defined through extensive experiments. As a result of the aforementioned process, a tuple $(chunks_{z,x}, \alpha_{z,x}, \beta_{z,x})$ is generated for every task z and device x combination. The aforementioned tuple is passed on as input to the *Execution Time Prediction Mechanism*, which predicts for each incoming task z the execution time on device x as a function of the number of tasks currently running on the device (Eq. B.12). The process remains the same for Gateway y.

$$et_{z,x}(n_x) = chunks_{z,x} \times n_x - compf_{z,x}(n_x) \tag{B.12}$$

Fundamental to the decision making of the framework is the *Utility Function* (❸,❽), which attempts to capture and quantify the cost of undertaking a particular task in terms of time and computational resources into a single value that acts as a bid in the sealed-bid auction orchestrated by the corresponding IoT node.

In order to successfully capture this value, every spawned task is assigned an initial reward r_z, set to 1 by default. Subsequently, two penalty functions are estimated and applied to this reward value, decreasing it in accordance with the workload being already executed on the device and the estimated execution time of the task. The first penalty function penalizes the final utility according to the estimated execution time of the task z, similarly to [369]. It becomes negative when the task is estimated to miss its deadline (Eq. B.13).

$$f_{z,x|z,x,y} = \frac{\Omega_z}{\Omega_z - et_{z,x|z,y}(n_{x|y}) - wt_{z,x|z,y} - tt_{z,x,y}} \tag{B.13}$$

where $wt_{z,x|z,y}$ denotes the waiting time for task z to begin execution on device x or on Gateway y. Variable $tt_{z,x,y}$ denotes the time required to transfer task z from IoT node x to Gateway y. In case that a task is not offloaded, $tt_{z,x,y}$ is set to zero. The second penalty function (Eq. B.14) penalizes the utility further, according to the number of currently executing tasks.

$$p_{x|y} = \begin{cases} c^{n_{x|y}} & \text{if } f_{z,x|z,x,y} < 0 \\ \frac{1}{c^{n_{x|y}}} & \text{if } f_{z,x|z,x,y} > 0 \end{cases} \tag{B.14}$$

where $c > 1$ constitutes a weight factor selected through extensive experimentation. Finally, the utility function is extracted on the IoT Node x or on Gateway y, as shown in Eq. B.15.

$$u_{z,x|z,x,y} = \frac{r_z}{f_{z,x|z,x,y} \times p_{x|y}} \tag{B.15}$$

B.4.3. IoT Node Module

The portion of SGRM executed on an IoT Node is composed of two fundamental parts, i.e. the *Auction Mechanism* (❹) and the *Offloading Mechanism* (❿). The former defines

Algorithm 6: SGRM functionality on IoT Node Module

```
   Data: IoT Node, Gateways Tuple, Set of Tasks
 1 IoT Node-Algorithm(IoT Node, GatewaysTuple, Tasks):
 2     for task in Tasks do
 3         u = utility(task) /* estimate utility */
 4         if u > threshold then
 5             execute(task) /* local execution */
 6         else
 7             initiateAuction(task,GatewaysTuple) /* start auction */
 8             waitForBids(GatewaysTuple) /* wait for bids */
 9             bids = receiveBids(GatewaysTuple) /* receive bids */
10             winner = auctionWinner(bids) /* determine winner */
11             offload(task, winner) /* offload task to winner */
```

whether a task will be executed locally (❺) or whether it will participate as a candidate for potential offloading (❻). Each independent task is assigned a utility value, as described in B.4.2. We define *threshold* as the utility value over which the task should not be selected for offloading and is defined throughout extensive experimentation. The *Auction Mechanism* (❹) advertises all the corresponding tasks participating in the auction to the Edge Gateways. Each Gateway returns a bid for the corresponding task, as described in Section B.4.4. Once all bids are collected by the IoT Node, the auction is performed. If the corresponding IoT node has made the highest bid then the task is executed locally (⓫), otherwise the Gateway with the highest bid is the winner of the auction. Next, the *Offloading Mechanism* (❿) is enabled and the input data is sent to the Gateway that won the auction (⓬, ⓭). Algorithm 6 summarizes the functionality of the *IoT Node Module*.

B.4.4. Gateway Module

The core functionality of the SGRM on the Gateways is dominated by the *Bidding Mechanism* (❾). For every individual task offloading request received in the corresponding Gateway, a utility value is calculated, as described in Section B.4.2. The utility value is given as bid to the auction conducted on the corresponding IoT Node. The Gateway that wins the auction conducted on the IoT Node receives the corresponding task and input for execution. The functionality of the *Gateway Module* is summarized in algorithm 7.

Algorithm 7: SGRM functionality on IoT Node Module

```
   Data: Gateway, IoT Nodes Tuple, Set of Tasks
1  Gateway-Algorithm(IoT Node, GatewaysTuple, Tasks):
2      for task in Tasks do
3          u = utility(task) /* estimate utility */
4          if u > threshold then
5              sendBid(task,IoTNodesTuple,u) /* send bid */
6              winner = receiveAuctionWinner(task) /* learn winner of auction */
7              if winner is Gateway then
8                  receive(task) /* receive task data */
9                  execute(task) /* start execution */
```

B.4.5. Implementation

The SGRM framework consists of the core functionality logic, as described previously, and is responsible for monitoring, coordinating and deploying the target system and the applications. The SGRM framework has been implemented using the Python programming language, and the wireless communication between the IoT Nodes and Gateways is achieved through utilizing the MQTT communication protocol. Moreover, aiming to avoid the latency overhead added by an application at startup, each application's process is launched in the background prior to data arrival, and remains idle until the reception of the corresponding input, i.e. the startup latency overhead is tackled once. Moreover, the components of SGRM are containerized, utilizing the Docker platform. The SGRM framework is publicly available with an open source licence[1].

B.5. Experimental Evaluation

B.5.1. Experimental Setup

To evaluate SGRM, we implement an IoT network that consists of heterogeneous embedded devices. We utilize three Raspberry Pi 4 Model B's (Quad-core Cortex-A72 @ 1.5GHz, ARMv7 32-bit Architecture), one Jetson TX1 (Quad-core Cortex-A57 @ 1.9 GHz, ARMv8 64-bit Architecture), one Jetson Nano (Quad-core Cortex-A57 @ 1.43 GHz, ARMv8 64-bit Architecture) and two powerful virtual machines (Octa-core Intel Xeon @ 2 GHz, x86 64-bit Architecture). All devices are equipped with 4GB of RAM.

[1] https://github.com/UphillD/edgebench

We utilize four real-life machine learning applications with alternative resource requirements. The focus is on real-world benchmarks rather than synthetic, thus making the evaluation of our approach representative of performances of applications running on real production. More specifically, we utilize *Deepspeech* [370], *Facenet* [371], *Lanenet* [372] and *Retain* [373]. For each application the corresponding input data is offloaded from the IoT node to the target Gateway, respectively.

In order to properly evaluate our approach, we conduct a set of varying experiments in terms of arrival time and density of the incoming tasks. Specifically, we consider two distinct cases: In the first one, created using the Normal distribution, tasks are arriving almost simultaneously, thus creating a huge demand for resources. The second scenario involves cases,where simultaneous high peaks of demand are avoided and the Poisson distribution is selected for this goal. The tasks are randomly distributed among the devices and their deadlines are defined through extensive experimentation.

B.5.2. Experimental Results

Comparative Analysis: We evaluate SGRM against various resource management algorithms. First, in order to quantify the performance of SGRM against the optimal solution, we augment the comparison by adding an Oracle prediction mechanism, which possesses all the incoming task information a priori and provides the optimal decision making through an exhaustive search method. Moreover, we compare against two naive approaches: (1) Offload None, which executes all tasks without offloading any computation to the Gateways and (2) Offload All, on which all input workload is offloaded. Last but not least, we develop from scratch and compare against a state-of-the-art market-based resource management algorithm presented in [343], namely DMRM. Since DMRM is designed for systems with one Gateway, we examine its efficiency only on single-Gateway experiments.

Small-scale Workload Analysis: In the first comparative experiment, we evaluate SGRM against all other alternatives over a single Gateway. However, due to the exhaustive behaviour of *Oracle* prediction, the experiment is conducted only for inputs with limited number of tasks and devices. Fig. B.2 illustrates the results of the comparison. The x-axis denotes the triplets of the number of incoming tasks, the number of IoT nodes and the number of Edge Gateways of the examined scenario. The y-axis represents the number of tasks that exceed their deadline. As shown, SGRM clearly reduces the number of the delayed tasks compared to the naive approaches of Offload All by 42.5% and Offload None by 50.9% on average, and it also outperforms DMRM by 15.3% on average. Moreover, our approach leads to near-optimal results, as the proposed approach approxi-

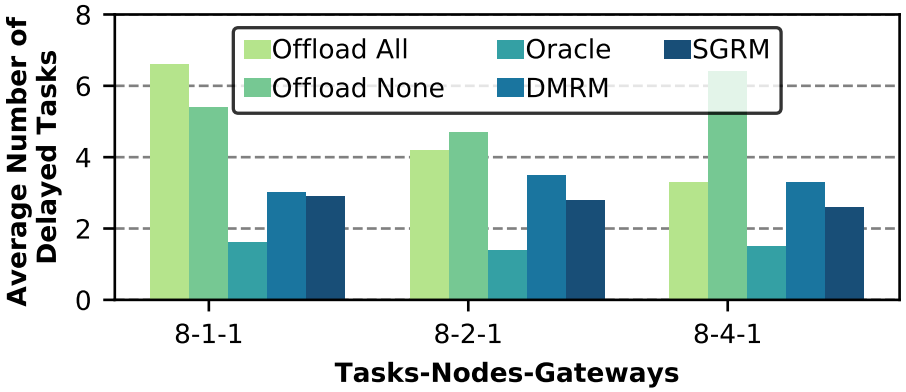
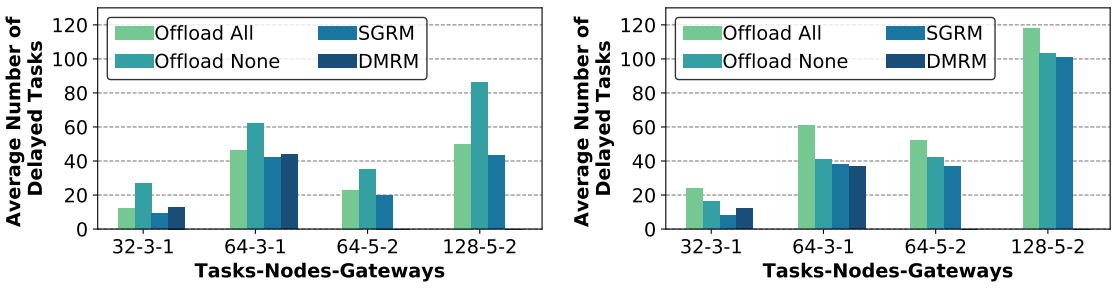


Figure B.2.: #Tasks with missed deadlines (small-scale workload)

mates the Oracle prediction results, closely. This initial evaluation illustrates that SGRM provides near-optimal results.

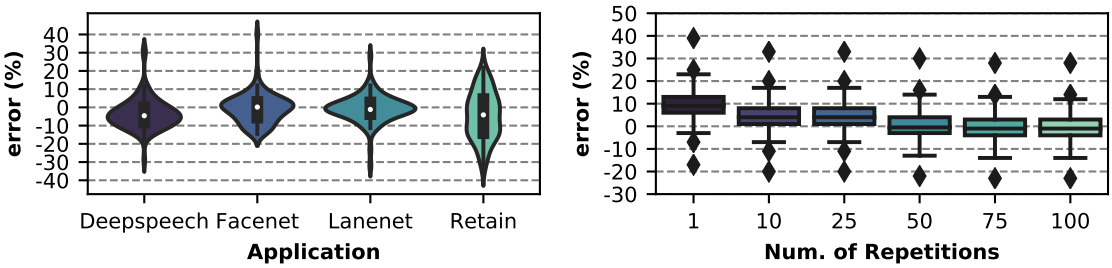
Larger-scale Workload Analysis: However, further investigation is required in order to evaluate the behaviour of our framework under increased workload, IoT Node and Gateway conditions. Fig. B.3 depicts this evaluation, maintaining the same format for the X and Y axes as the previous experiment. In the first scenario, tasks are generated according to the Normal distribution, while in the second scenario we utilize the Poisson distribution. Our proposed framework achieves up to 25% fewer deadline misses compared to Offload All and up to 66.6% compared to Offload None for Normal distribution workloads, as shown in Fig. B.3a. Similarly, for Poisson distribution, SGRM achieves 36.9% fewer deadline misses compared to Offload All and 17.7% compared to Offload None on average, as illustrated in Fig. B.3b. In both scenarios, for single-Gateway experiments, SGRM achieves up to 30.7% fewer missed deadlines compared to the DMRM approach.



(a) Normal Distribution

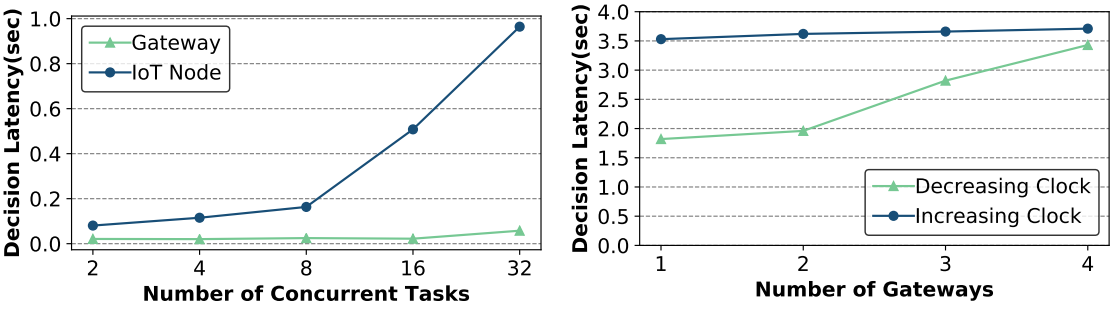
(b) Poisson Distribution

Figure B.3.: #Tasks with missed deadlines (larger-scale workload)



(a) Error Percentage **(b)** Rate of Convergence

Figure B.4.: Execution time prediction mechanism analysis



(a) Tasks Scalability **(b)** Gateways Scalability

Figure B.5.: SGRM's tasks and Gateways scalability

Execution Time Prediction Analysis: The functionality of our approach is directly linked to the efficiency of our proposed average time prediction mechanism. We evaluate the prediction mechanism in terms of percentage error and the rate of convergence of our learning approach, i.e. how fast our system learns, as depicted in Fig. B.4. The violin plots shown in Fig. B.4a present the average execution time prediction error percentage for each application. We observe an absolute error rate of 2.3% on average across all applications. Moreover, in Fig. B.4b, we evaluate the number of executions required over alternative interference levels for a single application in order to train our model and reach near-zero average error. We observe that our proposed mechanism converges for 50 repetitions.

Task Scalability Analysis: Moreover, we evaluate the scalability of SGRM, in terms of the number of concurrent tasks and Gateways, as illustrated in Fig. B.5. Figure B.5a highlights the decision latency of the SGRM framework on a single IoT Node and a single Gateway for up to 32 tasks generated concurrently. X-axis denotes the number of concurrent tasks generated on the IoT Node, while Y-axis denotes the decision latency in seconds. An efficient dynamic resource management scheme should be able to make decisions

in an online manner. We observe that Gateway's decision time remains almost constant as tasks increase. Additionally, the IoT Node processing latency rises exponentially, which is caused by the increased number of auctions being carried out in the system sequentially. However, the decision latency still satisfies real-time requirements. Moreover, for more than 32 consecutive tasks, network bandwidth starts to be congested, i.e. the scalability depends on the specifications of the underlying network.

Gateway Scalability Analysis: Lastly, we evaluate the scalability of our framework over an increasing number of Gateway devices, as shown in Fig. B.5b. In this experiment we dynamically add Gateways to the system. In the first scenario we add Gateways in decreasing CPU clock frequencies, while in the second scenario in increasing CPU clock frequencies. The decision latency of the latter gradually converges to the latency of the former, as we increase the number of Gateways, i.e., the decision time of the Gateways only depends to the computational capabilities of the most CPU-constrained device and no communication overhead is added by SGRM.

B.6. Conclusion

In this appendix we presented SGRM, a novel resource management and task allocation framework for Edge Computing systems. SGRM combines a novel execution time prediction mechanism along with Stackelberg games and Vickrey auctions to tackle the problem of efficient, real-time task offloading. We evaluate our framework over a real IoT network, showing that i) the proposed prediction mechanism can provide accurate predictions, achieving 2.3% absolute percentage error on average, ii) SGRM achieves near-optimal results and outperforms alternative solutions by up to 66.6% and iii) SGRM provides scalable, real-time and lightweight performance characteristics.

Appendix C.

RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems

Deep Neural Networks (DNNs) are becoming extremely popular for many modern applications deployed at the edge of the computing continuum. Despite their effectiveness, DNNs are typically resource intensive, making it prohibitive to be deployed on resource- and/or energy-constrained devices found in such environments. To overcome this limitation, partitioning and offloading part of the DNN execution from edge devices to more powerful servers has been introduced as a prominent solution. While previous works have proposed resource management schemes to tackle this problem, they usually neglect the high dynamicity found in such environments, both regarding the diversity of the deployed DNN models, as well as the heterogeneity of the underlying hardware infrastructure. In this chapter, we present RoaD-RuNNer, a framework for DNN partitioning and offloading for edge computing systems. RoaD-RuNNer relies on its prior knowledge and leverages collaborative filtering techniques to quickly estimate performance and energy requirements of individual layers over heterogeneous devices. By aggregating this information, it specifies a set of Pareto optimal DNN partitioning schemes that trade-off between performance and energy consumption. We evaluate our approach using a set of well-known DNN architectures and show that our framework i) outperforms existing state-of-the-art approaches by achieving 9.58× speedup on average and up to 88.73% less energy consumption, ii) achieves high prediction accuracy by limiting the prediction error down to 3.19% and 0.18% for latency and energy, respectively and iii) provides lightweight and dynamic performance characteristics.

This appendix text strongly relies on the following publications:

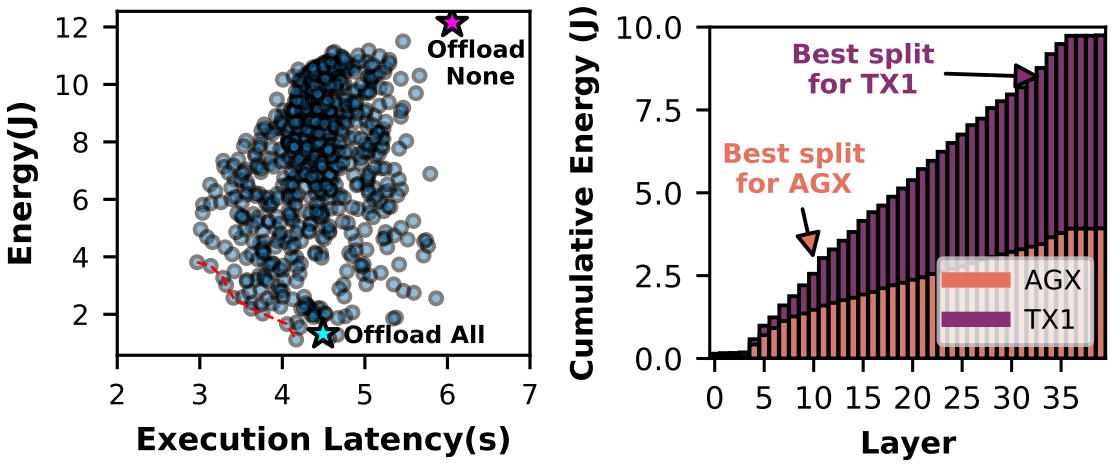
- *Kakolyris Andreas Kosmas, Katsaragakis Manolis, Masouros Dimosthenis, & Soudris Dimitrios (2023, April). RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems. In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1-6). IEEE. [374]*

C.1. Introduction

Over the last years, the growth of applications that utilize sophisticated Machine Learning (ML) techniques to make value out of complex data is rapidly increasing and is expected to grow further in the future. In this direction, Deep Neural Networks (DNNs) are being widely adopted by many application domains, including, but not limited to, autonomous driving [375], biomedical and healthcare applications [376] and Intelligent Personal Assistants (IPAs) [377], mainly due to their capability in offering high prediction accuracy.

Such types of applications are typically deployed at the edge of the computing continuum, closer to where data are generated, in order to enhance security and minimizing the data transfer latency to the cloud [378]. Even though DNNs provide extremely accurate results, their computational and memory requirements can skyrocket, thus introducing several barriers on how to be efficiently deployed on resource-contrained edge computing devices [379]. Considering also that DNNs are gradually becoming deeper to support more discriminative results and more levels of hierarchy are integrated in the learned representation [380], the aforementioned problem becomes even more intense.

Traditionally, to overcome this limitation, the DNN inference of edge devices was offloaded to high-end servers hosted on cloud premises (e.g., Amazon Elastic Inference, Azure Machine Learning). However, this approach leads to huge amount of data travelling back and forth in the continuum resulting in high latency and energy consumption. Moreover, all the computational effort is being concentrated in the cloud, rendering it incapable to cope with the ever-increasing demand for resources [381]. Aiming to deliver more efficient and energy proportional computing systems, hardware vendors (e.g., Nvidia, Xilinx) are introducing more powerful edge devices, which often integrate conventional CPUs, hardware accelerators and specialized units for DNN computations (e.g., Nvidia's Tensor cores) as a system-on-chip (SoC). While such hardware devices deliver increased compute density, they are also extremely power hungry, thus, becoming restrictive for applications that operate under certain energy or battery constraints.



(a) ResNet101 energy and latency for different partitioning schemes

(b) ResNet101 energy's optimal partitioning for different devices

Figure C.1.: DNN optimal partitioning schemes w.r.t. *a)* different optimization goals and *b)* device heterogeneity.

Aiming to identify the "golden ratio" between performance and energy efficiency for edge computing devices, while also limiting the network latency bottleneck, *DNN partitioning and offloading* has been identified as a promising solution [382, 383]. The aim of DNN partitioning is to provide a fine-grained layer-level split of the DNN, where a portion of the computation is performed locally on the edge device and the rest on the cloud. Still, identifying an optimal partitioning scheme is not trivial and depends on a triptych of user-defined requirements and hardware-oriented criteria, i.e., *i)* the architecture of the deployed DNN model (e.g., computational/energy requirements and data offloading overhead per layer), *ii)* the inherent nature of the application itself (e.g., latency critical applications would sacrifice energy for performance) and *iii)* the underlying hardware characteristics. As a motivational example, Fig. C.1a shows the performance and energy of all the possible partitioning schemes for a ResNet101 architecture. We see that model partitioning can provide significantly faster execution and less energy consumption compared to on-board and offloaded execution, while also each optimization objective is attained through different partitioning schemes. Moreover, Fig. C.1b further reveals, that the optimal partitioning scheme also relies on the underlying hardware, with different edge devices providing different splits.

To tackle these challenges, several prior research approaches have examined the problem of DNN partitioning and offloading [382–384]. These solutions assume that the architecture of the deployed model, as well as the underlying hardware are known a priori and apply the offloading scheme based on extensive profiling of the DNN. However, edge computing

environments are extremely dynamic, both with respect to the devices arriving in the network and the alternative applications deployed. Moreover, novel DNN architectures are emerging [378], leading to more and more complex techniques, such as skip-layer connections, early exits and others.

In order to adapt to the challenges that new DNN architectures impose and to the dynamic nature of recent edge computing systems, we design an efficient resource management framework to dynamically allocate, partition and offload DNN layers among edge and cloud resources, aiming to provide performance and/or energy consumption optimizations and trade-offs. **The novel contributions of this work are:**

- **We present `RoaD-RuNNer`**, a novel resource management framework consisting of a set of *Offline* and *Online Decision Making Mechanisms*.
- **We implement a collaborative filtering based prediction mechanism** in order to provide per layer predictions regarding execution time and energy consumption.
- **We design and integrate a dynamic partition mechanism**, for efficiently splitting and offloading DNN layers.
- **We conduct an extensive experimental evaluation of our proposed framework**. We compare our framework with a set of baseline algorithms and state-of-the-art DNN offloading approaches over real hardware and networking, showing that it outperforms existing state-of-the-art approaches by achieving 9.58× speedup on average and up to 88.73% less energy consumption average.

The remainder of this appendix is organized as follows: Section C.2 presents related work. In Section D.3 we analyze our proposed prediction and offloading mechanism. Section C.4 provides our experimental and comparative evaluation of the proposed resource management scheme, while Section D.5 concludes this work.

C.2. Related Work

Several works have been conducted in order to address the resource management challenges of DNNs deployed on edge computing systems. Deep learning model partitioning and offloading is becoming a rising trend in recent edge computing systems. Research conducted by [385] considers various present and future challenges for efficiently deploying

deep learning models on the edge. In a similar perspective, authors of [386] provide an overview of the overarching architectures, frameworks, and emerging key technologies towards training/inference for deep learning models at the network edge.

Aiming to provide DNN resource management schemes, authors of [384] propose a partitioning-based DNN offloading technique for edge computing, by dividing the DNN model into partitions and uploading them to the edge server. In a similar perspective, authors of [387] present a distributed progressive inference engine that addresses the challenge of partitioning CNN inference across device-server setups. In [388] a framework that dynamically partitions a DNN model that adapts to the changes of computational resources and network condition is proposed. Authors of [382] design a workload partitioning algorithm to decide efficient DNN partitioning policy in real-time, while in [389] authors utilize several IoT devices by creating a local collaborative network for a subset of deep learning models, mainly focusing on the impact of convolutional layers. Last but not least, authors of [383] present the efficiency of DNN processing on the cloud based on pre-loaded layers.

Although research has illuminated the resource management and offloading of neural network based applications on edge computing systems, no study to date, according to our knowledge, has incorporated collaborative filtering in order to produce an efficient, decentralized resource management solution for Neural Network offloading at single layer granularity, while targeting heterogeneous CPU/GPU architectures. *Neurosurgeon* [383], is the most similar approach to ours, however there exist several fundamental differences. We target a fully dynamic DNN offloading framework, in contrast to *Neurosurgeon*, where DNN weights are loaded a priori to the cloud server. Moreover, *Neurosurgeon* is limited to operate over models without skip-layer connections, in contrast to `RoaD-RuNNer`, which is designed to operate over all types of DNN/CNN.

C.3. `RoaD-RuNNer`: A collaborative DNN partitioning & offloading framework

`RoaD-RuNNer` tackles the problem of DNN partitioning and offloading over heterogeneous CPU/GPU edge computing systems, where a portion of the computational burden can be offloaded from the edge to the cloud for remote execution. Each individual edge node accommodates a set of DNN inference tasks which need to be executed. `RoaD-RuNNer`'s goal is to identify optimal DNN splittings, down to the granularity of a single layer and offload computationally heavy slices, aiming to optimize the total inference execution latency and/or the energy consumption. Figure D.3 depicts an overview

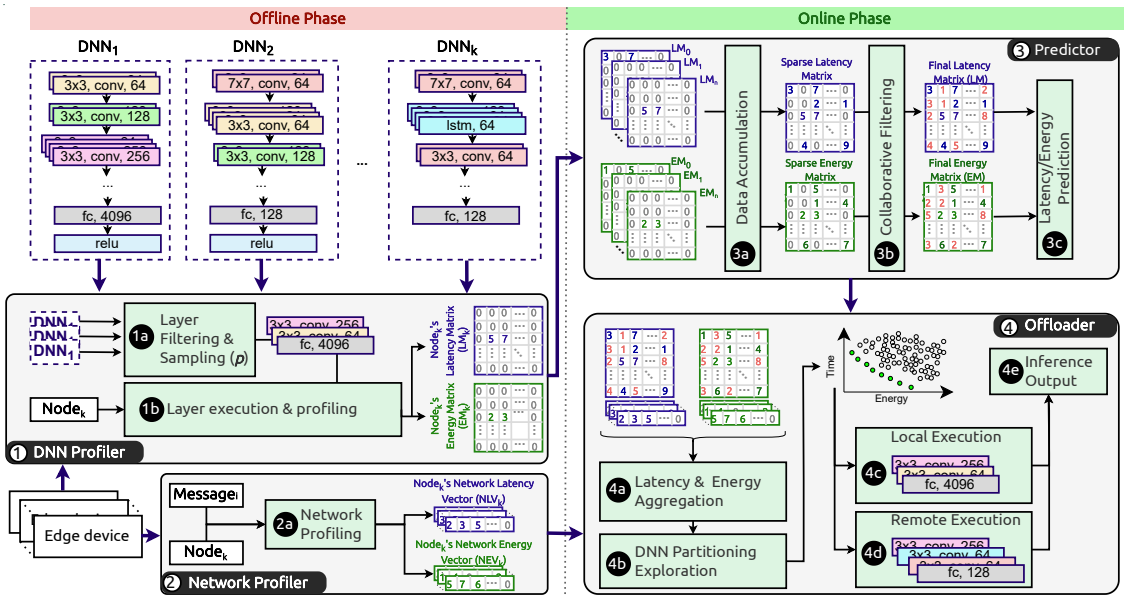


Figure C.2.: Overview of Online and Offline RoaD RuNNer Architecture.

of `RoaD-RuNNer`. The major components of our proposed framework are split in two major phases: (i) *Offline Phase* and (ii) *Online Phase*. The former is composed of *DNN-Profiler* and *Network Profiler*, while the latter consists of the *Predictor* and *Offloader* mechanisms. The core functionality of these components is described in the rest of this section.

C.3.1. Offline Phase

Offline Phase consists of two distinct components: (i) *DNN Profiler* (❶) and (ii) *Network Profiler* (❷), which are responsible for generating the required data and knowledge to be fed as input to the run-time mechanisms later. As input to the framework, we provide alternative neural network configurations to each single node of the composed edge computing network.

DNN Profiler: Aiming to take advantage of the inherent distributed nature of the edge computing paradigm and the heterogeneity in terms of edge devices and deep learning models, we implement a collaborative filtering mechanism [390,391], in order to train our system to efficiently predict per layer execution time and energy consumption for each device, respectively. As a first step of the offline decision making, we integrate a *Layer*

Filtering and Sampling (1a) component, aiming to filter and sample the layers of the alternative neural networks that are fed as input to our framework. More specifically, a small percentage p of the input layers in each DNN is randomly selected to be accurately profiled on the edge device itself. This ensures that profiling time does not skyrocket when the total number of layers in the DNN increases. The percentage p is defined through experimentation.

Next, the layers that have been selected through the filtering and sampling process are propagated as input to the *Layer Execution & Profiling* (1b) step. Each of the sampled layers is executed locally on each node, respectively. The execution is profiled, aiming to gather data regarding the execution latency and the energy consumption per layer. Thus, for every single node k, we extract the Node's Latency Matrix (LM_k) and Node's Energy Matrix (EM_k), respectively. The LM_k and EM_k are propagated as input for further processing to the *Predictor* mechanism, as described in Section C.3.2. This process is triggered once for each node.

Network Profiler: The efficiency of an edge computing system is directly related to its ability to effectively operate over the existing network infrastructure. Moreover, offloading decision making mechanisms should consider the overhead imposed by the underlying network. Thus, for each individual node k we integrate an extended *Network Profiling* mechanism (2a). As input we provide a set of alternative workloads to be sent/received from each edge device to the cloud server. We profile the transmission latency and power of data sent and received from and to the edge device. The profiled messages range from KB to GB orders of magnitude.

After the profiling is finalized, each node k is characterized by two vectors: (i) The Network Latency Vector (NLV_k) and the Network Energy Vector (NEV_k), which represent the latency and energy requirements for alternative message sizes, respectively. The produced vectors are utilized as dataset to produce polymonial trendlines, in order to predict the latency and energy requirements of a given input message size. Thus, for each individual device, fourth-order polynomial curves are generated in order to be utilized for run-time prediction during the *Online Phase*.

C.3.2. Online Phase

An efficient dynamic resource management scheme should be able to make decisions in a run-time manner. Thus, we integrate into our framework two key components: (i) *Predictor* (3) and (ii) *Offloader* (4). The former is responsible for dynamically predicting latency and energy per layer, while the latter is responsible for network

and collaborative filtering data aggregation, dynamic DNN partitioning andRoa offloading.

Predictor: The output matrices produced by the *DNN Profiler* (❶) of each single node k during the offline phase are accumulated to the cloud server (3a). Two new sparse matrices are produced, i.e. *Sparse Latency Matrix* and *Sparse Energy Matrix*, respectively. Each row of the matrix represents an edge node, while each column denotes a single layer type. The *Collaborative Filtering* mechanism (3b) is triggered, based on matrix decomposition, i.e. based on the factorization of each matrix into a product of matrices. The unknown values are filled, and the *Final Latency Matrix(LM)* and the *Final Energy Matrix(EM)* are produced. After the predictions are finalized, each device retrieves its corresponding results from the cloud server, which are utilized for the final *Latency/Energy Prediction* (3c). Opposed to prior works, which mostly rely on extended profiling for each new deep learning model [384, 387], our framework is adaptable to dynamic scenarios. New nodes that dynamically join the network are integrated in the existing latency and energy matrices and the collaborative filtering matrices are updated. New incoming DNNs can benefit from the existing layers in the collaborative filtering matrices.

Offloader: The last component of the *Online Phase* is responsible for the aggregation of network and collaborative filtering data and the dynamic DNN partitioning and offloading. For each single node k, the network profiling vectors (NLV_k, NEV_k), and the collaborative filtering matrices (LM, EM) for latency and energy are aggregated (4a), in order to provide the final prediction per layer, including computation and transmission overhead. Next, we proceed to the *DNN Partitioning Exploration* (4b), in order to provide a set of Pareto optimal solutions, aiming to optimize performance and/or energy consumption. The DNN partitioning exploration algorithm is illustrated in Algorithm 8. A set of Pareto optimal solutions in generated. Each Pareto point corresponds to an alternative partition, on which a subset of DNN layers is executed locally (4c) and the rest are offloaded for remote execution on the cloud server (4d). Each partition is identified by two indexes i, j where layers 0 to i are executed locally, layers $i+1$ to j are executed on the cloud and the layers from $j+1$ and up to the end of the network are executed locally. Fully local and fully remote executions remain valid alternatives, in case they belong to the Pareto optimal solutions. Moreover, in contrast to existing approaches [383] our framework is designed to handle shortcut dependencies, i.e. skip-layer connections. Such kind of dependencies are resolved by encapsulating the layers that create the dependency in an larger atomic block that has a single input and output and are offloaded as an individual component.

Table C.1.: Technical characteristics of heterogeneous Edge nodes and Cloud Server

Device	CPU	L2	L3	DRAM	GPU
Jetson Nano	4×Cortex-A57@1.4GHz ARMv8 64-bit	2MB	-	4GB	128×Maxwell@0.9GHz
Tegra X1	4×Cortex-A57@1.4GHz ARMv8 64-bit	2.5MB	-	4GB	256×Maxwell@1.0GHz
Xavier NX	6×Carmel@1.4GHz ARMv8.2 64-bit	6MB	4MB	8GB	384×Volta@1.1GHz
Xavier AGX	8×Carmel@2.2GHz ARMv8.2 64-bit	8MB	4MB	32GB	512×Volta@1.4GHz
Cloud Server	2×20 core Intel Xeon Gold 5218R@2.1GHz	1MB	28MB	128GB	5120xVolta-V100@1.2GHz

Algorithm 8: DNN Partitioning Algorithm

```
1   for layer in (0,...,N-1) do
2       /* predict local for layers 0 to layer */
3       l1=predictLocal(0, layer)
4       for j in (layer+1,...,N-1) do
5           /* predict network,cloud for layers i + 1 to j */
6           n=predictNetwork(layer+1, j)
7           c=predictCloud(layer+1, j)
8           /* predict network,cloud for layers j + 1 to N − 1 */
9           l2=predictLocal(j+1, N-1)
10          totalPredictions=accumulatePredictions(l1,n,c,l2)
11  /* find Pareto Optimal */
12  paretoPoints = DesignSpaceExploration(totalPredictions)
13  return paretoPoints
```

C.4. Experimental Evaluation

C.4.1. Experimental Setup

Hardware Infrastructure: We deploy an in-house system setup consisting of heterogeneous CPU/GPU devices, the specifications of which are shown in Table C.1. As edge devices, we utilize a set of NVIDIA GPU-SoCs, to exploit power/performance trade-offs. As offloading machine, we employ a powerfull x86 server equipped with an NVIDIA V100 GPU, which forms a typical setup both in edge and cloud premises [392].

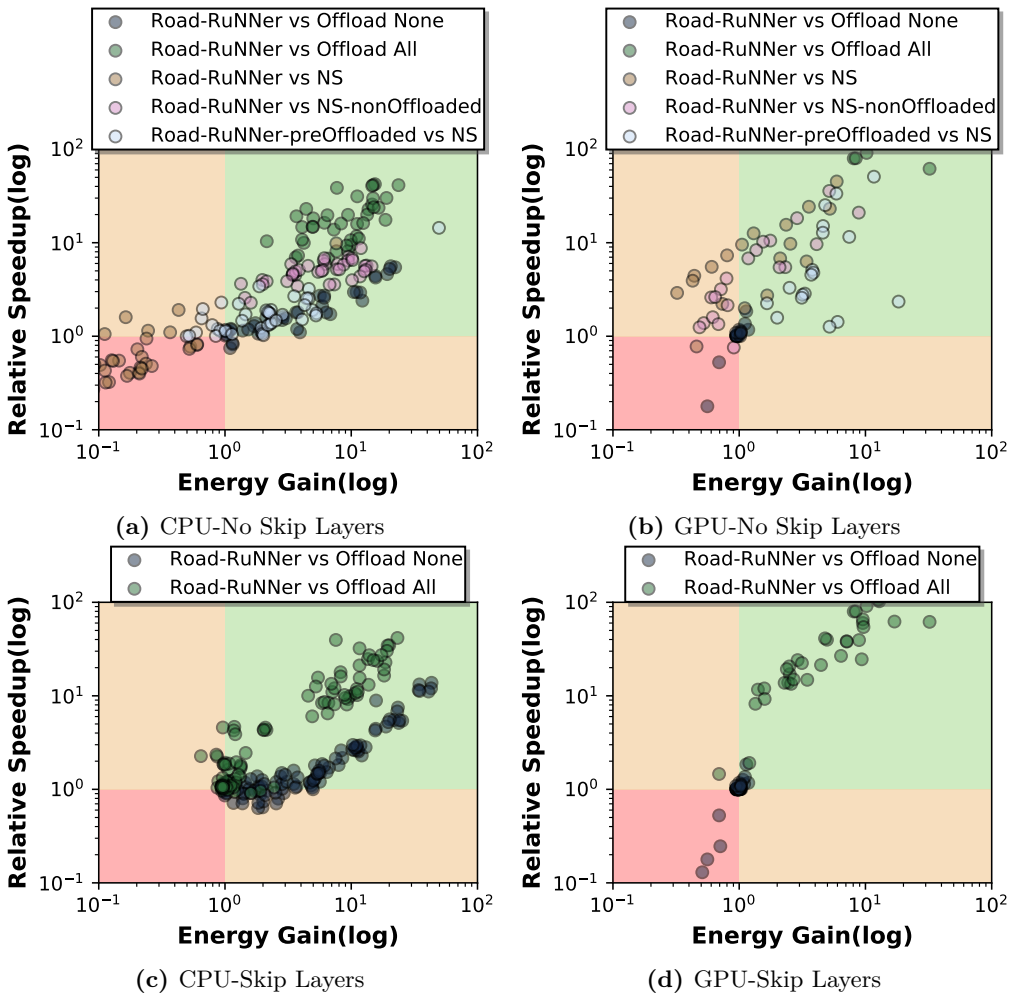
Technical Implementation: *RoaD-RuNNer* is implemented in Python programming language. All devices are interconnected through 80MB/s wireless network. We utilize the ZeroMQ messaging protocol for control and the FTP protocol for transmission of the actual data and layers. In order to overcome the architectural differences of heterogeneous CPUs all applications are integrated inside docker containers, while the corresponding GPU implementations are developed in CUDA. The collaborative filtering component

used for estimating the performance and energy impact of different layers is accelerated through the use of C++ for increased performance.

Examined DNN models: We examine famous DNN architectures and known variations, i.e. AlexNet (alex), MobileNetV2 (mv2), Resnet18 (res18), Resnet34 (res34), Resnet50 (res50), Resnet101 (res101), Resnet152 (res152), VGG11, VGG13 and VGG16, which are widely used for performing object detection and image classification tasks at the edge [393, 394], over alternative input image sizes (224×224, 512×512 and 768×768). The input models are derived from PyTorch [395] and are integrated to `RoaD-RuNNer`.

Reference Baselines: We evaluate the impact of our approach based on three key metrics: (i) performance (ii) energy consumption and (iii) prediction accuracy of our collaborative filtering approach. We compare against various partition and offloading mechanisms. First, as a baseline we utilize two naive approaches: (i) *Offload None*, which executes all tasks locally, without offloading anything on the cloud and (ii) *Offload All*, in which all tasks are offloaded to the cloud for remote execution. Moreover, we implement from scratch and compare against a state-of-the-art resource management algorithm for DNN offloading, namely *Neurosurgeon*(NS) [383]. Since *Neurosurgeon* is designed with the assumption to operate with a priori offloaded layers on the cloud infrastructure, we also implement a version of *Neurosurgeon* with online layer offloading, namely *NS-nonOffloaded* and a version of `RoaD-RuNNer`-*preOffloaded*, where the layers are offloaded a priori to the cloud.

C.4.2. Evaluation



(a) CPU-No Skip Layers

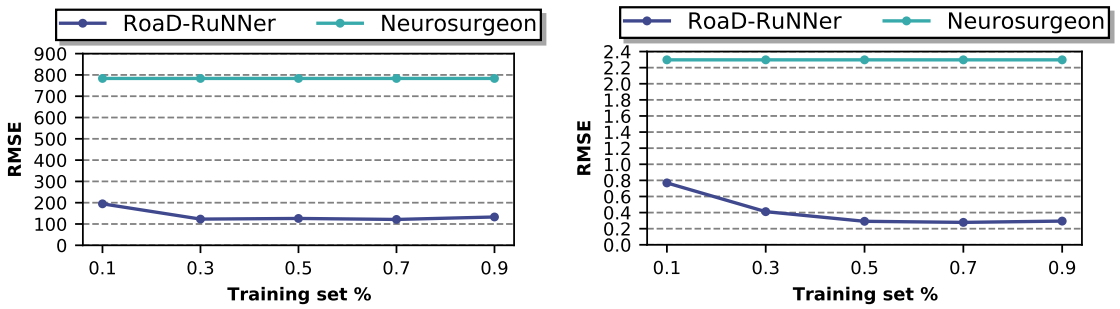
(b) GPU-No Skip Layers

(c) CPU-Skip Layers

(d) GPU-Skip Layers

Figure C.3.: Performance and Energy Comparison of RoaD-RuNNer framework against other approaches for CPU and GPU nodes for alternative DNN workloads.

Performance and Energy Evaluation: In the first comparative experiment, we evaluate `RoaD-RuNNer` in terms of performance and energy consumption against the approaches presented in Section C.4.1, as illustrated in Fig. C.3. Given the fact that *Neurosurgeon* is designed to operate only over non-Residual Neural Networks, we utilize as benchmarks the VGG11,13,16 and AlexNet models. X axis indicates the corresponding energy gain, while Y axis denotes the relative speedup of our framework compared to other approaches



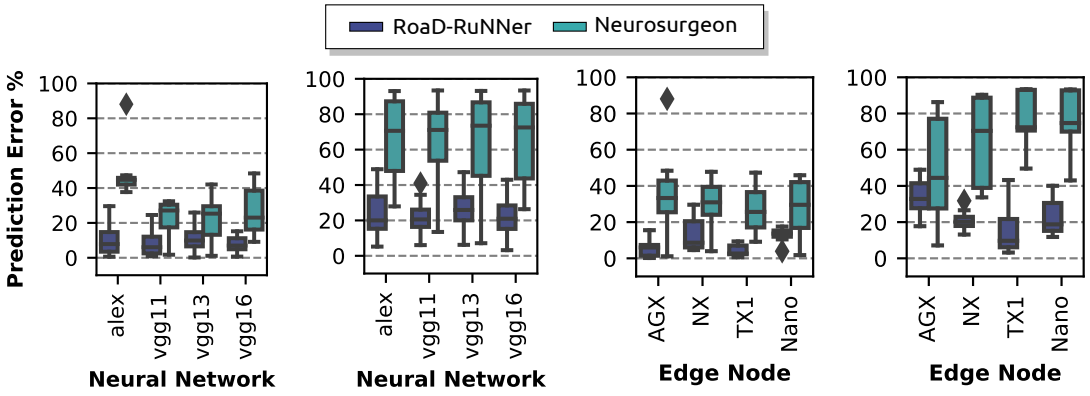
(a) Execution Time RMSE

(b) Energy Consumption RMSE

Figure C.4.: Root Mean Square Error(RMSE) of execution time and energy consumption over alternative Collaborative Filtering fill percentages.

for CPU (Fig.C.3a) and GPU (Fig.C.3b) execution, respectively. The output is divided in four distinct quadrants, on which `RoaD-RuNNer` does not achieve any speedup or energy gain (red), achieves either speedup only or energy gain only (orange) and achieves both speedup and energy optimization (green). We observe that *Road-RuNNer* clearly outperforms the *Offload All* and *Offload None* approaches, by leading up to 45.41× speedup, 95.84% energy reduction compared to the former and up to 6.61× performance optimization and 95.87% energy reduction compared to the latter, for CPU execution. Similar observations are derived for GPU execution. The initial version of *Neurosurgeon*, on which all layer weights are offloaded a priori, performs better in terms of performance and energy consumption, as the network overhead is a dominant factor. However, compared to the *NS-nonOffloaded*, where layer weights are offloaded dynamically, we observe that `RoaD-RuNNer` provides on average 4.97× optimized performance and 81.85% less energy consumption for CPU, and up to 35.74× optimized performance and 88.73% less energy consumption for GPU, respectively. `RoaD-RuNNer` is designed to provide up to two partition points, in contrast to *Neurosurgeon*, on which there exists only a single breakpoint. Therefore, the latter pays the penalty of either executing locally all the resource intensive layers or offloading their weights, thus paying the network penalty. Similarly, `RoaD-RuNNer`-*preoffloaded*, where all model layers are offloaded offline, our approach outperforms *Neurosurgeon* by up to 54.09× in terms of performance and up to 58.06× in terms of energy consumption.

In contrast to *Neurosurgeon*, `RoaD-RuNNer` is designed to operate efficiently over workloads consisting of DNNs with skip layer connections. Thus, we add to the existing benchmarks the Resnet18, Resnet34, Resnet50, Resnet101, Resnet152 and MobileNetV2 models and evaluate our approach. The evaluation is depicted in Fig. C.3c and Fig. C.3d for CPU and GPU executions, respectively. For CPU execution, our framework outperforms *Offload None*, by achieving up to 13.92× optimized performance and 46.07×



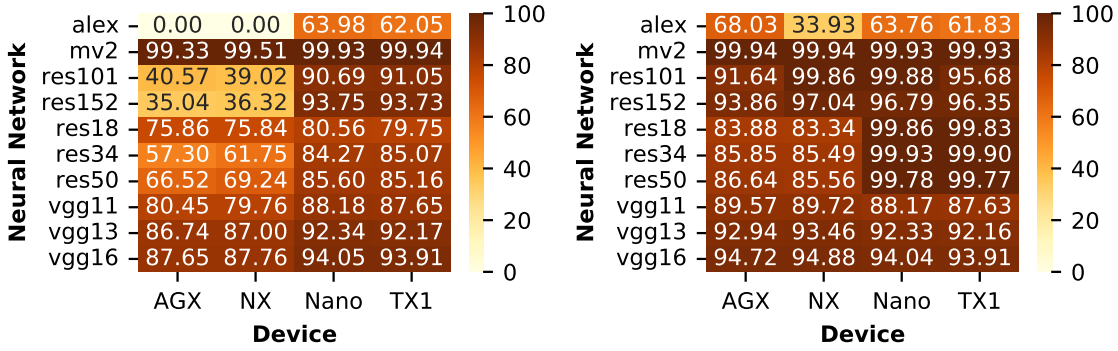
(a) Latency per DNN model **(b)** Energy per DNN model **(c)** Latency per Edge Device **(d)** Energy per Edge Device

Figure C.5.: Execution Latency and Energy Consumption Prediction Accuracy for alternative DNN workloads and Edge nodes.

less energy consumption and *Offload All* by achieving up to 45.41× optimized performance and 24.05× less energy consumption, respectively. Similarly, for GPU executions, `RoaD-RuNNer` achieves up to 1.85× speedup and up to 1.34× less energy consumption compared to *Offload None* and 112.98× speedup and 16.59× less energy consumption on average compared to *Offload All*.

Prediction Accuracy: The efficacy of our framework is directly related to the accuracy of the collaborative filtering mechanism. First, we evaluate the learning phase of the prediction mechanism in terms of Root Mean Square Error (RMSE) related to the size (percentage) of training set and compare with the *Neurosurgeon's* prediction approach. We observe that providing the 30% of our training set, the execution time and energy RMSE has converged. Thus, we set our training set size to 0.3. Compared to *Neurosurgeon*, our proposed mechanism achieves up to 6.45× and 8.48× less RMSE, for execution time and energy, respectively. *Neurosurgeon* displays linear behavior, as it utilizes 100% of the dataset during the training phase.

Next, as illustrated in Fig. C.5, we compare the execution time accuracy and energy prediction accuracy of `RoaD-RuNNer` (Fig. C.5a, C.5b) and *Neurosurgeon* per deep learning model and Edge node (Fig. C.5c, C.5d), respectively. Our framework achieves 68.01% less execution time prediction error and 63.8% less energy prediction error per DNN on average, while we achieve up to 69.6% less execution time prediction error and up to 34.9% less energy prediction error per edge device. In contrast to our prediction mechanism, *Neurosurgeon* is based on linear and logarithmic regression, making it impossible to provide high accuracy (or low error) and capture non-linear behaviors in the system,



(a) Latency Objective **(b)** Energy Objective

Figure C.6.: DNN percentage offloading over heterogeneous Edge nodes for latency and energy optimization objectives.

thus leading to high RMSE.

DNN Offload Analysis: Further discussion can be conducted on the decision making of `RoaD-RuNNer`, in order to achieve latency and energy optimization objectives. Thus, in Fig. C.6 the layer offloading percentage in terms of the number of layers offloaded for each model and edge node is depicted, in order to achieve each optimization objective, respectively. First, as depicted in Fig. C.6a, we see that the more powerful devices (AGX, NX) offload less computation for remote execution, compared to less powerful devices (Nano, TX1). More specifically, for AGX and NX the 58.7% of the target DNN is offloaded on average, opposed to Nano and TX1, where the 87.1% is offloaded. Similar observations are extracted for the energy optimization objective, as shown in Fig. C.6b. Furthermore, having execution latency as the optimization objective, the 74.45% of layers is offloaded on average, while for the energy optimization objective the corresponding percentage rises to 90.1%. This is due to the fact that the network imposes high latency overhead, thus making the data and layer transmission prohibitive in order to meet latency optimization objectives.

C.5. Conclusion

This work presents `RoaD-RuNNer`, a novel resource management framework for DNN partitioning and offloading over heterogeneous CPU/GPU edge computing systems. Our framework strongly leverages collaborative filtering techniques to estimate performance and energy requirements of individual DNN layers over heterogeneous devices. By aggregating this information, it specifies a set of Pareto optimal DNN partitioning schemes that

trade-off between performance and energy consumption. Our approach outperforms existing state-of-the-art approaches by achieving 9.58× speedup on average and up to 88.73% less energy consumption and high prediction accuracy by limiting the prediction error down to 3.19% and 0.18% for latency and energy, respectively.

Appendix D.

Decoupled Access-Execute enabled DVFS for tinyML deployments on STM32 microcontrollers

Over the last years the rapid growth Machine Learning (ML) inference applications deployed on the Edge is rapidly increasing. Recent Internet of Things (IoT) devices and microcontrollers (MCUs), become more and more mainstream in everyday activities. In this chapter we focus on the family of STM32 MCUs. We propose a novel methodology for CNN deployment on the STM32 family, focusing on power optimization through effective clocking exploration and configuration and decoupled access-execute convolution kernel execution. Our approach is enhanced with optimization of the power consumption through Dynamic Voltage and Frequency Scaling (DVFS) under various latency constraints, composing an NP-complete optimization problem. We compare our approach against the state-of-the-art TinyEngine inference engine, as well as TinyEngine coupled with power-saving modes of the STM32 MCUs, indicating that we can achieve up to 25.2% less energy consumption for varying QoS levels.

This appendix text strongly relies on the following publications:

- *Elisavet Lydia Alvanaki, Manolis Katsaragakis, Dimosthenis Masouros, Sotirios Xydis & Dimitrios Soudris. Decoupled Access-Execute enabled DVFS for tinyML deployments on STM32 microcontrollers. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE 2024) (pp. 1-6). IEEE. [267]*

D.1. Introduction

Over the last years, *Edge AI* [396] has appeared to the forefront as a novel computing paradigm. Edge AI refers to the deployment of complex Deep Neural Network (DNN) models directly on edge devices, such as sensors, Internet of Things (IoT) devices, and other embedded systems. Furthermore, the increasing demand for edge-centric digital signal processing applications, such as video analytics [397], mobile visual tasks [398] and others has established Convolutional Neural Networks (CNNs) as the go-to technology for efficiently handling these tasks at the edge. Notably, major technology providers have responded to this paradigm shift by offering enterprise-grade solutions designed to optimize and deploy CNN models at the edge. These solutions include software-based stacks (e.g., Amazon SageMaker Edge [399]) as well as custom, purpose-built hardware chips (e.g., Google's Edge TPU ASIC [400]).

A significant amount of research efforts are focusing on exploring the complex DNN/CNN design space of dataflow schedules [401, 402] to optimize energy. Despite their sophistication, these solutions often assume the presence of hardware accelerators or OS-supported and network-connected edge devices. However, as the computing continuum extends to the far edge of networks, arrays of resource-constrained devices, typified by microcontrollers (MCUs), are introduced into the ecosystem. Unlike their more capable counterparts, MCUs typically employ bare-metal application execution with custom or lightweight run-times and, at times, disconnected from the network grid. This paradigm shift to the far edge has highlighted the concept of "tinyML" [403]. TinyML involves the optimization of DNN/CNN models to operate seamlessly on these bare-metal MCUs, thus, extending the boundaries of where ML can be applied. In the face of limited processing power and memory, tinyML models are designed to offer efficient inference capabilities, making it possible to bring intelligence to devices previously excluded from the ML landscape.

Deploying CNNs on MCUs introduces two major challenges. First, the limited memory capacity of MCUs poses difficulties in both storing and executing CNN models. This becomes even more complex, since emerging CNN architectures tend to become much deeper, aiming to provide better accuracy and/or support more complex applications [374]. Second, given that MCUs are frequently embedded in battery-operated edge devices, preserving energy resources becomes crucial, since the execution of resource-intensive and computationally hungry DNNs can rapidly deplete the battery, particularly concerning devices with extended operational requirements.

Focusing on CNN inference, prior research [404] and widely-used frameworks (e.g., TFLite Micro [405], Microsoft's NNI [406], ARM's CMSIS-NN [407], etc.) offer optimization

techniques that can enable the deployment of tiny CNN models on resource-constrained MCUs. These techniques typically include static, model-specific, post-training optimizations, such as mixed precision arithmetic [408], model's weight pruning [409] and quantization [410, 411], as well as downsampling [412] and Neural Architecture Search (NAS) methodologies [403,413]. Other works examine schedule and source code optimizations either to enable efficient data and computation mapping on the underlying hardware [1,414] or to divide CNN layers into independently distributable tasks that can be offloaded and executed in parallel [379].

Although a vast amount of optimizations have been investigated for DNN deployment on MCUs, they mostly focus on models' structural features, e.g., weights, arithmetic, architecture etc., while little attention is given to system-level runtime optimization knobs. In the context of power consumption and/or energy efficiency optimization, Dynamic Voltage and Frequency Scaling (DVFS) [415] forms an efficient tuning knob to balance performance and energy consumption by right-adjusting the clock frequency of the MCU according to the computational demands of the deployed model. Still, the customization of DVFS policy to DNN specific features as well as the materialization of DVFS in practice is not straightforward, as it may impose switching overheads [416] not straightforwardly analyzable and increased leakage power due to longer execution times [417].

In this work, we introduce an end-to-end methodology that exploits DVFS for optimizing the energy consumption of CNN inference on low-end MCUs. The proposed methodology is built upon three pillar concepts, i.e. i) selection of energy optimal MCU clocking scheme among differing iso-latency configurations, ii) employment of Decoupled Access Execute (DAE) computing scheme for CNN convolution layers to unlock higher DVFS efficiencies and iii) extraction of DVFS optimal allocation decisions tailored to target CNN architectures and MCU. More specifically, we propose a Decoupled Access Execution scheme, which splits the execution of convolution layers' kernels into memory-bound and compute-bound regions, thus, allowing the exploitation of processor idling during memory accesses. We characterize this new design space for energy-efficient DNNs and we cast/formulate a knapsack optimization problem taking into consideration the MCU's clock parameters and DAE granularities to obtain the optimal DVFS strategies given the neural network architecture and the corresponding QoS constraint. To the best of our knowledge, *this is the first work that examines the application of DVFS for CNN inference on low-end MCUs.* We note that our methodology is orthogonal to optimization frameworks proposed in the past and can be seamlessly applied to pre-/post-training optimized models to further increase the energy efficiency at runtime (e.g., optimized models exported from TFLite Micro [405] or TinyEngine [1]). We showcase the validity and efficacy of our methodology by applying it on top of three CNN inference models derived from the TinyEngine and deploying the resulting model on a STM32 MCU, one

of the most popular and widely used MCUs in the embedded systems domain. Our experimental results show that we can achieve up to 25.2% power optimization compared to existing state-of-the-art approaches.

D.2. Clocking Scheme of STM32 Microcontrollers

The clocking system of STM32 microcontrollers is managed by the Reset and Clock Control (RCC) peripheral. The RCC provides a wide range of clocks and clock sources which cater to various system requirements, e.g., peripheral Bus and UART clocks [418]. In this work, we focus on specific clocks and settings, which determine the frequency of the system clock (`SYSCLK`) of the MCU, responsible for driving the CPU core, memory, and some peripheral modules. Figure D.1 illustrates the simplified circuit diagram of how the `SYSCLK` is determined. Specifically, the output frequency of the clock can be configured by the following clock sources and settings:

- **High-Speed Internal (`HSI`) Clock:** The `HSI` clock source is an internal oscillator within the MCU. The `HSI` clock operates at 16MHz by default. `SYSCLK` can be derived directly from `HSI` or through the `PLL` when `HSI` is selected as the `PLL` input source.

- **High-Speed External (`HSE`) Clock:** The `HSE` clock source is an external clock provided by an external crystal oscillator or clock generator. Depending on the MCU, the `HSE` clock can be configured to run at various frequencies, with our examined board supporting a range from 1 to 50MHz. Similar to the `HSI`, the `SYSCLK` can be derived directly from `HSE` or through the `PLL` when `HSE` is selected as the `PLL` input.

- **Phase-Locked Loop (`PLL`):** The `PLL` is a hardware module which allows to multiply the frequency of the selected input clock source (either `HSI` or `HSE`) by a programmable factor to generate a higher-frequency output. As shown in Fig. D.1, this programmable factor is determined by input and output dividers within the `PLL` circuit, which simplify the `PLL` design and achieve stability requirements within a wider range of input and output frequencies [418]. Specifically, the frequency of the system clock can be calculated by the following equation:

$$F_{\texttt{SYSCLK}} = F_{\{\texttt{HSE},\texttt{HSI}\}} * \frac{\texttt{PLLN}}{\texttt{PLLM} * \texttt{PLLP}} \tag{D.1}$$

  where, `PLLM` is a factor that determines how much the input frequency is multiplied before it reaches the Voltage-Controlled Oscillator (`VCO`) in the `PLL`; `PLLN` is the mul-

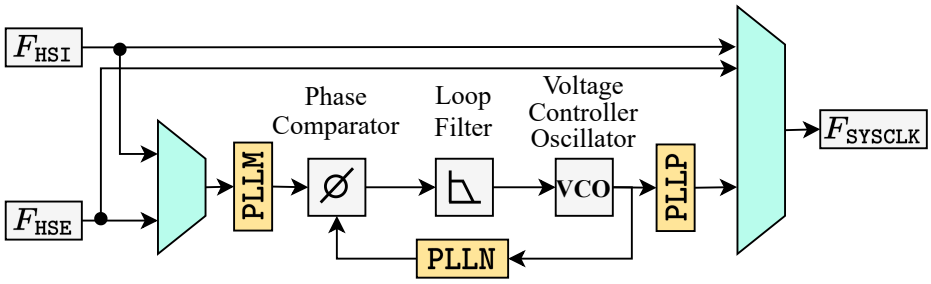


Figure D.1.: Simplified circuit diagram for clock configuration through HSE and PLL parameters.

tiplication factor for the VCO input frequency to determine the VCO output frequency, which is then provided as feedback to the Phase Comparator, aiming to provide input/output synchronization. The loop filter is utilized in order to ensure system stability and mitigate ripple effects during clock startup; Last, PLLP determines the division factor to obtain the final SYSCLK frequency.

D.2.1. Considerations of SYSCLK Frequency Scaling

Determining the frequency of the system's clock, $F_{\texttt{SYSCLK}}$, can be achieved in different ways (e.g., directly through the HSI/HSE clock sources or through the PLL circuit). Choosing between these alternatives involves important trade-offs. To examine the impact of each alternative on the operational efficiency and power consumption of the MCU, we develop and execute a specialized microbenchmark, designed to execute repetitive addition operations within a loop. We focus our exploration specifically on the HSE and PLL parameters, since the HSI clock source yields higher power consumption compared to the HSE and is also prone to drift and jitter, thus, providing less stability and precision [418]. We set the value of the PLLP equal to 2, which is the minimum possible value for the divider, since for the same $F_{\texttt{SYSCLK}}$, selecting a higher PLLP value leads to a higher required VCO frequency and, thus, higher power consumption (as evident from Fig. D.1 and Eq. D.1).

Power consumption of iso-frequency configurations: Figure D.2 illustrates the impact of different HSE, PLLM and PLLN configurations on the power consumption of the board for different SYSCLK frequencies. Two major observations are derived from Fig. D.2: (i) the same output frequency can be generated through different HSE, PLLM and PLLN combinations, however (ii) the selected configuration strongly affects the power consumption on the STM32 MCU. For instance, generating an SYSCLK output of 100MHz

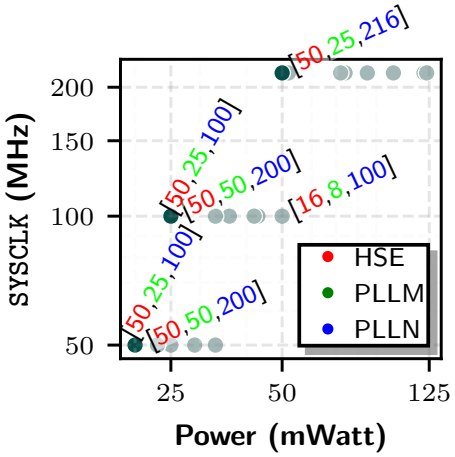


Figure D.2.: Clock Frequency and Power for different `HSE`, `PLLM` and `PLLN` configurations.

for the tuples $\{50, 25, 216\}$ and $\{16, 8, 100\}$ leads to 50% power gap. The combinations that minimize the power consumption are selected for the target `SYSCLK`. Different combinations may generate the same output frequency and power consumption, e.g. $\{50, 25, 100\}$ and $\{50, 50, 200\}$, thus further investigation is required for selecting the optimal combination. Similar observations are derived for other `SYSCLK` configurations.

Switching between different `SYSCLK` frequencies: Generating the `SYSCLK` frequency using the `PLL` module introduces a notable switching overhead ($\approx 200\mu sec$), since, when modifying the `PLL` parameters, the circuit has to be restarted, resulting in a substantial delay per switch. On the other hand, switching from the `PLL` frequency to the `HSE` clock occurs almost instantly, due to the direct wiring of the `HSE` with the `SYSCLK` (Fig. D.1). Thus, for high-to-low (i.e., $< 50MHz$)) frequency switches, opting for the `HSE` clock over re-calibrating the `PLL` parameters can be beneficial.

D.3. Decoupled Access Execute enabled DVFS on STM32 MCUs

This section introduces the proposed methodology for optimizing energy consumption of CNN model inference on a STM32 MCU under latency constraints (a.k.a. QoS).The rationale behind our approach is that end-users often have distinct inference latency/throughput constraints for their applications and/or even operating in the context of battery-

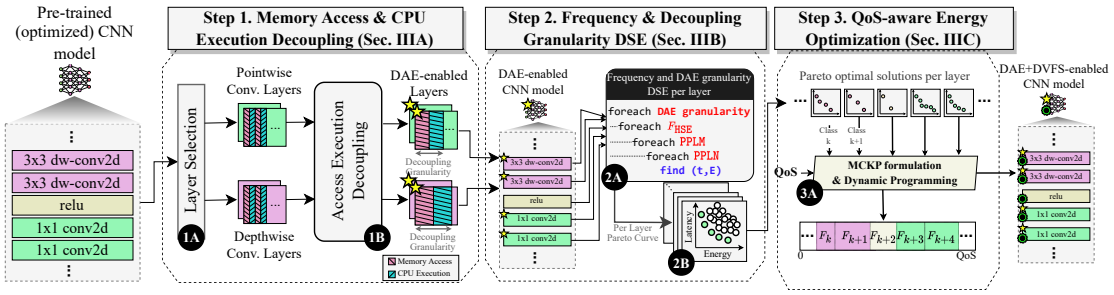


Figure D.3.: Overview of the proposed methodology.

operated far-edge MCUs. Figure D.3 shows an end-to-end overview of the proposed approach. The proposed methodology consists of three distinct phases (described in Sections D.3.1 - D.3.3), applicable to both unoptimized CNN models and optimized ones, e.g., models exported from TinyEngine [1].

D.3.1. Step 1: Memory Access & CPU Execution Decoupling

The first step of our proposed methodology focuses on source-code level restructuring, with the objective to **D**ecouple memory **A**ccesses from CPU **E**xecution (DAE), thus, creating memory-bound and compute-bound sub-segments within the layer's structure. DAE restructuring is a key-enabler for our strategy, as it provides more fine-grained control over when and how frequently memory accesses and computations occur (Sec. D.3.2). This enables the application of different frequencies for extended durations and with finer control, tailored to the specific requirements of each operation, such as memory access and computation, thus, resulting in the mitigation of clock switching overhead issues. We first identify the CNN model's most computationally-intensive and time-consuming layers (1A). We focus and apply DAE (1B) on two specific layer types, i.e., *i)* depthwise and *ii)* pointwise convolutions. These layer types make up over 80% of the total number of layers found in deep lightweight CNN models, e.g., Mobilenet [398], which employ the concept of depthwise separable convolution to reduce model's size and complexity.

Depthwise Convolutions: Depthwise convolution is a specialized CNN operation where each input channel is convolved with a separate learnable filter, capturing spatial features per channel. Typically, CNNs gradually learn increasingly complex features, each one represented by a different channel. For instance, in an image, the initial 3 input channels (RGB) increase as the network processes the image to extract and represent more complex

Listing D.1: Simplified source code modification for enabling decoupled access-execution (DAE) in depthwise convolutions.

```
for (channel = 0; channel < in_channels/g; channel+=g) {
    //LFO for memory bound operations (Sec.IIIB)
    ClockSwitchHSE(hse);
    //Memory Bound Segment: Load g channels from the feature maps
    q15_t buf1 ,..., bufg = getChannels(ch1,...,chg);
    //HFO for computation (Sec. IIIB)
    ClockSwitchPLL(pllm,plln,hse);
    // Compute Bound Segment: Perform depthwise convolution for each channel
    convolve_depthwise(kernel, buf1, **args);
    convolve_depthwise(kernel, buf2, **args);
     ...
    convolve_depthwise(kernel, bufg, **args);
}
```

features, such as textures, specific objects and others. State-of-the-art frameworks, such as CMSIS-NN [407] and TinyEngine [1] implement a per-channel computation approach for depthwise convolutions. In contrast, our DAE approach introduces a parametric unrolling factor called the "decoupling granularity", denoted by g. This factor determines the number of channels that are buffered in cache memory before the convolution operation is executed on each of them, thus, separating the code into memory-bound and compute-bound subsegments. Listing D.1 provides a simplified code snippet that illustrates the practical implementation of DAE optimization, showing how the decoupling granularity enables the efficient execution of depthwise convolutions. For example, when $g = 4$, four channels are fetched in the MCU's cache memory before proceeding to the actual computation. This division enables the application of different clock frequencies per segment, which we further discuss in Sec. D.3.2.

Pointwise Convolutions: Depthwise convolution is often followed by pointwise convolution to perform channel-wise mixing and dimensionality reduction, thus, reducing model size and computational complexity while maintaining performance. Pointwise convolutions involve `1x1` kernel sizes and are applied to each element within input channels. CMSIS-NN [407] and TinyEngine [1] implement pointwise convolutions in a per-column manner. Each column consists of one element per input channel. Our approach performs decoupling on the per channel memory accesses, thus splitting the code segment into memory and compute-bound regions. Just as in the depthwise convolution, we introduce the concept of the decoupling granularity, denoted as g, for modular buffering support w.r.t. the number of columns fetched from memory. For instance, for a 8x8x3 input image and a 1x1x3 kernel, g columns are loaded into the cache before performing computation for each one, in contrast with TinyEngine and CMSIS-NN, which load a single 1x1x3 image column at a time. In general, integrating multiple buffers leads to the generation of larger

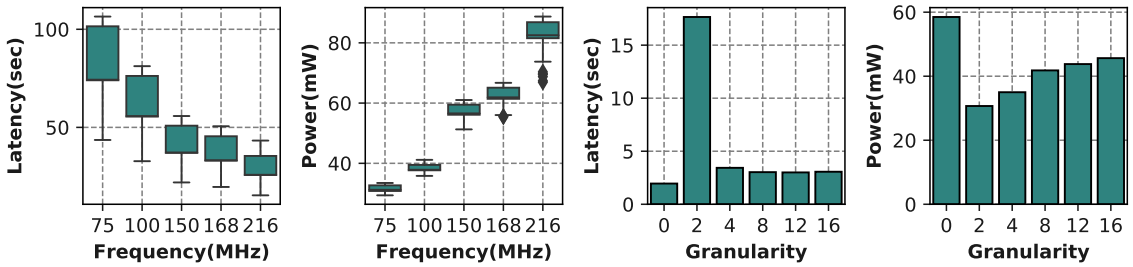


Figure D.4.: Impact of different DAE and clocking configurations on latency and power of depthwise and pointwise layers.

memory/compute-bound regions, thus we can minimize the frequency switching overhead while avoiding high power consumption. However, very high buffer size can lead the cache misses to skyrocket resulting in performance degradation.

After the DAE phase is performed, the CNN layers are encapsulated with a configurable decoupling granularity factor for the layer unrolling. The modified DAE-enabled CNN model is propagated to the next step of our proposed methodology (Step 2.) for effective exploration and configuration of the DAE granularity factor, alongside with the DVFS parameters exploration. DAE-enabled CNNs entail no accuracy drops.

D.3.2. Step 2: DAE and Clocking Co-exploration

At this step, we analyze the performance and energy consumption in a per-layer manner(2A) considering the interplay effects of DAE and clocking scheme configurations. To measure the energy consumption and performance of each layer, we have developed and integrated a custom run-time monitoring mechanism for supporting per-layer monitoring and profiling. Our mechanism relies on the on-board timers of the target MCU, which are triggered in-between the layers' code segments. Furthermore, we take advantage of STM32 MCUs integrated support for power sampling and we monitor the power consumption prior and after the DVFS integration on every CNN layer. The power and performance metrics for the DVFS in each layer are aggregated and utilized for the design space exploration for DAE and clocking configuration.

More specifically, we co-explore and gain insights on the design space defined by the following three key parameters: *i)* the decoupling granularity factor g described in Sec. D.3.1; *ii)* the clock frequency of the `SYSCLK` clock; and *iii)* the selection of parameters for the `PLL` module (namely `PLLM` and `PPLN`), both described in Sec. D.2. Regarding the decoupling granularity, determining the most suitable value per layer depends on both

board-related specifications (e.g., cache size) as well as code-related characteristics (e.g., number of output channels and kernel size). In our case, we examine six different values, i.e., $g \in \{0, 2, 4, 8, 12, 16\}$, where $g = 0$ indicates no DAE optimization and corresponds to the default input model.

Regarding the different clock alternatives, we define two parametric operating modes, namely the *Low Frequency Operation* (LFO) and the *High Frequency Operation* (HFO). LFO exclusively employs the `HSE` clock source at a predefined frequency (50MHz) and aims to reduce power consumption on the board. In contrast, HFO configures the system's clock using the `PLL` circuit, where the final `SYSCLK` value is determined by varying combinations of the `PLLN` and `PLLM` parameters, specifically `PLLN` $\in \{75, 100, 150, 168, 216, 336, 432\}$ and `PLLM` $\in \{25, 50\}$. This distinction between the two operating modes allows us to quickly transition between different `SYSCLK` frequencies, thus, mitigates susceptibility to the switching overhead associated with the `PLL`, as elaborated in Sec. D.2.1. Overall, DVFS switching is performed between the memory (Lst. D.1:3) and compute (Lst. D.1:7) bound regions , in order to exploit the decoupled access-execution transformation optimally, with LFO applied to the memory-bound and HFO to the compute-bound subsegments respectively. A similar approach is followed in pointwise convolution layers.

DSE Insights: Figure D.4 shows the impact of varying operating frequencies (left) and granularity factors g (right) on the layer latency and power consumption. First, we observe that as the number of operating frequency increases, the power consumption is traded for better performance, thus composing the design space. Moreover, changing values of the granularity factor can provide significant variation on the latency and the power consumption. For instance, power consumption can drop up to 54.2% compared to the initial execution. Thus, the effective co-exploration of granularity g and frequency leads to power/latency trade-offs. The result of the DSE is a solution space per layer, where each solution trade-offs between performance and power consumption (2B). In this space, we select the Pareto optimal points, to be propagated to Step 3.

D.3.3. Step 3: QoS-aware Energy Optimization

In the final stage, we determine the optimal frequencies for each layer within the CNN, aiming to minimize the model's total energy consumption while satisfying a predefined latency budget (QoS). We denote the set of all possible frequencies generated either through the `PLL` or the `HSE` clock as F and the set of all possible granularity factors as G. Let n be the total number of layers of the CNN model and $P_k = \{..., p_j^k = \{t_j^k, E_j^k\}, ...\}, k \in$

$\{1, ..., n\}, j \in \{1, ..., |P_k|\}$ be the set of Pareto optimal solutions (from Step 2) of layer k, where t_j^k and E_j^k denote the latency and the energy consumption of the jth pareto optimal solution for layer k when operating with DVFS enabled with an HFO frequency $f \in F$ and granularity $g \in G$. We consider the minimization of the overall energy E of the CNN deployed on the target STM32 MCU, so that the overall execution time T does not go beyond a user-defined QoS. Then, the target optimization problem can be formulated as follows:

$$\text{minimize} \quad E = \Sigma_{k=1}^{n} \Sigma_{j \in P_k} E_j^k x_{kj} \tag{D.2}$$

$$\text{s.t.} \quad T = \Sigma_{k=1}^{n} \Sigma_{j \in P_k} t_j^k x_{kj} \leq QoS \tag{D.3}$$

$$\Sigma_{j \in P_k} x_{kj} = 1, \quad k = 1, ..., n \tag{D.4}$$

$$x_{kj} \in 0, 1, \qquad k = 1, ..., n, j \in P_k. \tag{D.5}$$

We model our problem according to the Multiple-Choice Knapsack Problem (MCKP) [419], which extends the classical knapsack problem by categorizing items into distinct classes(3A). In this formulation, the binary decision of including an item is replaced by the selection of precisely one item from each class and the goal is to maximize the value of items included in the knapsack while not exceeding its size. In our case, each individual class represents the various Pareto optimal solutions p_j^k per layer k. Each item in the class is characterized by its own value (i.e., energy consumption E_j^k) and size (i.e., latency t_j^k). Our goal is to minimize the overall energy consumption (E) while adhering to the size constraint ($T < QoS$). We convert our minimization objective to a maximization one using the transformation found in [419]. Last, we solve the optimization problem using a pseudo-polynomial time solution based on a dynamic programming (DP) approach.

D.4. Experimental Setup and Evaluation

Experimental Setup: Our experimental evaluation is conducted on an STM32F767ZI Nucleo board, equipped with an ARM Cortex M7 CPU featuring a 16KB L1-cache. The board incorporates a High-Speed External (`HSE`) clock, ranging from 1MHz to 50MHz. For power consumption monitoring, we employed the INA219 power sensor. To mitigate potential variations arising from temperature-induced power fluctuations, we systematically compared each power measurement with the power consumption of the baseline input model at the corresponding timestamp. Our proposed methodology is evaluated over three pre-trained CNN models, namely Visual Wake Words (`VWW`), Person Detection (`PD`), and Mobilenet-V2 (`MBV2`), derived from the MCUNet [1] inference library.

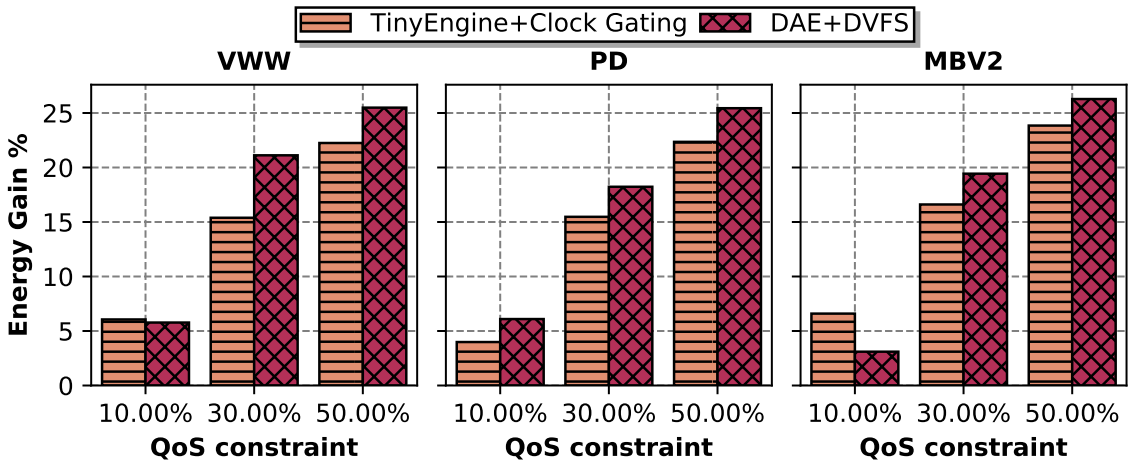


Figure D.5.: Energy consumption gains of our approach over the TinyEngine [1] baseline. We compare against TinyEngine with Clock Gating over the examined CNN models.

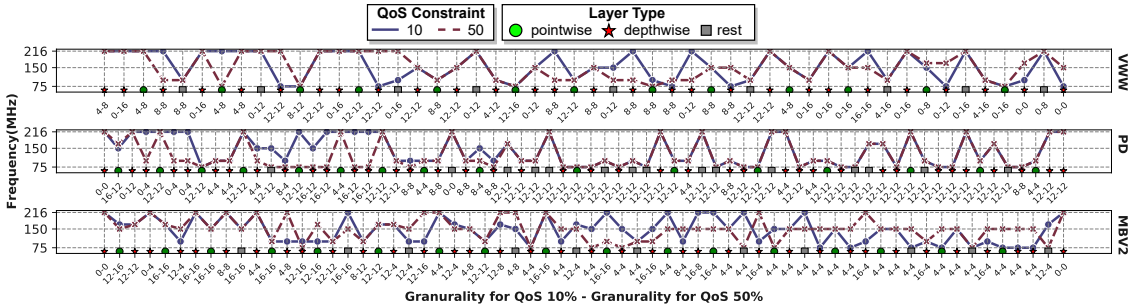


Figure D.6.: Frequency distribution throughout layer progression over the examined CNN models for 10% and 50% QoS constraints.

Hardware-Aware Neural Architecture Search and linear int8 quantization have been used for quantization. We conduct a comparative analysis between our approach and the state-of-the-art `TinyEngine` [1], which serves as the baseline for evaluation. Our experiments are conducted in an iso-latency execution scenario, where we measure energy consumption for a certain period specified by a QoS constraint. In the case of TinyEngine, this entails the board remaining in an idle state with a constant frequency of 216MHz after an inference, until the QoS threshold is met. We also consider `TinyEngine` enhanced with clock gating, a technique designed to optimize power consumption by selectively deactivating non-utilized board clocks and the voltage regulator, thus minimizing power leakage throughout the CNN inference.

Energy Comparison and Analysis: Figure D.5 illustrates a comparison of energy con-

sumption between our proposed approach, and the two configurations of the `TinyEngine`: one without any optimization and the other with clock gating applied. This evaluation encompasses various CNN inference models, each subject to discrete Quality of Service (QoS) constraints set at 10% (tight), 30% (moderate), and 50% (relaxed). X-axis illustrates the QoS constraints, while Y-axis represents the normalized energy consumption. Our proposed approach surpasses both instances of the `TinyEngine`, exhibiting a reduction in energy consumption up to 25.2%. Additionally, compared to the `TinyEngine` equipped with clock gating we achieve up to 7.2% less energy consumption. Furthermore, our observations indicate that relaxing the QoS constraints can lead to a notable reduction in energy consumption, albeit at the cost of some performance trade-offs. For instance, when examining the Mobilenet-V2 model, the energy consumption of our approach under a relaxed 50% QoS constraint decreases to 20.4% compared to the stringent 10% constraint.

Frequency Scaling Analysis: Figure D.6 illustrates the HFO for each examined CNN. X axis indicates the corresponding layer type as the CNN execution proceeds and the granularities selected for 10% and 50% QoS constraint, respectively, while Y axis shows the operating frequency per layer. The LFO configuration at 50MHz of the memory-bound segment of each layer is excluded for simplicity. The observations derived are the following. Firstly, the operational frequency is configured to maximum (216MHz) mostly for performing pointwise convolutions, i.e. 58.8% against 21.4% for depthwise convolutions. The latter are less compute-intensive, thus decreasing the operational frequency will not lead to significant performance degradation. Furthermore, the 46.1% of the pointwise convolutions and 43.4% of the depthwise convolutions are executed over the lowest operating frequencies, i.e. 75MHz and 100MHz, aiming to boost the optimization objective of power minimization. Last but not least, we investigate the impact of QoS constraints on the operating frequency. Our experiments indicate that 18.6% more layers are operating at 216MHz for tight constraints (10%). Regarding the granularity analysis, for the relaxed QoS(50%), 22.3% more layers operate with granularity factor 16, compared to 10% constraint. This is due to the fact that there is higher space to trade latency, thus the computation-bound parts are split to bigger segments, aiming to minimize switching overhead and provide power reduction.

D.5. Conclusion

In this work, we present a novel end-to-end methodology that exploits DVFS for optimizing the energy consumption of CNN inference on low-end STM32 MCUs. Our approach strongly leverages Decoupled Access Execute techniques to discretize memory-bound and

compute-bound layer segments. Our approach outperforms existing state-of-the-art approaches by achieving up to 25.2% less energy consumption.

Abbreviations and Acronyms

AI	Artificial Intelligence
ASIC	Application System Integrated Chip
AWS	Amazon Web Services
CAGR	Compound Annual Growth Rate
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CXL	Compute Express Link
DAE	Decoupled Access Execute
DC	Data Center
DCPM	Data Center Persistent Memory
DDTR	Dynamic Data Type Refinement
DIMM	Dual In-line Memory
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
DSE	Design Space Exploration
DVFS	Dynamic Voltage Frequency Scaling

FeRAM	Ferroelectric RAM
GPU	Graphics Processing Unit
HDD	Hard-Disk Drive
HFO	High Frequency Operation
HPC	High-Performance Computing
HSE	High Speed Internal
HSI	High Speed External
HW	Hardware
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IPA	Intelligent Personal Assistant
KPI	Key Performance Indicator
LC	Latency Critical
LFO	Low Frequency Operation
LLC	Last-Level Cache
LLM	Large Language Model
LSTM	Long Short-Term Memory
LRU	Least Recently-Used
MBA	Memory Bandwidth Allocation

MCU	Micro-Controller Unit
ML	Machine Learning
NN	Neural Network
NUMA	Non Unified Memory Access
NVM	Non-Volatile Memory
NVMe	NVM express
PCM	Performance Counter Monitoring
PCM	Phase-Change Memory
QoS	Quality of Service
PLL	Phase-Locked Loop
PM	Persistent Memory
RAM	Random Access Memory
ReRAM	Resistive RAM
RF	Random Forest
RMSE	Root Mean Square Error
RNN	Recurrent Neural Networks
SCM	Storage Class Memory
SLA	Service Level Agreements
SSD	Solid-State Drive
STL	Standard Template Library

STT-MRAM	Spin-Transfer Torque Magnetic RAM
SW	Software
TSMC	Taiwan Semiconductor Manufacturing Co.
TCO	Total Cost Ownership
TPC	Transaction Processing Performance Council
TPU	Tensor Processing Unit
YCSB	Yahoo! Cloud Serving Benchmark
ZB	Zettabytes

Publications

Journals

1. Katsaragakis Manolis, Papadopoulos Lazaros, Konijnenburg Mario, Catthoor Francky, & Soudris Dimitrios (2023). A memory footprint optimization framework for Python applications targeting edge devices. Journal of Systems Architecture, 142, 102936. [212]

2. Katsaragakis Manolis, Konstantina Skovola, Christos Baloukas, Papadopoulos Lazaros, Catthoor Francky, & Soudris Dimitrios. Performance and Energy Consumption Exploration Methodology for Persistent B+-tree Indexes over Heterogeneous DRAM/NVM Systems, IEEE Transactions on Sustainable Computing(TSuC), 2024 (Under review).

3. Katsaragakis Manolis, Christos Baloukas, Papadopoulos Lazaros, Catthoor Francky, & Soudris Dimitrios. Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems, ACM Transactions on Architecture and Code Optimization (TACO), 2024 (Under review)

4. Katsaragakis Manolis, Christos Lamprakos, Peter Kourzanov, Manu Komalan Perumkunnil, Lazaros Papadopoulos, Francky Catthoor & Dimitrios Soudris. Optimizing Cache Configurations via Dynamic Memory Management in Edge Computing Systems, IEEE Embedded Systems Letter (ESL), 2024 (Under review)

5. Katsaragakis Manolis, Christos Baloukas, Lazaros Papadopoulos, Francky Catthoor & Dimitrios Soudris. Software-based Data Placement over Heterogeneous Memory Systems (To be submitted)

Conferences

1. Katsaragakis Manolis, Masouros Dimosthenis, Tsoutsouras Vasileios, Samie Farzad, Bauer Lars, Henkel Jorg., & Soudris Dimitrios (2019, March). DMRM: Distributed market-based resource management of edge computing systems. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1391-1396). IEEE. [343]

2. Katsaragakis Manolis, Papadopoulos Lazaros, Konijnenburg Mario, Catthoor Francky, & Soudris Dimitrios (2020, October). Memory footprint optimization techniques for machine learning applications in embedded systems. In 2020 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-4). IEEE. [143]

3. Achilleas Tzenetopoulos, Evangelos Apostolakis, Aphrodite Tzomaka, Christos Papakostopoulos, Konstantinos Stavrakakis, Manolis Katsaragakis, Ioannis Oroutzoglou, Dimosthenis Masouros, Sotirios Xydis, Dimitrios Soudris: FaaS and Curious: Performance Implications of Serverless Functions on Edge Computing Platforms. ISC Workshops 2021: 428-438 [420]

4. Karteris Antonis, Katsaragakis Manolis, Masouros Dimosthenis, & Soudris Dimitrios (2022, March). SGRM: stackelberg game-based resource management for edge computing systems. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1203-1208). IEEE. [361]

5. Katsaragakis Manolis, Papadopoulos Lazaros, Baloukas Christos, & Soudris Dimitrios (2022, March). 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) (pp. 748-753). IEEE. [161]

6. Katsaragakis Manolis, Baloukas Christos, Papadopoulos Lazaros, Kantere Verena, Catthoor Francky, & Soudris Dimitrios (2022, December). 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) (pp. 75-84). IEEE. [160]

7. Katsaragakis Manolis, Stavrakakis Konstantinos, Masouros Dimosthenis, Papadopoulos Lazaros, & Soudris Dimitrios (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 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik. [190]

8. Kakolyris Andreas Kosmas, Katsaragakis Manolis, Masouros Dimosthenis, & Soudris Dimitrios (2023, April). RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems. In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1-6). IEEE. [374]

9. Katsaragakis Manolis, Masouros Dimosthenis, Papadopoulos Lazaros, Catthoor Francky, & Soudris Dimitrios (2023, May). On the Implications of Heterogeneous Memory Tiering on Spark In-Memory Analytics. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 945-952). IEEE. [240]

10. Ferikoglou Aggelos, Chrysomeris Panos, Tzenetopoulos Achilleas, Katsaragakis Manolis, Masouros Dimosthenis, & Soudris Dimitrios (2023, July). IRIS: Interference and Resource Aware Predictive Orchestration for ML Inference Serving. In 2023 IEEE 16th International Conference on Cloud Computing (CLOUD) (pp. 1-12). IEEE. (Best Paper Award) [421]

11. Elisavet Lydia Alvanaki, Manolis Katsaragakis, Dimosthenis Masouros, Sotirios Xydis & Dimitrios Soudris. Decoupled Access-Execute enabled DVFS for tinyML deployments on STM32 microcontrollers. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1-6). IEEE. [267]

12. Christos Lamprakos, Panagiotis Xanthopoulos, Manolis Katsaragakis, Sotirios Xydis, Francky Catthoor, Dimitrios Soudris. Practical Static Offset Assignment for Deep Learning: Close—But No Cigar Yet, on CM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25) (Under review).

13. Manolis Katsaragakis, Aris Grivas, Christos Baloukas, Lazaros Papadoulos, Francky Catthoor & Dimitrios Soudris. Lifetime-aware Data Placement over Heterogeneous Memory Systems (To be submitted).

Bibliography

[1] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, *et al.*, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11711–11722, 2020.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[3] Z. Pei, F. Catthoor, Z. Tokei, and C. Pan, "Beyond-cu intermediate-length interconnect exploration for sram application," *IEEE Transactions on Nanotechnology*, vol. 21, pp. 367–373, 2022.

[4] Z. Pei, M. Mayahinia, H.-H. Liu, M. Tahoori, F. Catthoor, Z. Tokei, and C. Pan, "Graphene-based interconnect exploration for large sram caches for ultrascaled technology nodes," *IEEE Transactions on Electron Devices*, vol. 70, no. 1, pp. 230–238, 2022.

[5] Z. Pei, M. Mayahinia, H.-H. Liu, M. Tahoori, F. Catthoor, Z. Tokei, and C. Pan, "Technology/memory co-design and co-optimization using e-tree interconnect," in *Proceedings of the Great Lakes Symposium on VLSI 2023*, pp. 159–162, 2023.

[6] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, "Nom: Network-on-memory for inter-bank data transfer in highly-banked memories," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 80–83, 2020.

[7] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, 2017.

[8] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, 2024.

[9] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[10] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *Proceedings*

of the 2018 Conference of the ACM Special Interest Group on Data Communication, pp. 327–341, 2018.

[11] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 574–587, 2023.

[12] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, 2023.

[13] J. Wenskovitch and C. North, "Interactive artificial intelligence: designing for the" two black boxes" problem," *Computer*, vol. 53, no. 8, pp. 29–39, 2020.

[14] A. Aiken, M. Fähndrich, and R. Levien, "Better static memory management: Improving region-based analysis of higher-order languages," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pp. 174–185, 1995.

[15] D. Walker, K. Crary, and G. Morrisett, "Typed memory management via static capabilities," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 4, pp. 701–771, 2000.

[16] R. Iyer, H. Wang, and L. N. Bhuyan, "Design and analysis of static memory management policies for cc-numa multiprocessors," *Journal of systems architecture*, vol. 48, no. 1-3, pp. 59–80, 2002.

[17] F. Lai, D. Schmidt, and O. Chipara, "Static memory management for efficient mobile sensing applications," in *2015 International Conference on Embedded Software (EMSOFT)*, pp. 187–196, IEEE, 2015.

[18] D. Evans, "Static detection of dynamic memory errors," *ACM SIGPLAN Notices*, vol. 31, no. 5, pp. 44–53, 1996.

[19] I. Lamprou, Z. Zhang, J. de Juan, H. Yang, Y. Lai, E. Filhol, and C. Bastoul, "Safe optimized static memory allocation for parallel deep learning," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[20] J. Zhu, "Static memory allocation by pointer analysis and coloring," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 785–790, IEEE, 2001.

[21] Z. Dvir, A. Golovnev, and O. Weinstein, "Static data structure lower bounds imply rigidity," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pp. 967–978, 2019.

[22] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, Dttawa, Dntorio, Canada, 2007.

[24] J. Yang, B. Li, and D. J. Lilja, "Exploring performance characteristics of the optane 3d xpoint storage technology," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 5, no. 1, pp. 1–28, 2020.

[25] S.-S. Sheu, K.-H. Cheng, M.-F. Chang, P.-C. Chiang, W.-P. Lin, H.-Y. Lee, P.-S. Chen, Y.-S. Chen, T.-Y. Wu, F. T. Chen, *et al.*, "Fast-write resistive ram (rram) for embedded applications," *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 64–71, 2010.

[26] "Performance counter monitor." https://github.com/opcm/pcm.

[27] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, *et al.*, "Spin-transfer torque magnetic random access memory (stt-mram)," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, p. 13, 2013.

[28] T. Ali, P. Polakowski, S. Riedel, T. Büttner, T. Kämpfe, M. Rudolph, B. Pätzold, K. Seidel, D. Löhr, R. Hoffmann, *et al.*, "High endurance ferroelectric hafnium oxide-based fefet memory without retention penalty," *IEEE Transactions on Electron Devices*, vol. 65, no. 9, pp. 3769–3774, 2018.

[29] "Intel optane dc product specs." https://www.mouser.com/datasheet/2/612/optane-dc-persistent-memory-brief-1710301.pdf.

[30] "Daos architecture."

[31] R. Stevens, J. Ramprakash, P. Messina, M. Papka, and K. Riley, "Aurora: Argonne's next-generation exascale supercomputer," tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States), 2019.

[32] D. D. Sharma and S. Tavallaei, "Compute express link 2.0 white paper," *CXL. Retrieved October*, vol. 31, p. 2021, 2020.

[33] "Samsung Memory Expander." https://www.nextplatform.com/2022/08/23/samsung-shows-off-cxl-server-memory-expander/.
Accessed: March 2023.

[34] J. Chou, L. Chen, H. Ding, J. Tu, and B. Xu, "A method of optimizing django based on greedy strategy," in *2013 10th web information system and application conference*, pp. 176–179, IEEE, 2013.

[35] N. Moshiri, "Treeswift: A massively scalable python tree package," *SoftwareX*, vol. 11, p. 100436, 2020.

[36] A. R. Shajii, I. Numanagić, A. T. Leighton, H. Greenyer, S. Amarasinghe, and B. Berger, "A python-based optimization framework for high-performance genomics," *bioRxiv*, 2020.

[37] R. Monat, A. Ouadjaout, and A. Miné, "Value and allocation sensitivity in static python analyses," in *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pp. 8–13, 2020.

[38] M. Jump and K. S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 31–38, 2007.

[39] L. Cen, R. Marcus, H. Mao, J. Gottschlich, M. Alizadeh, and T. Kraska, "Learned garbage collection," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 38–44, 2020.

[40] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and exploiting contiguity for fast memory virtualization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 515–528, IEEE, 2020.

[41] D. Mishra and P. Kulkarni, "A survey of memory management techniques in virtualized systems," *Computer Science Review*, vol. 29, pp. 56–73, 2018.

[42] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 462–477, 2019.

[43] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "Dptree: differential indexing for persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.

[44] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 424–439, 2021.

[45] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[46] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 183–196, 2012.

[47] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[48] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 371–386, 2016.

[49] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *arXiv preprint arXiv:2003.07302*, 2020.

[50] P. Zuo, Y. Hua, and J. Wu, "{Write-Optimized} and {High-Performance} hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 461–476, 2018.

[51] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "{Write-Optimized} dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pp. 31–44, 2019.

[52] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 631–644, 2015.

[53] R. Lasch, T. Legler, N. May, B. Scheirle, and K.-U. Sattler, "Cost modelling for optimal data placement in heterogeneous main memory," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2867–2880, 2022.

[54] R. Lasch, R. Schulze, T. Legler, and K.-U. Sattler, "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 2021)*, pp. 1–8, 2021.

[55] A. Hassan, D. S. Nikolopoulos, and H. Vandierendonck, "Fast and energy-efficient olap data management on hybrid main memory systems," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1597–1611, 2019.

[56] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 392–407, 2021.

[57] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 293–304, 2020.

[58] J. Liu, D. Li, R. Gioiosa, and J. Li, "Athena: High-performance sparse tensor contraction sequence on heterogeneous memory," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 190–202, 2021.

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

[60] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 401–413, IEEE, 2018.

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

[62] Z. Xie, J. Liu, J. Li, and D. Li, "Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness," 2023.

[63] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Ngraph: Parallel graph processing in hybrid memory systems," *IEEE Access*, vol. 7, pp. 103517–103529, 2019.

[64] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Hngraph: Parallel graph processing in hybrid memory based numa systems," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 388–397, IEEE, 2021.

[65] E. Ozfatura, S. Ulukus, and D. Gündüz, "Straggler-aware distributed learning: Communication–computation latency trade-off," *Entropy*, vol. 22, no. 5, p. 544,

2020.

[66] L. Chen, J. Zhao, C. Wang, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, G. H. Xu, *et al.*, "Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories," *ACM Transactions on Computer Systems (TOCS)*, vol. 39, no. 1-4, pp. 1–38, 2022.

[67] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "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*, pp. 347–362, 2019.

[68] W. Rang, D. Yang, Z. Li, and D. Cheng, "Scalable data management on hybrid memory system for deep neural network applications," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 1470–1480, IEEE, 2021.

[69] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 598–611, IEEE, 2021.

[70] M. Han, J. Hyun, S. Park, and W. Baek, "Hotness-and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 377–391, 2019.

[71] J. Ren, J. Luo, I. Peng, K. Wu, and D. Li, "Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 203–214, 2021.

[72] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for gpu accelerated computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

[73] B. Simner, A. Armstrong, J. Pichon-Pharabod, C. Pulte, R. Grisenthwaite, and P. Sewell, "Relaxed virtual memory in armv8-a," in *European Symposium on Programming*, pp. 143–173, Springer International Publishing Cham, 2022.

[74] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *International Conference on Machine Learning*, pp. 1919–1928, PMLR, 2018.

[75] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher: a case study," in *Proceedings of the International Symposium on Memory Systems*, pp. 305–311, 2017.

[76] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 37–48, 2019.

[77] S. Han and Y. Jiang, "Risc-v-based evaluation and strategy exploration of mram triple-level hybrid cache systems," *IEEE Transactions on Very Large Scale Inte-*

gration (VLSI) Systems, 2023.

[78] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. Wong, "Boom-explorer: Risc-v boom microarchitecture design space exploration framework," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.

[79] C. Bai, J. Zhai, Y. Ma, W. Lyu, Z. Chen, B. Yu, and M. D. Wong, "Risc-v microarchitecture exploration via reinforcement learning," 2022.

[80] H. Jang, K. Han, S. Lee, J.-J. Lee, S.-Y. Lee, J.-H. Lee, and W. Lee, "Developing a multicore platform utilizing open risc-v cores," *IEEE Access*, vol. 9, pp. 120010–120023, 2021.

[81] Z. Fu, M. Wang, Y. Zhang, and Z. Tan, "Cache coherent framework for risc-v many-core systems,"

[82] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[83] A. Makarov, V. Sverdlov, and S. Selberherr, "Emerging memory technologies: Trends, challenges, and modeling methods," *Microelectronics Reliability*, vol. 52, no. 4, pp. 628–634, 2012.

[84] K. Kim, "Future memory technology: Challenges and opportunities," in *2008 International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*, pp. 5–9, IEEE, 2008.

[85] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale research letters*, vol. 9, pp. 1–33, 2014.

[86] E. Patel and D. S. Kushwaha, "Clustering cloud workloads: K-means vs gaussian mixture model," *Procedia computer science*, vol. 171, pp. 158–167, 2020.

[87] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for olap workloads," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 339–351, 2021.

[88] T. D. Doudali and A. Gavrilovska, "Comerge: Toward efficient data placement in shared heterogeneous memory systems," in *Proceedings of the International Symposium on Memory Systems*, pp. 251–261, 2017.

[89] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 37–48, 2019.

[90] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "Hinuma: Numa-aware data placement and migration in hybrid memory systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 367–375, IEEE, 2019.

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

[92] T. C. Effler, A. P. Howard, T. Zhou, M. R. Jantz, K. A. Doshi, and P. A. Kulka-

rni, "On automated feedback-driven data placement in multi-tiered memory," in *Architecture of Computing Systems–ARCS 2018: 31st International Conference, Braunschweig, Germany, April 9–12, 2018, Proceedings 31*, pp. 181–194, Springer, 2018.

[93] T. C. Effler, B. Kammerdiener, M. R. Jantz, S. Sengupta, P. A. Kulkarni, K. A. Doshi, and T. Jones, "Evaluating the effectiveness of program data features for guiding memory management," in *Proceedings of the International Symposium on Memory Systems*, pp. 383–395, 2019.

[94] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, "Reliability-aware data placement for heterogeneous memory architecture," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 583–595, IEEE, 2018.

[95] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi, "Cross-layer memory management for managed language applications," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 488–504, 2015.

[96] M. Katsaragakis, K. Stavrakakis, D. Masouros, L. Papadopoulos, and D. Soudris, "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 2023)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[97] D. Knyaginin and P. Stenstrom, "Rock: a framework for pruning the design space of hybrid main memory systems," in *Proceedings of the International Symposium on Memory Systems*, pp. 337–347, 2017.

[98] M. Laghari and D. Unat, "Object placement for high bandwidth memory augmented with high capacity memory," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 129–136, IEEE, 2017.

[99] H. T. Mai, K. H. Park, H. S. Lee, C. S. Kim, M. Lee, and S. J. Hur, "Dynamic data migration in hybrid main memories for in-memory big data storage," *Etri Journal*, vol. 36, no. 6, pp. 988–998, 2014.

[100] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 875–890, 2020.

[101] D. Moura, V. Petrucci, and D. Mosse, "Learning to rank graph-based application objects on heterogeneous memories," in *The International Symposium on Memory Systems*, pp. 1–14, 2021.

[102] A. A. García, D. May, and E. Nutting, "Garbage collection for edge computing,"

in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 319–319, IEEE, 2020.

[103] E. Oyekanlu, "Predictive edge computing for time series of industrial iot and large scale critical infrastructure based on open-source software analytic of big data," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 1663–1669, IEEE, 2017.

[104] M. M. Rahman, M. H. Sharker, and R. Paudel, "Active and collaborative learning based dynamic instructional approach in teaching introductory computer science course with python programming," in *2020 IEEE Integrated STEM Education Conference (ISEC)*, pp. 1–7, IEEE, 2020.

[105] R. Power and A. Rubinsteyn, "How fast can we make interpreted python?," *arXiv preprint arXiv:1306.6047*, 2013.

[106] T. W. Barr, R. Smith, and S. Rixner, "Design and implementation of an embedded python run-time system," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 297–308, 2012.

[107] T. Baroudi, V. Loechner, and R. Seghir, "Static versus dynamic memory allocation: a comparison for linear algebra kernels," in *IMPACT 2020, in conjunction with HiPEAC 2020*, 2020.

[108] C. K. Chui, Y.-T. Lin, and H.-T. Wu, "Real-time dynamics acquisition from irregular samples—with application to anesthesia evaluation," *Analysis and Applications*, vol. 14, no. 04, pp. 537–590, 2016.

[109] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 451–461, IEEE, 2020.

[110] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: Improving memory management for mobile operating systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 873–887, 2020.

[111] K. Wu and D. Li, "Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory for high performance computing," *Journal of Computer Science and Technology*, vol. 36, pp. 90–109, 2021.

[112] Y. Pisarchyk and J. Lee, "Efficient memory management for deep neural net inference," *arXiv preprint arXiv:2001.03288*, 2020.

[113] S. Resch, H. Cilasun, Z. Chowdhury, M. Zabihi, Z. Zhao, J.-P. Wang, S. Sapatnekar, and U. R. Karpuzcu, "On endurance of processing in (nonvolatile) memory," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–13, 2023.

[114] S. Kargar and F. Nawab, "Extending the lifetime of nvm: challenges and opportunities," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3194–3197, 2021.

[115] R. Strenz, "Review and outlook on embedded nvm technologies–from evolution

to revolution," in *2020 IEEE International Memory Workshop (IMW)*, pp. 1–4, IEEE, 2020.

[116] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "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*, pp. 288–303, 2019.

[117] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, p. 1, ACM, 2013.

[118] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pp. 191–196, 2012.

[119] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[120] S.-M. Syu, Y.-H. Shao, and I.-C. Lin, "High-endurance hybrid cache design in cmp architecture with cache partitioning and access-aware policy," in *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI*, pp. 19–24, 2013.

[121] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," *ACM SIGARCH computer architecture news*, vol. 37, no. 3, pp. 34–45, 2009.

[122] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (l 3 cs): Low-leakage sram, low write-energy stt-ram, and refresh-optimized edram," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 143–154, IEEE, 2013.

[123] Y.-T. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman, "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 45–50, IEEE, 2012.

[124] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 50–61, IEEE, 2011.

[125] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement," in *2008 45th ACM/IEEE Design Automation Conference*, pp. 554–559, IEEE, 2008.

[126] M. H. Kryder and C. S. Kim, "After hard drives—what comes next?," *IEEE Transactions on Magnetics*, vol. 45, no. 10, pp. 3406–3413, 2009.

[127] B. Nitin, W. Randy, I. Shinichiro, F. Eiji, R. Shibata, S. Yumiko, and O. Megumi, "Ddr5 design challenges," in *2018 IEEE 22nd Workshop on Signal and Power Integrity (SPI)*, pp. 1–4, IEEE, 2018.

[128] D.-H. Kim, B. Song, H.-a. Ahn, W. Ko, S. Do, S. Cho, K. Kim, S.-H. Oh, H.-Y. Joo, G. Park, *et al.*, "A 16gb 9.5 gb/s/pin lpddr5x sdram with low-power schemes exploiting dynamic voltage-frequency scaling and offset-calibrated readout sense amplifiers in a fourth generation 10nm dram process," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, pp. 448–450, IEEE, 2022.

[129] J. Choe, "Memory technology 2021: Trends & challenges," in *2021 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, pp. 111–115, IEEE, 2021.

[130] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, *et al.*, "Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–26, IEEE, 2021.

[131] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, *et al.*, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2021.

[132] S. Park, H. Kim, K. Kim, J. So, J. Ahn, W. Lee, D. Kim, Y. Kim, J. Seok, J. Lee, *et al.*, "Scaling of memory performance and capacity with cxl memory expander," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, pp. 1–27, IEEE Computer Society, 2022.

[133] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The sap hana database–an architecture overview.," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[134] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, *et al.*, "Memcached design on high performance rdma capable interconnects," in *2011 International Conference on Parallel Processing*, pp. 743–752, IEEE, 2011.

[135] M. Hennecke, "Daos: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, p. 40, 2020.

[136] https://www.tsmc.com/english.

[137] H.-H. Liu, S. M. Salahuddin, D. Abdi, R. Chen, P. Weckx, P. Matagne, and F. Catthoor, "Extended methodology to determine sram write margin in resistance-dominated technology node," *IEEE Transactions on Electron Devices*, vol. 69, no. 6, pp. 3113–3117, 2022.

[138] M. K. Gupta, P. Weckx, P. Schuddinck, D. Jang, B. Chehab, S. Cosemans, J. Ryckaert, and W. Dehaene, "The complementary fet (cfet) 6t-sram," *IEEE Transactions on Electron Devices*, vol. 68, no. 12, pp. 6106–6111, 2021.

[139] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[140] "Pmdk." https://github.com/pmem/pmdk.

[141] A. Bartzas, S. Mamagkakis, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris, and A. Thanailakis, "Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1, pp. 6–pp, IEEE, 2006.

[142] T. Papastergiou, L. Papadopoulos, and D. Soudris, "Platform-aware dynamic data type refinement methodology for radix tree data structures," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 78–85, IEEE, 2015.

[143] M. Katsaragakis, L. Papadopoulos, M. Konijnenburg, F. Catthoor, and D. Soudris, "Memory footprint optimization techniques for machine learning applications in embedded systems," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, IEEE, 2020.

[144] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quirós, A. Bartzas, F. Catthoor, and D. Soudris, *Dynamic memory management for embedded systems*. Springer, 2015.

[145] A. Andrzejak, F. Eichler, and M. Ghanavati, "Detection of memory leaks in c/c++ code via machine learning," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 252–258, IEEE, 2017.

[146] B. Powers, D. Tench, E. D. Berger, and A. McGregor, "Mesh: compacting memory management for c/c++ applications," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 333–346, 2019.

[147] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for c++ server workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 541–556, 2020.

[148] C. Baloukas, L. Papadopoulos, R. Pyka, D. Soudris, and P. Marwedel, "An automatic framework for dynamic data structures optimization in c," in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pp. 155–160, Sep. 2010.

[149] M. Ismail and G. E. Suh, "Quantitative overhead analysis for python," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 36–47, IEEE, 2018.

[150] G. Barany, "Python interpreter performance deconstructed," in *Proceedings of the Workshop on Dynamic Languages and Applications*, pp. 1–9, 2014.

[151] B. Ilbeyi, C. F. Bolz-Tereick, and C. Batten, "Cross-layer workload characterization of meta-tracing jit vms," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 97–107, IEEE, 2017.

[152] M. Ismail and G. E. Suh, "Hardware-software co-optimization of memory manage-

ment in dynamic languages," *ACM SIGPLAN Notices*, vol. 53, no. 5, pp. 45–58, 2018.

[153] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: Effective selection of data structures," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 86–97, 2011.

[154] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "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*, pp. 757–773, 2020.

[155] H. Bu, M.-K. Dong, J.-F. Yi, B.-Y. Zang, and H.-B. Chen, "Revisiting persistent indexing structures on intel optane dc persistent memory," *Journal of Computer Science and Technology*, vol. 36, pp. 140–157, 2021.

[156] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 194–209, 2021.

[157] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 789–806, 2020.

[158] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.

[159] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 785–798, 2021.

[160] M. Katsaragakis, C. Baloukas, L. Papadopoulos, V. Kantere, F. Catthoor, and D. Soudris, "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)*, pp. 75–84, IEEE, 2022.

[161] M. Katsaragakis, L. Papadopoulos, C. Baloukas, and D. Soudris, "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)*, pp. 748–753, IEEE, 2022.

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

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

[164] V. Mironov, I. Chernykh, I. Kulikov, A. Moskovsky, E. Epifanovsky, and A. Kudryavtsev, "Performance evaluation of the intel optane dc memory with scientific benchmarks," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 1–6, IEEE, 2019.

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

[166] I. Peng, K. Wu, J. Ren, D. Li, and M. Gokhale, "Demystifying the performance of hpc scientific applications on nvm-based memory systems," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 916–925, IEEE, 2020.

[167] Y. He, D. Lu, K. Huang, and T. Wang, "Evaluating persistent memory range indexes: Part two," *arXiv preprint arXiv:2201.13047*, 2022.

[168] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "utree: a persistent b+-tree with low tail latency," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.

[169] M. Weiland and B. Homoelle, "Usage scenarios for byte-addressable persistent memory in high-performance and data intensive computing," *Journal of Computer Science and Technology*, vol. 36, no. 1, pp. 110–122, 2021.

[170] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 488–505, 2022.

[171] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 293–304, 2020.

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

[173] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, pp. 417–427, 2019.

[174] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "Hinuma: Numa-aware data placement and migration in hybrid memory systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 367–375, IEEE, 2019.

[175] Z. Li and M. Wu, "Transparent and lightweight object placement for managed workloads atop hybrid memories," in *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 72–80, 2022.

[176] N. S. Das and A. Chowdhury, "A survey paper on organic–inorganic layered nanohybrid embedded polymer-based non-volatile resistive switching memory de-

vices," *Recent Trends in Thermal Engineering: Select Proceedings of ICAST 2020*, pp. 181–192, 2022.

[177] H.-K. Liu, D. Chen, H. Jin, X.-F. Liao, B. He, K. Hu, and Y. Zhang, "A survey of non-volatile main memory technologies: state-of-the-arts, practices, and future directions," *Journal of Computer Science and Technology*, vol. 36, pp. 4–32, 2021.

[178] K. Suzuki and S. Swanson, "A survey of trends in non-volatile memory technologies: 2000-2014," in *2015 IEEE International Memory Workshop (IMW)*, pp. 1–4, IEEE, 2015.

[179] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[180] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, "Emerging nvm: A survey on architectural integration and research challenges," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 2, pp. 1–32, 2017.

[181] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1524–1537, 2014.

[182] S. Mittal, "A survey of soft-error mitigation techniques for non-volatile memories," *Computers*, vol. 6, no. 1, p. 8, 2017.

[183] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2015.

[184] G. O. Puglia, A. F. Zorzo, C. A. De Rose, T. Perez, and D. Milojicic, "Non-volatile memory file systems: a survey," *IEEE Access*, vol. 7, pp. 25836–25871, 2019.

[185] A. Ambore *et al.*, "A survey on data placement strategy in big data heterogeneous environments," in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 439–443, IEEE, 2019.

[186] A. Kaur, P. Gupta, M. Singh, and A. Nayyar, "Data placement in era of cloud computing: a survey, taxonomy and open research issues," *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 377–398, 2019.

[187] P. Machanick, "Approaches to addressing the memory wall," *School of IT and Electrical Engineering, University of Queensland*, 2002.

[188] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones, "Online application guidance for heterogeneous memory systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–27, 2022.

[189] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[190] M. Katsaragakis, K. Stavrakakis, D. Masouros, L. Papadopoulos, and D. Soudris, "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 Archi-*

tectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[191] "Intel vtune profiler." https://software.intel.com/en-us/vtune.

[192] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[193] P. Olivier, A. F. Mehrab, S. Lankes, M. L. Karaoui, R. Lyerly, and B. Ravindran, "Hexo: Offloading hpc compute-intensive workloads on low-cost, low-power embedded systems," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 85–96, 2019.

[194] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, pp. 1–42, 2010.

[195] M. Abadi, "Tensorflow: learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pp. 1–1, 2016.

[196] F. Chen, M. P. Mesnier, and S. Hahn, "A protected block device for persistent memory," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, IEEE, 2014.

[197] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proceedings of the 16th Annual Middleware Conference*, pp. 37–49, ACM, 2015.

[198] Z. Duan, H. Liu, X. Liao, and H. Jin, "Hme: A lightweight emulator for hybrid memory," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1375–1380, IEEE, 2018.

[199] P. Chakraborty and P. R. Panda, "Spm-sieve: A framework for assisting data partitioning in scratch pad memory based systems," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 1–10, IEEE, 2013.

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

[201] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "{MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} based {KV} stores with matrix container in {NVM}," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 17–31, 2020.

[202] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with nvm," *ACM Transactions on Storage (TOS)*, vol. 13, no. 2, pp. 1–13, 2017.

[203] "Chatgpt."

[204] A. Moallemy-Oureh, S. Beddar-Wiesing, R. Nather, and J. M. Thomas, "Fdgnn: Fully dynamic graph neural network," *arXiv preprint arXiv:2206.03469*, 2022.

[205] J. den Hartog, N. Zannone, *et al.*, "Security and privacy for innovative automotive

applications: A survey," *Computer Communications*, vol. 132, pp. 17–41, 2018.

[206] W. J. d. S. Diniz and F. Canduri, "Bioinformatics: an overview and its applications," *Genet Mol Res*, vol. 16, no. 1, p. 17, 2017.

[207] S. Khan, K. A. Shakil, and M. Alam, "Cloud-based big data analytics—a survey of current research and future directions," *Big Data Analytics: Proceedings of CSI 2015*, pp. 595–604, 2018.

[208] N. Ashtari, A. Bunt, J. McGrenere, M. Nebeling, and P. K. Chilana, "Creating augmented and virtual reality applications: Current practices, challenges, and opportunities," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, pp. 1–13, 2020.

[209] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International journal of web and grid services*, vol. 14, no. 4, pp. 352–375, 2018.

[210] R. Beck, "Beyond bitcoin: The rise of blockchain world," *Computer*, vol. 51, no. 2, pp. 54–58, 2018.

[211] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 79–92, 2021.

[212] M. Katsaragakis, L. Papadopoulos, M. Konijnenburg, F. Catthoor, and D. Soudris, "A memory footprint optimization framework for python applications targeting edge devices," *Journal of Systems Architecture*, vol. 142, p. 102936, 2023.

[213] M. Katsaragakis, L. Papadopoulos, C. Baloukas, and D. Soudris, "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)*, pp. 748–753, 2022.

[214] H. G. Abreha, M. Hayajneh, and M. A. Serhani, "Federated learning in edge computing: a systematic survey," *Sensors*, vol. 22, no. 2, p. 450, 2022.

[215] "Pypl popularity of programming language," Feb. 2022.

[216] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[217] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pp. 256–267, 2017.

[218] J. M. Redondo and F. Ortin, "A comprehensive evaluation of common python implementations," *IEEE Software*, vol. 32, no. 4, pp. 76–84, 2014.

[219] C. P. Lamprakos, L. Papadopoulos, F. Catthoor, and D. Soudris, "The impact of dynamic storage allocation on cpython execution time, memory footprint and energy consumption: An empirical study," in *International Conference on Embedded*

Computer Systems, pp. 219–234, Springer, 2022.
[220] "Micropython." https://micropython.org/, Feb. 2022.
[221] https://www.python.org/, April 2022.
[222] "Finalizers." https://docs.python.org/2/c-api/init.html.c.Py_Finalize, April 2023.
[223] https://docs.python.org/3/c-api/memory.html.
[224] https://realpython.com/python-memory-management/#cpythons-memory-management.
[225] https://rushter.com/blog/python-memory-managment/.
[226] https://github.com/pythonprofilers/memory_profiler.
[227] https://github.com/zhuyifei1999/guppy3/.
[228] https://pypi.org/project/pyflakes/.
[229] https://pypi.org/project/autoflake/.
[230] https://coverage.readthedocs.io/en/v4.5.x/.
[231] https://docs.python.org/3/library/trace.html.
[232] H. Zhai, C. Casalnuovo, and P. Devanbu, "Test coverage in python programs," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 116–120, IEEE, 2019.
[233] S. Vappangi, N. K. Penjarla, S. E. Mathe, and H. K. Kondaveeti, "Applications of raspberry pi in bio-technology: A review," in *2022 2nd International Conference on Artificial Intelligence and Signal Processing (AISP)*, pp. 1–6, IEEE, 2022.
[234] "Pyperformance." https://pyperformance.readthedocs.io/.
[235] https://github.com/OmarAflak/Medium-Python-Neural-Network, April 2023.
[236] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 international joint conference on neural networks (IJCNN)*, pp. 2921–2926, IEEE, 2017.
[237] A. Liaw, M. Wiener, *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.
[238] "Imec." https://www.imec-int.com/en, Feb. 2022.
[239] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, *et al.*, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, 2017.
[240] M. Katsaragakis, D. Masouros, L. Papadopoulos, F. Catthoor, and D. Soudris, "On the implications of heterogeneous memory tiering on spark in-memory analytics," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 945–952, IEEE, 2023.
[241] R. S. Venkatesh, T. Mason, P. Fernando, G. Eisenhauer, and A. Gavrilovska, "Scheduling hpc workflows with intel optane persistent memory," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 56–65, IEEE, 2021.
[242] S. D. Smart, T. Quintino, and B. Raoult, "A high-performance distributed object-store for exascale numerical weather prediction and climate," in *Proceedings of the*

Platform for Advanced Scientific Computing Conference, pp. 1–11, 2019.

[243] J. Ejarque, R. M. Badia, L. Albertin, G. Aloisio, E. Baglione, Y. Becerra, S. Boschert, J. R. Berlin, A. D'Anca, D. Elia, *et al.*, "Enabling dynamic and intelligent workflows for hpc, data analytics, and ai convergence," *Future Generation Computer Systems*, vol. 134, pp. 414–429, 2022.

[244] T. Ungerer and P. Carpenter, "Eurolab-4-hpc long-term vision on high-performance computing," *arXiv preprint arXiv:1807.04521*, 2018.

[245] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 302–313, 2013.

[246] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, p. 786–797, feb 2015.

[247] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in Byte-Addressable persistent B+-Tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, (Oakland, CA), pp. 187–200, USENIX Association, Feb. 2018.

[248] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, (New York, NY, USA), p. 183–196, Association for Computing Machinery, 2012.

[249] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pp. 187–200, 2018.

[250] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[251] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 184–198, 2020.

[252] "IPCMTL." https://docs.pmem.io/ipmctl-user-guide/. Accessed: 15-10-2023.

[253] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, "Emerging nvm: A survey on architectural integration and research challenges," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 2, pp. 1–32, 2017.

[254] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pp. 409–439, 2018.

[255] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[256] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[257] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 729–730, 2012.

[258] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[259] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

[260] S. S. Gill and R. Buyya, "A taxonomy and future directions for sustainable cloud computing: 360 degree view," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–33, 2018.

[261] S. Shiratake, "Scaling and performance challenges of future dram," in *2020 IEEE International Memory Workshop (IMW)*, pp. 1–3, IEEE, 2020.

[262] F. Li, "Cloud-native database systems at alibaba: Opportunities and challenges," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2263–2272, 2019.

[263] "Google Cloud and Intel Optane DCPM." https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory. Accessed: 15-10-2023.

[264] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, "Large-scale in-memory analytics on intel® optane™ dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pp. 1–8, 2020.

[265] L. Pentecost, A. Hankin, M. Donato, M. Hempstead, G.-Y. Wei, and D. Brooks, "Nvmexplorer: A framework for cross-stack comparisons of embedded non-volatile memories," *arXiv preprint arXiv:2109.01188*, 2021.

[266] H. Kellerer, U. Pferschy, D. Pisinger, H. Kellerer, U. Pferschy, and D. Pisinger, *Multidimensional knapsack problems*. Springer, 2004.

[267] E. L. Alvanaki, M. Katsaragakis, D. Masouros, S. Xydis, and D. Soudris, "Decoupled access-execute enabled dvfs for tinyml deployments on stm32 microcontrollers," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2024.

[268] S. Kargar and F. Nawab, "Challenges and future directions for energy, latency, and lifetime improvements in nvms," *Distributed and Parallel Databases*, vol. 41, no. 3, pp. 163–189, 2023.

[269] D. D. Sharma, "Compute express link®: An open industry-standard interconnect

enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 5–12, IEEE, 2022.

[270] "TPC-C." https://www.tpc.org/tpcc/.
Accessed: 15-10-2023.

[271] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1567–1581, 2016.

[272] "Memory latency checker." https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[273] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," *Noise reduction in speech processing*, pp. 1–4, 2009.

[274] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černockỳ, "Strategies for training large scale neural network language models," in *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*, pp. 196–201, IEEE, 2011.

[275] A. Nanda, S. Gupta, and M. Vijrania, "A comprehensive survey of olap: recent trends," in *2019 3rd International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 425–430, IEEE, 2019.

[276] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

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

[278] K. Kim, H. Kim, J. So, W. Lee, J. Im, S. Park, J. Cho, and H. Song, "Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander," *IEEE Micro*, vol. 43, no. 2, pp. 20–29, 2023.

[279] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid dram/nvm main memory," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pp. 1–8, 2015.

[280] D. Bittman, M. Gray, J. Raizes, S. Mukhopadhyay, M. Bryson, P. Alvaro, D. D. Long, and E. L. Miller, "Designing data structures to minimize bit flips on nvm," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 85–90, IEEE, 2018.

[281] C. Baloukas, J. L. Risco-Martin, D. Atienza, C. Poucet, L. Papadopoulos, S. Mamagkakis, D. Soudris, J. I. Hidalgo, F. Catthoor, and J. Lanchares, "Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems," *Journal of Systems and Software*, vol. 82, no. 4, pp. 590–602, 2009.

[282] "Hp quartz emulator." https://github.com/HewlettPackard/quartz.

[283] C. Igel, V. Heidrich-Meisner, and T. Glasmachers, "Shark," *Journal of Machine Learning Research*, vol. 9, pp. 993–996, 2008.

[284] "Chrono physics engine." https://projectchrono.org/.

[285] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[286] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, Ieee, 2010.

[287] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[288] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for multi-tiered memory systems," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (I. Calciu and G. Kuenning, eds.), pp. 715–728, USENIX Association, 2021.

[289] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, eds.), pp. 331–345, ACM, 2019.

[290] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real NVM," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (R. van Renesse and N. Zeldovich, eds.), pp. 392–407, ACM, 2021.

[291] C. Pinto, D. Syrivelis, M. Gazzetti, P. K. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pp. 868–880, IEEE, 2020.

[292] K. Katrinis, D. Syrivelis, D. N. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. López-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "Rack-scale disaggregated cloud data centers: The dredbox project vision," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016* (L. Fanucci and J. Teich, eds.), pp. 690–695, IEEE, 2016.

[293] A. Roozbeh, J. M. Soares, G. Q. M. Jr., F. Wuhib, C. Padala, M. Mahloo, D. Turull, V. Yadhav, and D. Kostic, "Software-defined "hardware" infrastructures: A survey

on enabling technologies and open research directions," *IEEE Commun. Surv. Tutorials*, vol. 20, no. 3, pp. 2454–2485, 2018.

[294] A. Belmonte, H. Oh, N. Rassoul, G. Donadio, J. Mitard, H. Dekkers, R. Delhougne, S. Subhechha, A. Chasin, M. Van Setten, *et al.*, "Capacitor-less, long-retention (> 400s) dram cell paving the way towards low-power and high-density monolithic 3d dram," in *2020 IEEE International Electron Devices Meeting (IEDM)*, pp. 28–2, IEEE, 2020.

[295] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, *et al.*, "Sap hana adoption of non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1754–1765, 2017.

[296] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li, "Adaptive code learning for spark configuration tuning," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 1995–2007, IEEE, 2022.

[297] F. V. Zacarias, R. Nishtala, and P. Carpenter, "Contention-aware application performance prediction for disaggregated memory systems," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pp. 49–59, 2020.

[298] K. T. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pp. 189–200, IEEE Computer Society, 2012.

[299] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*, pp. 41–51, IEEE, 2010.

[300] S. Akram, "Performance evaluation of intel optane memory for managed workloads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–26, 2021.

[301] "Intel mba tool."

[302] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 184–198, 2020.

[303] L. Peled, U. Weiser, and Y. Etsion, "A neural network prefetcher for arbitrary memory access patterns," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–27, 2019.

[304] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 126–136, IEEE, 2015.

[305] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asyn-

chronously," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 369–383, 2016.

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

[307] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.

[308] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[309] Y. Zhang, J. Zhan, J. Yang, W. Jiang, L. Li, and Y. Li, "Energy-aware page replacement for nvm based hybrid main memory system," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–6, IEEE, 2017.

[310] R. A. Rayan, I. Zafar, H. Rajab, M. A. M. Zubair, M. Maqbool, and S. Hussain, "Impact of iot in biomedical applications using machine and deep learning," *Machine Learning Algorithms for Signal and Image Processing*, pp. 339–360, 2022.

[311] S. Zhang, M. Vijayaraghavan, *et al.*, "Weak memory models: Balancing definitional simplicity and implementation flexibility," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 288–302, IEEE, 2017.

[312] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[313] "Structural simulation toolkit." https://sst-simulator.org/.

[314] T. Estro, P. Bhandari, A. Wildani, and E. Zadok, "Desperately seeking... optimal {Multi-Tier} cache configurations," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[315] "Zephyr real-time operating system." https://zephyrproject.org/.

[316] "Compute express link." https://computeexpresslink.org/.

[317] "4th gen amd epyc™ processor architecture." https://www.amd.com/en/campaigns/epyc-9004-architecture.

[318] "Samsung eletronics. accessed in 2023. scalable memory development kit v1.3.." https://github.com/OpenMPDK/SMDK.

[319] "Intel launches 4th gen xeon scalable processors, max series cpus." https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-xeon-scalable-processors-max-series-cpus-gpus.html#gs.5culhf.

[320] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 105–121, 2023.

[321] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in science & engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[322] "New memory architectures poised to reshape the future of data centers." https://www.nextplatform.com/microsite-landing/new-memory-architectures-poised-to-reshape-the-future-of-data-centers/.

[323] H. Al Maruf and M. Chowdhury, "Memory disaggregation: advances and open challenges," *ACM SIGOPS Operating Systems Review*, vol. 57, no. 1, pp. 29–37, 2023.

[324] J. Wang, "Using persistent memory to reduce cloud servers costs," 2021.

[325] A. Puri, J. Jose, and T. Venkatesh, "Design, modeling, and analysis of memory allocation policies for rack-scale memory disaggregation," 2023.

[326] M. Skublewska-Paszkowska, M. Milosz, P. Powroznik, and E. Lukasik, "3d technologies for intangible cultural heritage preservation—literature review for selected databases," *Heritage Science*, vol. 10, no. 1, p. 3, 2022.

[327] S. I. Shim, J. Jang, and J. Song, "Trends and future challenges of 3d nand flash memory," in *2023 IEEE International Memory Workshop (IMW)*, pp. 1–4, IEEE, 2023.

[328] G. Shan, Y. Zheng, C. Xing, D. Chen, G. Li, and Y. Yang, "Architecture of computing system based on chiplet," *Micromachines*, vol. 13, no. 2, p. 205, 2022.

[329] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[330] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "A review of near-memory computing architectures: Opportunities and challenges," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 608–617, IEEE, 2018.

[331] D. Fujiki, X. Wang, A. Subramaniyan, and R. Das, *In-/near-memory Computing*. Springer, 2021.

[332] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[333] P. Mannocci, M. Farronato, N. Lepri, L. Cattaneo, A. Glukhov, Z. Sun, and D. Ielmini, "In-memory computing with emerging memory devices: Status and outlook," *APL Machine Learning*, vol. 1, no. 1, 2023.

[334] H. Ning, Z. Yu, Q. Zhang, H. Wen, B. Gao, Y. Mao, Y. Li, Y. Zhou, Y. Zhou, J. Chen, *et al.*, "An in-memory computing architecture based on a duplex two-dimensional material structure for in situ machine learning," *Nature nanotechnology*, vol. 18, no. 5, pp. 493–500, 2023.

[335] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems*

Design and Implementation (NSDI 17), pp. 649–667, 2017.

[336] M. K. Aguilera, E. Amaro, N. Amit, E. Hunhoff, A. Yelam, and G. Zellweger, "Memory disaggregation: why now and what are the challenges," *ACM SIGOPS Operating Systems Review*, vol. 57, no. 1, pp. 38–46, 2023.

[337] W. Chen, S. Tappertzhofen, H. J. Barnaby, and M. N. Kozicki, "Conductive-bridging access memory," *Resistive Switching: Oxide Materials, Mechanisms, Devices and Operations*, p. 147, 2021.

[338] T. Alexoudi, G. T. Kanellos, and N. Pleros, "Optical ram and integrated optical memories: a survey," *Light: Science & Applications*, vol. 9, no. 1, p. 91, 2020.

[339] H. Mekawey, M. Elsayed, Y. Ismail, and M. A. Swillam, "Optical interconnects finally seeing the light in silicon photonics: Past the hype," *Nanomaterials*, vol. 12, no. 3, p. 485, 2022.

[340] C. Xie and B. Zhang, "Scaling optical interconnects for hyperscale data center networks," *Proceedings of the IEEE*, vol. 110, no. 11, pp. 1699–1713, 2022.

[341] Y. Su, B. Wu, S. Chen, J.-H. Sun, Y.-J. Yu, M.-P. Zhuo, Z.-S. Wang, and X.-D. Wang, "Organic branched heterostructures with optical interconnects for photonic barcodes," *Angewandte Chemie International Edition*, vol. 61, no. 22, p. e202117857, 2022.

[342] Y. Feng, H. Yan, F. Ding, T. Bai, Y. Nie, Y. Zhao, W. Feng, and B. Z. Tang, "Multiring-induced multicolour emission: hyperbranched polysiloxane with silicon bridge for data encryption," *Materials Chemistry Frontiers*, vol. 4, no. 5, pp. 1375–1382, 2020.

[343] M. Katsaragakis, D. Masouros, V. Tsoutsouras, F. Samie, L. Bauer, J. Henkel, and D. Soudris, "Dmrm: Distributed market-based resource management of edge computing systems," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1391–1396, IEEE, 2019.

[344] F. Samie, L. Bauer, and J. Henkel, "IoT technologies for embedded computing: A survey," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2016.

[345] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, 2016.

[346] V. Stantchev, A. Barnawi, S. Ghulam, J. Schubert, and G. Tamm, "Smart items, fog and cloud computing as enablers of servitization in healthcare," *Sensors & Transducers*, vol. 185, no. 2, p. 121, 2015.

[347] F. Samie, V. Tsoutsouras, S. Xydis, L. Bauer, D. Soudris, and J. Henkel, "Distributed QoS management for internet of things under resource constraints," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2016.

[348] K. Cao, G. Xu, J. Zhou, T. Wei, M. Chen, and S. Hu, "QoS-adaptive approximate real-time computation for mobility-aware IoT lifetime optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[349] L. Liu, Z. Chang, X. Guo, S. Mao, and T. Ristaniemi, "Multiobjective optimization for computation offloading in fog computing," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 283–294, 2018.

[350] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 161–176, 2014.

[351] T. Ebi, D. Kramer, W. Karl, and J. Henkel, "Economic learning for thermal-aware power budgeting in many-core architectures," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2011.

[352] H. Zhang, F. Guo, H. Ji, and C. Zhu, "Combinational auction-based service provider selection in mobile edge computing networks," *IEEE Access*, vol. 5, pp. 13455–13464, 2017.

[353] D. Chatzopoulos, M. Ahmadi, S. Kosta, and P. Hui, "Have you asked your neighbors? A hidden market approach for device-to-device offloading," in *International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2016.

[354] T. Melissaris, I. Anagnostopoulos, D. Soudris, and D. Reisis, "Agora: Agent and market-based resource management for many-core systems," in *International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 400–403, 2016.

[355] N. C. Luong, D. T. Hoang, P. Wang, D. Niyato, D. I. Kim, and Z. Han, "Data collection and wireless communication in internet of things (IoT) using economic analysis and pricing models: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, 2016.

[356] S. Paris, F. Martignon, I. Filippini, and L. Chen, "An efficient auction-based mechanism for mobile data offloading," *IEEE Transactions on Mobile Computing*, vol. 14, no. 8, pp. 1573–1586, 2015.

[357] F. M. F. Wong, C. Joe-Wong, S. Ha, Z. Liu, and M. Chiang, "Improving user QoE for residential broadband: Adaptive traffic management at the network edge," in *International Symposium on Quality of Service (IWQoS)*, pp. 105–114, 2015.

[358] D. E. Jeffcoat and R. L. Bulfin, "Simulated annealing for resource-constrained scheduling," *European Journal of Operational Research*, vol. 70, no. 1, pp. 43–51, 1993.

[359] M. H. Kashani and M. Jahanshahi, "Using simulated annealing for task scheduling in distributed systems," in *Intl. Conf. On Computational Intelligence, Modelling and Simulation*, pp. 265–269, 2009.

[360] H. Orsila, T. Kangas, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, "Automated memory-aware application distribution for multi-processor system-on-chips," *Journal of Systems Architecture*, vol. 53, no. 11, pp. 795–815, 2007.

[361] A. Karteris, M. Katsaragakis, D. Masouros, and D. Soudris, "Sgrm: stackelberg game-based resource management for edge computing systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1203–1208,

IEEE, 2022.

[362] A. Chowdhury and S. A. Raut, "A survey study on internet of things resource management," *Journal of Network and Computer Applications*, vol. 120, pp. 42–60, 2018.

[363] G. Tanganelli, C. Vallati, and E. Mingozzi, *Ensuring Quality of Service in the Internet of Things*, pp. 139–163.
06 2018.

[364] S. Li, X. Hu, and Y. Du, "Deep reinforcement learning and game theory for computation offloading in dynamic edge computing markets," *IEEE Access*, 2021.

[365] Y. Chen, Z. Li, B. Yang, K. Nai, and K. Li, "A Stackelberg game approach to multiple resources allocation and pricing in mobile edge computing," *Future Generation Computer Systems*, vol. 108, pp. 273–287, 2020.

[366] H. Zhang, Y. Xiao, S. Bu, D. Niyato, F. R. Yu, and Z. Han, "Computing resource allocation in three-tier IoT fog networks: A joint optimization approach combining Stackelberg game and matching," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1204–1215, 2017.

[367] Z. Han, D. Niyato, W. Saad, T. Başar, and A. Hjørungnes, *Game theory in wireless and communication networks: theory, models, and applications.*
Cambridge university press, 2012.

[368] L. M. Ausubel and P. Milgrom, "The Lovely but Lonely Vickrey Auction," in *Combinatorial Auctions*, ch. 1, pp. 17–40, MIT Press, 2006.

[369] D. Y. Zhang and D. Wang, "An integrated top-down and bottom-up task allocation approach in social sensing based edge computing systems," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 766–774, IEEE, 2019.

[370] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep Speech: Scaling up end-to-end speech recognition," 2014.

[371] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015.

[372] D. Neven, B. D. Brabandere, S. Georgoulis, M. Proesmans, and L. V. Gool, "Towards End-to-End Lane Detection: an Instance Segmentation Approach," 2018.

[373] E. Choi, M. T. Bahadori, J. Sun, J. Kulas, A. Schuetz, and W. Stewart, "RETAIN: An Interpretable Predictive Model for Healthcare using Reverse Time Attention Mechanism," in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[374] A. K. Kakolyris, M. Katsaragakis, D. Masouros, and D. Soudris, "Road-runner: Collaborative dnn partitioning and offloading on heterogeneous edge systems," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023.

[375] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[376] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2018.

[377] Y. Liang, D. O'Keeffe, and N. Sastry, "Paige: Towards a hybrid-edge design for privacy-preserving intelligent personal assistants," in *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, pp. 55–60, 2020.

[378] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[379] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[380] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[381] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[382] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[383] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[384] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 401–411, 2018.

[385] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[386] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–37, 2020.

[387] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th annual international conference on mobile computing and*

networking, pp. 1–15, 2020.

[388] L. Zhou, H. Wen, R. Teodorescu, and D. H. Du, "Distributing deep neural networks with containerized partitions at the edge," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[389] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward collaborative inferencing of deep neural networks on internet-of-things devices," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.

[390] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[391] E.-I. Christoforidis, S. Xydis, and D. Soudris, "Cf-tune: Collaborative filtering auto-tuning for energy efficient many-core processors," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 25–28, 2017.

[392] "Edge computing: gaining the digital edge." https://atos.net/en/solutions/edge-computing-infrastructure.
Accessed: 10-09-2022.

[393] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.

[394] E. Arnold, O. Y. Al-Jarrah, M. Dianati, S. Fallah, D. Oxtoby, and A. Mouzakitis, "A survey on 3d object detection methods for autonomous driving applications," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 10, pp. 3782–3795, 2019.

[395] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[396] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pp. 31–36, 2018.

[397] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[398] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[399] "Amazon sagemaker edge." https://aws.amazon.com/sagemaker/edge/.

[400] "Google edge tpu." https://cloud.google.com/edge-tpu.

[401] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," *IEEE Transactions on Computers*, 2021.

[402] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Mae-

stro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE micro*, 2020.

[403] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," *Proceedings of Machine Learning and Systems*, 2021.

[404] K. T. Chitty-Venkata and A. K. Somani, "Neural architecture search survey: A hardware perspective," *ACM Computing Surveys*, 2022.

[405] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[406] "Microsoft nni." https://github.com/microsoft/nni.

[407] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[408] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[409] J. D. De Leon and R. Atienza, "Depth pruning with auxiliary networks for tinyml," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2022.

[410] J. Moosmann, H. Mueller, N. Zimmerman, G. Rutishauser, L. Benini, and M. Magno, "Flexible and fully quantized ultra-lightweight tinyissimoyolo for ultra-low-power edge systems," *arXiv preprint:2307.05999*, 2023.

[411] O. Spantidi and I. Anagnostopoulos, "Automated energy-efficient dnn compression under fine-grain accuracy constraints," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[412] O. Saha, A. Kusupati, H. V. Simhadri, M. Varma, and P. Jain, "Rnnpool: Efficient non-linear pooling for ram constrained inference," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20473–20484, 2020.

[413] E. Liberis, Ł. Dudziak, and N. D. Lane, "μnas: Constrained neural architecture search for microcontrollers," in *Proceedings of the 1st Workshop on Machine Learning and Systems*, pp. 70–79, 2021.

[414] S. Jaiswal, R. K. K. Goli, A. Kumar, V. Seshadri, and R. Sharma, "Minun: Accurate ml inference on microcontrollers," in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 26–39, 2023.

[415] D. C. Snowdon12, E. Le Sueur, S. M. Petters, and G. Heiser, "A platform for os-level power management," *The European Professional Society on Computer Systems 2009*, 2009.

[416] B. Acun, K. Chandrasekar, and L. V. Kale, "Fine-grained energy efficiency using

per-core dvfs with an adaptive runtime system," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8, IEEE, 2019.

[417] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Proceedings of the 41st annual Design Automation Conference*, pp. 275–280, 2004.

[418] W. Gay, "Beginning stm32," *Beginning STM32*, 2018.

[419] H. Kellerer, U. Pferschy, D. Pisinger, H. Kellerer, U. Pferschy, and D. Pisinger, "The multiple-choice knapsack problem," *Knapsack Problems*, pp. 317–347, 2004.

[420] A. Tzenetopoulos, E. Apostolakis, A. Tzomaka, C. Papakostopoulos, K. Stavrakakis, M. Katsaragakis, I. Oroutzoglou, D. Masouros, S. Xydis, and D. Soudris, "Faas and curious: Performance implications of serverless functions on edge computing platforms," in *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*, pp. 428–438, Springer, 2021.

[421] A. Ferikoglou, P. Chrysomeris, A. Tzenetopoulos, M. Katsaragakis, D. Masouros, and D. Soudris, "Iris: Interference and resource aware predictive orchestration for ml inference serving," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pp. 1–12, IEEE, 2023.