Efficient State Management with Persistent Memory

Lawrence Benson

Universitätsdissertation
zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)

in der Wissenschaftsdisziplin
IT-Systems Engineering

eingereicht an der
Digital-Engineering-Fakultät
der Universität Potsdam

Datum der Disputation: 09.11.2023

Betreuer

Prof. Dr. Tilmann Rabl
Hasso Plattner Institut, Universität Potsdam

Gutachter

Prof. Tianzheng Wang, PhD
Simon Fraser University

Prof. Pinar Tözün, PhD
IT University of Copenhagen

Abstract

Efficiently managing large state is a key challenge for data management systems. Traditionally, state is split into fast but volatile state in memory for processing and persistent but slow state on secondary storage for durability. Persistent memory (PMem), as a new technology in the storage hierarchy, blurs the lines between these states by offering both byte-addressability and low latency like DRAM as well persistence like secondary storage. These characteristics have the potential to cause a major performance shift in database systems.

Driven by the potential impact that PMem has on data management systems, in this thesis we explore their use of PMem. We first evaluate the performance of real PMem hardware in the form of Intel Optane in a wide range of setups. To this end, we propose PerMA-Bench, a configurable benchmark framework that allows users to evaluate the performance of customizable database-related PMem access. Based on experimental results obtained with PerMA-Bench, we discuss findings and identify general and implementation-specific aspects that influence PMem performance and should be considered in future work to improve PMem-aware designs. We then propose Viper, a hybrid PMem-DRAM key-value store. Based on PMem-aware access patterns, we show how to leverage PMem and DRAM efficiently to design a key database component. Our evaluation shows that Viper outperforms existing key-value stores by 4–18× for inserts while offering full data persistence and achieving similar or better lookup performance. Next, we show which changes must be made to integrate PMem components into larger systems. By the example of stream processing engines, we highlight limitations of current designs and propose a prototype engine that overcomes these limitations. This allows our prototype to fully leverage PMem's performance for its internal state management. Finally, in light of Optane's discontinuation, we discuss how insights from PMem research can be transferred to future multi-tier memory setups by the example of Compute Express Link (CXL).

Overall, we show that PMem offers high performance for state management, bridging the gap between fast but volatile DRAM and persistent but slow secondary storage. Although Optane was discontinued, new memory technologies are continuously emerging in various forms and we outline how novel designs for them can build on insights from existing PMem research.

Zusammenfassung

Die effiziente Verwaltung großer Zustände ist eine zentrale Herausforderung für Datenverwaltungssysteme. Traditionell wird der Zustand in einen schnellen, aber flüchtigen Zustand im Speicher für die Verarbeitung und einen persistenten, aber langsamen Zustand im Sekundärspeicher für die Speicherung unterteilt. Persistenter Speicher (PMem), eine neue Technologie in der Speicherhierarchie, lässt die Grenzen zwischen diesen Zuständen verschwimmen, indem er sowohl Byte-Adressierbarkeit und geringe Latenz wie DRAM als auch Persistenz wie Sekundärspeicher bietet. Diese Eigenschaften haben das Potenzial, die Leistung von Datenbanksystemen grundlegend zu verändern.

Aufgrund der potenziellen Auswirkungen, die PMem auf Datenverwaltungssysteme hat, untersuchen wir in dieser Arbeit ihre Verwendung von PMem. Zunächst evaluieren wir die Leistung von echter PMem-Hardware in Form von Intel Optane in einer Vielzahl von Konfigurationen. Zu diesem Zweck stellen wir PerMA-Bench vor, ein konfigurierbares Benchmark-Framework, mit dem Benutzer die Leistung von anpassbaren datenbankbezogenen PMem-Zugriffen untersuchen können. Auf der Grundlage der mit PerMA-Bench erzielten experimentellen Ergebnisse diskutieren wir unsere Erkenntnisse und identifizieren allgemeine und implementierungsspezifische Aspekte, die die PMem-Leistung beeinflussen und in zukünftigen Arbeiten berücksichtigt werden sollten, um PMem-fähige Designs zu verbessern. Anschließend präsentieren wir Viper, einen hybriden PMem-DRAM Key-Value-Store. Basierend auf PMem-bewussten Zugriffsmustern zeigen wir, wie PMem und DRAM effizient genutzt werden können, um eine wichtige Datenbankkomponente zu entwickeln. Unsere Evaluierung zeigt, dass Viper bestehende Key-Value-Stores bei Einfügungen um 4- bis 18-mal übertrifft, während er gleichzeitig vollständige Datenpersistenz bietet und ähnliche oder bessere Lookup-Leistung erzielt. Als nächstes zeigen wir, welche Änderungen vorgenommen werden müssen, um PMem-Komponenten in größere Systeme zu integrieren. Am Beispiel von Datenstromverarbeitungssystemen zeigen wir die Einschränkungen aktueller Designs auf und stellen einen Prototyp eines Systems vor, das diese Einschränkungen überwindet. Dadurch kann unser Prototyp die Leistung von PMem für die interne Zustandsverwaltung voll ausnutzen. Schließlich erörtern wir angesichts der Abkündigung von Optane, wie Erkenntnisse aus der PMem-Forschung am Beispiel von Compute Express Link (CXL) auf künftige mehrstufige Speicher-Setups übertragen werden können.

Insgesamt zeigen wir, dass PMem eine hohe Leistungsfähigkeit für die Zustandsverwaltung bietet und die Lücke zwischen schnellem, aber flüchtigem DRAM und beständigem, aber langsamem Sekundärspeicher schließt. Obwohl Optane eingestellt wurde, entstehen ständig neue Speichertechnologien in verschiedenen Formen, und wir skizzieren, wie neuartige Entwürfe für sie auf den Erkenntnissen aus der bestehenden PMem-Forschung aufbauen können.

Contents

Abstract iii

Zusammenfassung v

Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Contribution . . . 5
- 1.3 Technical Contributions and Impact . . . 6
- 1.4 Additional Contributions . . . 8
- 1.5 Thesis Outline . . . 9

2 Background 11
- 2.1 Persistent Memory . . . 11
  - 2.1.1 Types . . . 11
  - 2.1.2 Intel Optane . . . 12
  - 2.1.3 Accessing Persistent Memory . . . 15
  - 2.1.4 Atomicity and Durability . . . 18
  - 2.1.5 Programming Interfaces and APIs . . . 20

3 Benchmarking Persistent Memory Access 23
- 3.1 Introduction . . . 23
- 3.2 Introducing PerMA-Bench . . . 25
  - 3.2.1 Runtime . . . 25
  - 3.2.2 Custom Workloads and Configuration . . . 27
  - 3.2.3 Persist Instructions . . . 29
- 3.3 PerMA-Bench Results . . . 29
  - 3.3.1 Setup And Methodology . . . 30
  - 3.3.2 Raw Performance Workloads – Bandwidth . . . 31

3.3.3 Raw Performance Workloads - Latency 35
3.3.4 Database-Related Workloads 38
3.3.5 Single Server Performance 43
3.4 Server Price-Performance 47
3.5 Discussion 50
3.6 Related Work 51
3.7 Conclusion 52

4 Viper: An Efficient Hybrid PMem-DRAM Key-Value Store 53
4.1 Introduction 53
4.2 Background 55
4.3 Viper: A Hybrid Key-Value Store 55
4.3.1 Hybrid Design 56
4.3.2 Architecture 59
4.4 Key-Value Store Operations 62
4.4.1 Viper Client 63
4.4.2 Put 64
4.4.3 Get 66
4.4.4 Update 67
4.4.5 Delete 67
4.4.6 Space Reclamation 68
4.4.7 Recovery 68
4.5 Evaluation 69
4.5.1 Setup and Methodology 69
4.5.2 Other Systems 69
4.5.3 Micro Benchmarks 70
4.5.4 YCSB 79
4.6 Related Work 80
4.7 Conclusion 82

5 Darwin: Scale-In Stream Processing 83
5.1 Introduction 83
5.2 Background 85
5.3 Current SPE Challenges 86
5.3.1 Focus of Existing Systems 86
5.3.2 State Management 88
5.3.3 Resource Inefficiency 88
5.3.4 Overall System Optimization 88

5.4 Scale-In Stream Processing . . . 89
5.4.1 Opportunities for State Management . . . 90
5.4.2 Opportunities for Resource Inefficiency . . . 93
5.4.3 Opportunities for System Optimization . . . 94
5.5 Introducing Darwin . . . 95
5.5.1 Darwin Architecture . . . 95
5.5.2 Performance . . . 97
5.6 Conclusion . . . 98

6 What We Can Learn from Persistent Memory for CXL **99**
6.1 Introduction . . . 99
6.2 Compute Express Link . . . 100
6.3 Transferring Insights from PMem to CXL-Attached Memory . . . 100
6.4 Conclusion . . . 104

7 Conclusion & Outlook **107**
7.1 Conclusion . . . 107
7.2 Research Outlook . . . 108

References **111**

List of Figures

1.1 PMem sits between SSDs and DRAM in the storage hierarchy. . . 2

2.1 Six interleaved Optane DIMMs span a continuous 24 KiB region. . 12
2.2 Standard PMem access modes. 16
2.3 Writing to NVDIMM-Ps from the CPU. 17

3.1 Execution cycle of a benchmark in PerMA-Bench. 26
3.2 Sequential and random read bandwidth 31
3.3 Thread and access size impact on sequential writes. 33
3.4 Impact of persist instruction on write bandwidth. 34
3.5 256 Byte random read + write latency. 36
3.6 Double-flush latency. 37
3.7 PMem index workloads compared to DRAM. 39
3.8 Hash index in PerMA and Dash. 40
3.9 Tree index in PerMA and FAST+FAIR. 41
3.10 Impact of eADR on write performance. 43
3.11 Impact of prefetcher on random read bandwidth. 44
3.12 Performance-impact of varying memory bus speeds. 45
3.13 Impact of number of DIMMs in the server. 46

4.1 Write latency for various write patterns to DRAM and PMem. . . 57
4.2 Viper's storage aligned with 4 KB PMem layout. 58
4.3 Viper's architecture. 59
4.4 VPage layout with example entries. 61
4.5 Client requesting new VBlock. 63
4.6 Performance of core KVS operations. 71
4.7 Key-value size impact. 73
4.8 Variable-sized ~216 Byte records. 74
4.9 Total memory. 75
4.10 Update strategy. 75
4.11 Viper versions. 76
4.12 Operation breakdown. 77
4.13 Space reclamation. 78
4.14 YCSB latency and throughput. 79

5.1 Insert and get performance. 90
5.2 Checkpoint and recovery duration. 91
5.3 Grouped state access performance. 92
5.4 Darwin’s architecture and execution flow. 95
5.5 Throughput of Darwin, Grizzly, and Flink. 98

6.1 Performance of PerMA and actual index implementations. 101
6.2 Impact of prefetcher on PMem random read bandwidth. 102

List of Tables

2.1 Maximum per-DIMM Optane performance per generation. 14

3.1 Evaluated servers. 30
3.2 PMem price-performance comparison. 48

5.1 Feature set of existing SPEs. 87

6.1 Price-performance of PMem and DRAM. 103

1 Introduction

1.1 Motivation

For decades, data management systems were designed with a divide between data in main memory and data on secondary storage. While storage devices, such as HDDs and SSDs, have significantly lower access performance than DRAM, they are necessary to provide durability. Thus, systems that offer data persistence face a performance drop when durably writing data to storage and when retrieving it back for processing. In addition to providing durability, storage devices are significantly cheaper per GB than memory and offer much higher capacity. For economical reasons, systems often choose to reduce the amount of required DRAM and limit its use to processing and intermediate results, while the majority of the actual data is stored on HDDs or SSDs. However, limited DRAM may entail access to secondary storage for large intermediate results or even for auxiliary structures such as indexes.

To mitigate the storage access performance drop, various research areas focus on designing efficient storage solutions for data management systems. Such research areas are buffer management [34, 119], persistent indexes [11], and storage engines [35, 38]. For the majority of these, key design goals are to reduce storage access in the first place and to leverage efficient access patterns to storage whenever possible. Due to the high importance of these systems, they are continuously adapted to new hardware characteristics [89, 92].

After decades of established DRAM and storage database components, the announcement and arrival of large-scale persistent memory (PMem) technology in the form of Intel Optane [69] led to a rethinking of how persistence is achieved in data management systems. With byte-addressability and performance like DRAM as well as persistence like secondary storage, PMem as a new technology has the potential to cause a major performance shift in database storage and systems. In traditional designs, byte-addressable data is used for fast random access but is generally considered volatile, while persistent data layouts are optimized for slower storage, i.e., page-sized and/or sequential access. PMem blurs these lines, as it achieves fast random access on persistent data, removing the need for a clear distinction between memory and storage.

In Figure 1.1, we show how PMem fits between DRAM and SSDs in the classical

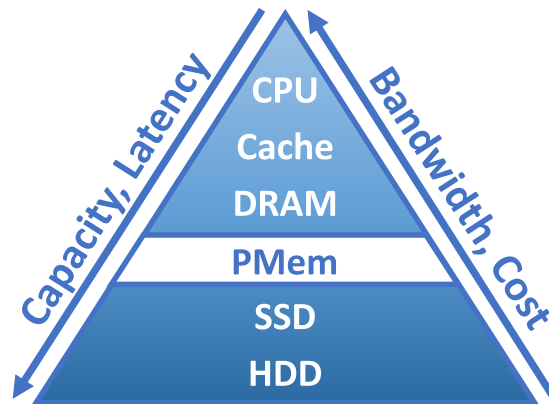

Figure 1.1: PMem sits between SSDs and DRAM in the storage hierarchy.

storage hierarchy. We present this based on four factors, *capacity*, *cost (GB/$)*, *latency*, and *bandwidth*. Due to its characteristics, PMem fits between SSDs and DRAM, both in anticipated performance before the commercial availability of Optane [123, 129] and with the actual performance of Optane [147].

With capacities of 128–512 GB per DIMM, Optane has a higher capacity than DRAM, which is currently available in ~ 8–64 GB per DIMM. SSDs, on the other hand, are available with multiple TB per device. Based on these capacities, SSDs offer more GB/$ than PMem [19], which in turn offers more GB/$ than DRAM [15]. Modern SSDs are in the order of tens of cents per GB, Optane is in the order of dollars per GB, and DRAM ranges from dollars to low tens of dollars per GB.

With random access latency in the order of hundreds of nanoseconds, PMem's latency is very close to DRAM's [147], compared to tens or hundreds of microseconds latency to SSD [49]. The bandwidth of a single PMem DIMM is currently similar to that of modern NVMe SSDs, i.e., in the low GB/s [19]. Thus, PMem has more potential for latency-critical applications than for bandwidth-heavy ones.

Even before PMem was publicly available, the promise of these characteristics sparked research into how index structures [58, 123, 148], storage engines [6, 145], and buffer managers [129] can be redesigned for PMem. With the public availability of real PMem in the form of Optane, research in these areas shifted to Optane-specific designs [26, 91, 97, 155]. These systems raised the question of how to integrate PMem into data management systems and they led to an initial understanding of multi-tier memory in these systems. In this thesis, we address many remaining open questions on how to best design data management systems to fully leverage large-scale PMem as a new technology.

As Optane is currently the only instance of large-scale PMem and it was discontinued by Intel in 2022 [2], the technology's future is currently unclear. Two key aspects that influence the use of PMem are the underlying technology and the way this is connected to the CPU. On the technology side, various other PMem designs have been announced in the past decade [10, 48, 86, 100, 127, 154], but none of them are available yet. Intel's choice to discontinue Optane is mainly driven by economic factors [41], so a new technology with different properties might change the economics of PMem, making it more viable. On the connection side, Compute Express Link (CXL) is an emerging interconnect protocol that allows arbitrary memory to be attached to the CPU via PCIe instead of memory channels. With different types of memory on one server, and potentially even alternative PMem technologies, multi-tier memory setups face similar challenges as with a two-tiered DRAM/PMem setup. CXL may also alter the economics of PMem, as it does not require a high-end Intel CPU to use PMem. Regardless of the exact development of PMem, future technology will likely disrupt the classical storage hierarchy similarly to Optane. With each such technology, an open question remains how to integrate it best. While this thesis mainly covers various challenges around the integration of PMem, we also provide insights into challenges and opportunities for future multi-tier memory setups based on our learnings from PMem.

Driven by the promising results of pre-Optane designs and the potential impact that PMem has on database systems, in this thesis we explore the use of PMem in data management systems. We do this along four goals, starting with understanding its performance, over designing database components for PMem and their integration into larger systems, to transferring PMem insights to future research on multi-tier memory. We outline these four motivational challenges in the following.

Challenge 1: Understanding PMem performance. Before PMem became commercially available, research assumed a performance range that PMem may cover, i.e., experiments were often conducted in simulators that supported varying access latency [123, 148]. This led to a wide range of inconsistent performance numbers. With the arrival of Intel Optane, research shifted towards understanding the *"actual"* performance of PMem [47, 138, 147]. Based on these findings, designs specifically targeted towards Optane emerged, fully leveraging its intricacies and performance characteristics [26, 97, 101]. However, with limited hardware availability, different server configurations, and multiple hardware generations, it is not clear how consistently Optane performs in a wider range of setups. Because of this limited insight, it is also unclear whether the Optane-optimized designs generalize or if they are tailored only toward a single server.

Challenge 2: Designing PMem-aware storage systems. Key-value stores

(KVSs) have become a foundation for state management in modern data-intensive applications and systems. Current designs are either based on secondary storage to offer persistence, e.g., in a storage layer [38], or are purely in-memory for fast access, e.g., for caching [128]. Regardless of the intended use, as a fundamental part of modern database stacks, the performance of KVSs is essential to the overall system performance. Due to their comparatively simple interface around *put()* and *get()* calls on individual records, most logic in a KVS is concerned with record access, i.e., storing and retrieving these records. Leveraging PMem and its characteristics has the potential to significantly improve this access and in turn, improve the performance of all systems building on such a PMem-aware KVS. Early work in this space shows that PMem shifts the performance from traditional KVSs based on slow secondary storage towards faster but volatile in-memory KVS, while still offering persistence [26, 91]. However, it is unclear how to best design such a PMem-aware KVS, especially concerning fundamental memory access patterns and the interaction between faster but limited DRAM and slower but ample PMem.

Challenge 3: Integrating PMem components into larger systems. While it is important to research and develop standalone database components for PMem, simply drop-in replacing these into existing systems does not necessarily yield the best results [19, 81]. It is important to understand the limitations of current systems to fully leverage PMem's potential by using it in the right places. In recent years, stream processing engines (SPEs), such as Apache Flink [21] or Spark Streaming [150], have become widely used in industry for low-latency processing of large data volumes. As SPEs commonly have high ingestion rates, they put a lot of pressure on the underlying storage system, making the potential performance gains from PMem-aware storage engines very high. Due to a wide range of SPE designs and goals, it is unclear which design choices must be made to integrate novel PMem storage engines and which performance can be achieved with the bandwidth and low latency that PMem promises.

Challenge 4: Implications of PMem research on multi-tier memory. Besides PMem, other approaches to and designs for multi-tier memory exist. The Compute Express Link (CXL) [30] interconnect is one of these approaches, which offers byte-addressability to and from arbitrary memory and storage via a PCIe-based protocol. After Optane was discontinued by Intel in 2022 in favor of CXL [41], it is yet unclear which insights gained from PMem research can be transferred to future multi-tier memory setups as given with CXL.

1.2 Research Contribution

In this section, we present this thesis' research question and briefly outline our contribution to each of the individual goals on the path to answering this question. Combining all four motivational challenges described above (*Challenge 1–4*), in this thesis we discuss the following research question:

> *With the emergence of a fundamentally different memory technology in the form of persistent memory, how do data management systems need to be designed to leverage its unique properties for efficient state management and how can we extend these insights to future disruptive memory technologies?*

To answer this question, we make the following contributions. Before analyzing PMem performance or designing novel systems, we provide an overview of PMem terminology and concepts in general and specifically for Optane (Chapter 2).

Contribution 1: Understanding PMem performance. In Chapter 3, we evaluate the performance characteristics of Optane PMem to answer the questions outlined in *Challenge 1*. We propose PerMA-Bench, a configurable benchmark framework that allows users to evaluate the bandwidth, latency, and operations per second for customizable database-related PMem access. Based on PerMA-Bench, we perform an extensive evaluation of PMem performance across four different server configurations, containing both first- and second-generation Optane, with additional parameters such as DIMM power budget and number of DIMMs per server. We validate our results with existing systems and show the impact of low-level design choices. We discuss our findings and identify eight general and implementation-specific aspects that influence PMem performance and should be considered in future work to improve PMem-aware designs. The content of Chapter 3 was published in PVLDB 15(11), 2022 [15].

Contribution 2: Designing PMem-aware storage systems. In Chapter 4, we investigate how to design a PMem-aware storage system in the form of a KVS, as motivated in *Challenge 2* above. We propose three PMem-specific access patterns and implement them in a hybrid PMem-DRAM KVS called Viper. We employ a DRAM-based hash index and a PMem-aware storage layout to utilize the random write speed of DRAM and the efficient sequential write performance of PMem. Our evaluation shows that Viper significantly outperforms existing KVSs for core KVS operations while providing full data persistence. The content of Chapter 4 was published in PVLDB 14(9), 2021 [14].

Contribution 3: Integrating PMem components into larger systems. In Chapter 5, we discuss open challenges and design goals for future SPEs that incor-

porate fast PMem storage to overcome these challenges, as outlined in *Challenge 3*. To this end, we present Darwin, a novel SPE prototype that tailors its execution toward the target environment through adaptive storage backends and query compilation. Due to high ingestion rates, a key challenge in SPEs is the management of large state. The efficiency of inserting, retrieving, and checkpointing data heavily impacts the overall performance of SPEs. By integrating a PMem-aware state store, Darwin's adaptive execution leverages PMem's high performance while supporting larger-than-memory state as with traditional storage-based SPE state backends. Our early results show that this combination achieves an order of magnitude speed-up over current scale-out systems and matches processing rates of scale-up systems. The content of Chapter 5 was published at CIDR 2022 [16].

Contribution 4: Implications of PMem research on multi-tier memory. In Chapter 6, we give an outlook on how PMem research can be transferred to future multi-tier memory setups by the example of CXL (outlined in *Challenge 4*). We discuss how limited hardware availability impacts the performance generalization of new designs, how existing CPU components are not adapted towards different access characteristics, how multi-tier memory setups offer different price-performance trade-offs, and when explicit memory fences may still be needed. To support future CXL research in each of these areas, we discuss how our insights apply to CXL and which problems researchers may encounter along the way. The content of Chapter 6 was published at the NoDMC workshop 2023 [17].

1.3 Technical Contributions and Impact

Throughout our research towards this thesis, our contributions (Chapters 3 through Chapter 6) were published.

Journal Papers. Two chapters of this thesis (Chapter 3 and Chapter 4) were published in a top-tier journal.

- *Lawrence Benson*, Leon Papke, Tilmann Rabl. PerMA-Bench: Benchmarking Persistent Memory Access. PVLDB, 15(11): 2463–2476, 2022.
  In this publication, the author performed the conceptual work, as well as the majority of the implementation, analysis, and writing. Leon Papke assisted with the implementation and writing. Tilmann Rabl guided the conceptual work as the supervisor and aided with writing.

- *Lawrence Benson*, Hendrik Makait, Tilmann Rabl. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. PVLDB, 14(9): 1544–1556, 2021.

In this publication, the author performed the conceptual work, as well as the majority of the implementation, analysis, and writing. Hendrik Makait assisted with the implementation. Tilmann Rabl guided the conceptual work as the supervisor and aided with writing.

Conference Papers. Two chapters (Chapter 5 and Chapter 6) were published in top-tier international and national conferences.

» *Lawrence Benson*, Tilmann Rabl. Darwin: Scale-In Stream Processing. In Conference on Innovative Data Systems Research (CIDR), 2022.
In this publication, the author performed the conceptual work, the implementation, analysis, and the majority of the writing. Tilmann Rabl guided the conceptual work as the supervisor and aided with writing.

» *Lawrence Benson*, Marcel Weisgut, Tilmann Rabl. What We Can Learn from Persistent Memory for CXL. In Datenbanksysteme für Business, Technologie und Web (BTW), 2023.
In this publication, the author performed the conceptual work, the analysis, and the majority of the writing. Marcel Weisgut assisted with the conceptual CXL parts as well as with writing. Tilmann Rabl guided the conceptual work as the supervisor and aided with writing.

Open Source Contributions. To allow for reproducibility and comparison, we have made the code of PerMA-Bench and Viper available. We also contributed patches to existing large open-source projects.

» **PerMA-Bench Code:** github.com/hpides/perma-bench
This repository contains the source code and results for our benchmarking framework PerMA-Bench (Chaper 3). Made available under MIT License. The PerMA-Bench code is used as the foundation of an ongoing benchmarking project by other members of the author's research group.

» **Viper Code:** github.com/hpides/viper
This repository contains the source code and results for our PMem-ware key-value store Viper (Chapter 4). Made available under MIT License. Based on our available code, Viper has been used as an evaluation baseline in various papers succeeding our work (e.g. [54, 140, 142]).

» Added a NUMA feature to Intel's Persistent Memory Development Kit (PMDK). Pull Request: github.com/pmem/pmdk/pull/5067

» Submitted multiple patches to the LLVM compiler for more efficient vector instruction selection.
Commits: github.com/llvm/llvm-project/commits?author=lawben&since=2023-01-01

1.4 Additional Contributions

In this section, we list additional publications that were made during the course of this thesis, but which are not part of it.

First Authorship. The author also explored SIMD vectorization in databases leveraging compilers' internal vector representations.

» *Lawrence Benson*, Richard Ebeling, Tilmann Rabl. Evaluating SIMD Compiler-Intrinsics for Database Systems. In Workshop on Accelerating Analytics and Data Management Systems (ADMS), 2023.
Source code: github.com/hpides/autovec-db

Student Supervision. During his PhD at HPI, the author supervised and worked with various students, which led to four publications at top-tier conferences and journals (in chronological order).

» Björn Daase, Lars Jonas Bollmeier, *Lawrence Benson*, Tilmann Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In Proceedings of the International Conference on Management of Data (SIGMOD), 2021.
Source code: github.com/hpides/pmem-olap

» Maximilian Böther, Otto Kißig, *Lawrence Benson*, Tilmann Rabl. Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs. In Proceedings of the International Workshop on Data Management on New Hardware (DaMoN), 2021.
Source code: github.com/hpides/pmem-nvme-dropin

» Tobias Maltenberger, Till Lehmann, *Lawrence Benson*, Tilmann Rabl. Evaluating In-Memory Hash Joins on Persistent Memory. In Proceedings of the International Conference on Extending Database Technology (EDBT), 2022.

» Maximilian Böther, *Lawrence Benson*, Ana Klimovic, Tilmann Rabl. Analyzing Vectorized Hash Tables Across CPU Architectures. PVLDB, 16(11): 2755 - 2768, 2023.
Source code: github.com/hpides/vectorized-hash-tables

Co-Authorship. The author also co-authored the following paper:

» Wang Yue, *Lawrence Benson*, Tilmann Rabl. Desis: Efficient Window Aggregation in Decentralized Networks. In Proceedings of the International Conference on Extending Database Technology (EDBT), 2023.
Source code: github.com/wywclmqf/DESengine

1.5 Thesis Outline

The remainder of this thesis is structured as follows.

In **Chapter** 2, we introduce persistent memory as the key memory technology on which our contributions are based.

In **Chapter** 3, we present PerMA-Bench, a configurable benchmark framework to analyze bandwidth, latency, and operations per second for customizable database-related PMem access. Based on PerMA-Bench, we evaluate synthetic workloads and real systems on various PMem server configurations and perform a price-performance evaluation.

In **Chapter** 4, we introduce Viper, a hybrid PMem-DRAM key-value store designed for Optane. Based on access pattern microbenchmarks, we propose three key design choices for building a hybrid key-value store and show how they are implemented in Viper. We show that our design outperforms existing solutions at the time.

In **Chapter** 5, we discuss current challenges around high-performance stream processing systems. We outline how efficient state management, as a key challenge, can be improved using our PMem-aware state store Viper in our streaming engine prototype Darwin.

In **Chapter** 6, we discuss how our findings and insights from PMem research can be transferred to Compute Express Link. We do this in light of Intel discontinuing the Optane product line in favor of the emerging CXL interconnect.

In **Chapter** 7, we summarize this thesis and provide an outlook on how our work contributes to future research in this area.

2 Background

This chapter is an extended version of content published in [14, 15].

2.1 Persistent Memory

In this chapter, we present the necessary background on persistent memory as the core technology used throughout this thesis. We cover various aspects of PMem in general and related to Intel's Optane PMem product line. We first discuss general PMem classification types (Section 2.1.1), followed by details specific to Intel's Optane product (Section 2.1.2). Then, we discuss how data is transferred between the CPU and PMem in two different API modes (Section 2.1.3) and how to ensure atomic writes and durability (Section 2.1.4). In Section 2.1.5, we present programming interfaces and APIs for PMem.

2.1.1 Types

The Storage Networking Industry Association (SNIA) defines persistent memory as a "storage technology with performance characteristics suitable for a load and store programming model" [134], i.e., as a technology that is *persistent* but can be used like regular *memory*. In this thesis, we use the term persistent memory or PMem to describe this technology, but over the years various other names have emerged. These are, e.g., non-volatile memory (NVM), storage-class memory (SCM), and non-volatile RAM (NVRAM). Alternative abbreviations for PMem include PMEM, PM, or PMM. For consistency, we use only PMem.

Large-scale persistent memory is currently based on one of two designs: 3D XPoint (NVDIMM-*P*) or DRAM + flash (NVDIMM-*N*). 3D XPoint, developed by Intel and Micron, is the underlying technology of Optane [60]. It is the only publicly available *true* PMem, in which a single storage medium allows for both byte-addressability and persistence. DRAM + flash storage designs are employed in PMem offered by, e.g., HPE [36]. These battery-backed NVDIMM-Ns flush their state to flash chips on power failure.

According to the JEDEC standards, NVDIMM-Ns are seen as regular DRAM by the server while NVDIMM-Ps are viewed as separate storage with additional changes

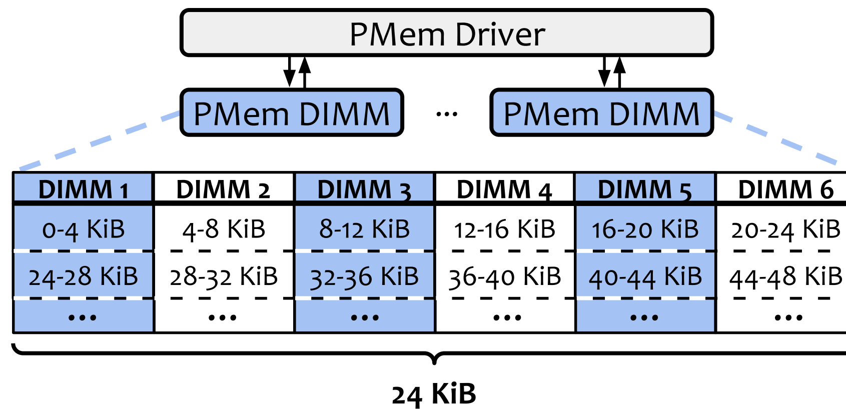

Figure 2.1: Six interleaved Optane DIMMs span a continuous 24 KiB region.

to the DDR4 protocol [72, 73]. Future PMem technology is expected to follow the NVDIMM-P standard, as this allows for larger capacity and extended functionality, while NVDIMM-Ns are limited by DRAM [74]. Currently, Optane PMem is the only available NVDIMM-P implementation. Various other PMem designs have been announced or are actively developed. These include Nano-RAM [100], phase change memory [86, 127, 154], resistive RAM [10], and magnetoresistive RAM [48].

While NVDIMM-Ns have been available for many years, they have not achieved widespread adoption. On the other hand, Optane, as a new technology, has received a lot of attention in academia and achieved initial adoption in industry, e.g., in SAP HANA [53]. As NVDIMM-Ns are essentially DRAM and have DRAM performance, we focus on NVDIMM-Ps in the form of Optane in this thesis.

2.1.2 Intel Optane

Intel Optane is currently the only commercially available NVDIMM-P implementation. It is based on Intel's 3D-XPoint media. While parts of the description in this section apply to all NVDIMM-P-compliant technology, certain aspects are Optane-specific. In this section, we cover some details of Optane that are not specified for NVDIMM-Ps and may be different in other implementations.

Interleaving

When installing Optane PMem in a server, users can choose between an *interleaved* and *non-interleaved* setup. With interleaving, data is striped across all available DIMMs, as shown in Figure 2.1. This setup is similar to RAID 0 for secondary storage. One stripe is 4 KiB, so in a common setup with six DIMMs, a continuous

24 KiB chunk of data is spread across all six DIMMs. With this layout, users get transparent parallelism due to a uniform distribution of data across DIMMs.

Without interleaving, each DIMM covers the continuous memory region given by its capacity. In this case, the application must handle parallelism for continuous access, as it is served from a only single DIMM. This layout is beneficial if the application on top explicitly handles data placement, as it gives developers full control over data locations.

Access Modes

Optane, in combination with Intel Xeon CPUs, offers two modes to run in, *Memory Mode* and *App Direct Mode*. In Memory Mode, PMem acts as a large volatile memory extension to regular DRAM. In this mode, memory is split into *near* and *far* memory, where near memory is DRAM and far memory is PMem. Users have no control over where memory is written to or accessed from, as this is handled by the OS. The OS treats DRAM as an "L4" cache, to which it first writes all data. Only when DRAM is full, does it write to PMem. As DRAM acts as a cache in front of PMem, the total memory capacity of the system is that of all PMem DIMMs without DRAM. For example, a system with 768 GB PMem and 96 GB DRAM has a total memory capacity of 768 GB and not 864 GB.

A key advantage of this mode is that legacy applications can use it without any code modification, as all memory is exposed as DRAM with a higher capacity. However, persistence is not guaranteed in this mode, i.e., users must treat all memory as volatile. Additionally, as DRAM acts as a cache, an L3 cache miss first results in an "L4" lookup to DRAM, which in turn leads to a PMem lookup on a miss. This increases latency by an additional DRAM access compared to direct access to PMem.

In App Direct Mode, memory is split into two explicit regions over which the developer has full control. The key advantage of this mode is that persistence is guaranteed if used correctly. However, existing applications must be adapted to explicitly access PMem. As memory is split into two regions, the system's capacity is combined, i.e., to 864 GB in the example above. As this mode offers more control as well as persistence, we focus on it in this thesis. The remainder of this section assumes that we run PMem in App Direct Mode.

Access Granularity

The internal physical media access size of Optane is 256 bytes. Conceptually, this is similar to a block device with a, e.g., 4 KiB page size. It is possible to load and

	100 Series	200 Series	300 Series
read	1.75 GB/s	2.03 GB/s	5.28 GB/s
write	0.58 GB/s	0.79 GB/s	1.63 GB/s

Table 2.1: Maximum per-DIMM Optane performance per generation for random 64-byte reads and writes, as reported by Intel.

store individual 64-byte cache lines but this results in read and write amplification in the Optane DIMM. Most systems designed for Optane optimize for this 256-byte granularity, as access at this size yields the best performance. To mitigate write amplification, Optane employs a write combining buffer that tries to combine four adjacent 64-byte cache lines into a single 256-byte write.

Optane Generations

The Optane PMem product is available in three generations, the 100 Series (code name Apache Pass), the 200 Series (Barlow Pass), and the 300 Series (Crow Pass). All generations offer DIMMs with 128, 256, and 512 GB capacity. Each generation requires a new corresponding Intel Xeon CPU generation, i.e., the 100 Series requires at least a 2nd generation Xeon CPU (Cascade Lake), the 200 Series requires a 3rd generation Xeon CPU (Ice Lake), and the 300 Series requires a 4th generation Xeon CPU (Sapphire Rapids). Key differences between generations are that the 200 Series comes with eADR support (see Section 2.1.4) compared to ADR in the 100 Series and that the 300 Series supports Compute Express Link (CXL) 1.1.

As the Optane generations are tied to CPU generations, the maximum PMem capacity per generation is also tied to the CPU. With Cascade Lake, a single CPU has six memory channels, supporting a maximum of 6×512 GB = 3072 GB. Ice Lake and Sapphire Rapids CPUs have eight channels, allowing for a total of 4096 GB per socket.

As Optane and DRAM share the same memory bus, the bus speed must be set equally. Optane is only supported in a 2 DPC (DIMM per channel) setup with one DRAM DIMM per Optane DIMM. Using Optane slightly decreases the maximum bus speed from a DRAM-only 1 DPC setup. For 100 Series Optane and Cascade Lake, 2 DPC is limited to 2666 MT/s [60, 64], compared to 2933 MT/s for a DRAM-only 1 DPC configuration. For 200 Series Optane and Ice Lake CPUs, the limit is 3200 MT/s for both 1 and 2 DPC [61, 65]. The third Optane generation and Sapphire Rapids CPUs support up to 4400 MT/s [66, 67] for 2 DPC, which is lower than Sapphire Rapid's 4800 MT/s limit for 1 DPC.

We perform an extensive evaluation of 100 and 200 Series Optane in Chapter 3, but this does not include the recent 300 Series. To provide a small performance overview, we show the 64-byte random access performance per DIMM as reported by Intel [60, 61, 66] in Table 2.1. In it, we see that each new generation brings a performance boost, especially from the 200 to 300 Series. Nominally, a common 100 Series setup with six Optane DIMMs supports 6 × 1.75 GB/s = 10.5 GB/s random reads and 3.48 GB/s random writes. A 200 Series setup with eight DIMMs can achieve up to 8 × 2.03 GB/s = 16.24 GB/s random reads and 6.32 GB/s random writes and a 300 Series setup can achieve up to 8 × 5.28 GB/s = 42.24 GB/s random reads and 13.04 GB/s random writes. Due to Intel discontinuing Optane, we were not able to verify these numbers for the 300 Series. For the 100 and 200 Series, the performance aligns with our results as shown in Chapter 3.

Optane in the Storage Hierarchy

To position Optane in the storage hierarchy, we briefly compare it to DRAM and SSDs. Due to widely different performance characteristics across devices, memory/storage technologies, and generations of the same technology, we provide only a rough performance outline of Optane in the *"storage jungle"* [56]. A single DDR-5 DRAM DIMM with 4800 MT/s achieves ~38 GB/s read bandwidth, which is 7× higher than the Optane 300 Series. Depending on the generation, a single Optane DIMM achieves comparable bandwidth to modern NVMe SSDs, which can read and write between 2 and 7 GB/s per device (with 4 KB page granularity). As multiple Optane DIMMs are commonly installed together, the accumulated bandwidth exceeds a single NVMe SSD. However, recent work shows that multiple NVMe SSDs combined in a RAID also achieve more than 50 GB/s, which is similar to a fully-stocked Optane server [50]. While the bandwidth is similar between Optane and modern NVMe SSDs, a key difference is access latency. Random PMem access is in the order of hundreds of nanoseconds, while SSDs require tens or hundreds of microseconds, i.e., one order of magnitude difference [50]. Thus, the potential of Optane over SSDs is greatest when used for workloads requiring low latency.

2.1.3 Accessing Persistent Memory

The Storage Networking Industry Association (SNIA) defines an NVM Programming Model (NPM) [134], which specifies a unified access model for PMem. This model allows for the integration of a wide range of storage technologies, beyond only Intel's Optane product. We show the two PMem access modes of this model in Figure 2.2. Applications either access PMem via regular filesystem interfaces (shown

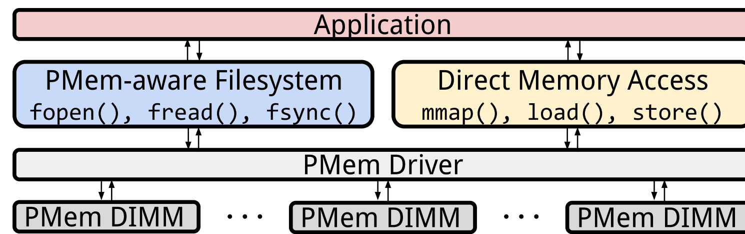

Figure 2.2: Standard PMem access modes.

on the left side) using calls such as `fopen`, `fread`, `fsync`, or they access PMem via memory-mapping (shown on the right side) using calls such as `mmap`, and `load` and `store` instructions.

The filesystem interface allows existing applications to use PMem as a drop-in replacement for common disk-based interaction, while the second mode allows applications to use PMem identically to DRAM. The programming model allows for memory mapping of files, i.e., combining both modes. In this case, files are used to logically structure raw memory chunks but PMem is accessed directly without the overhead of regular file I/O.

File I/O

To use PMem as a drop-in replacement for secondary storage, users can access it via a regular filesystem. In this case, PMem is exposed as a block device, and a filesystem, e.g., ext4 or xfs, is created on top of it. From a user's perspective, this is identical to a regular disk-based filesystem.

Regular file I/O commonly goes through the operating system's page cache, i.e., a copy of the storage content is kept in DRAM for faster access and modification. Due to PMem's byte-addressability, this copy is not necessary but it consumes DRAM capacity. To avoid this copy, PMem-aware filesystems offer *direct access* (DAX). In a DAX filesystem, data is read directly from the PMem DIMM and modifications are written directly to it without an intermediate copy. However, due to the block-device characteristics, all access occurs at page granularity, i.e., 4 KiB. This does not leverage the byte-addressability of PMem and causes high read and write amplification for small reads and writes. To better utilize PMem bandwidth, the preferred access mode is via memory mapping and load/store semantics, which we cover in the following.

In previous work, we show that access via the filesystem has a 5–10% overhead compared to direct memory access to and from a character device due to memory zeroing on page faults [31]. Directly mapping memory from the character device

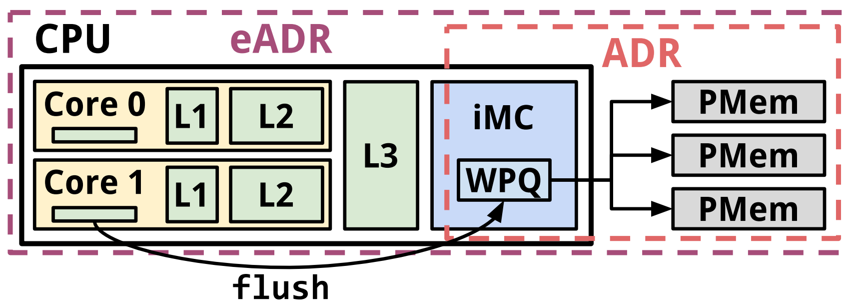

Figure 2.3: Writing to NVDIMM-Ps from the CPU.

avoids this overhead but access to the raw device requires full control of it without the advantages of a filesystem, e.g., structuring data or fine-grained access control.

Load/Store

To leverage the byte-addressability of PMem, developers can memory-map PMem directly from the PMem DIMM into their application's virtual address space. To do this, they issue an `mmap` call to the PMem-aware DAX filesystem or the PMem character device directly. Due to DAX, every byte that is accessed in the memory-mapped virtual address range is read and written directly from and to PMem via regular CPU load and store instructions, allowing for access at cache line granularity. This enables developers to create complex data structures in PMem as in DRAM.

For data modifications in DRAM, in most cases, it is not important when and how data is actually moved from the CPU to memory, as an application crash or power loss results in all data being lost. To ensure persistence with PMem, modified data in the CPU's current cache line must be explicitly written back (or flushed) to PMem, as data in the caches is not necessarily persisted. We outline how data is moved from the CPU to PMem in Figure 2.3. This model is based on Intel's Xeon processors [62] and Optane.

A CPU contains one or more *integrated memory controllers* (iMCs), which are directly connected to PMem via memory channels. To write data to PMem, the CPU must flush cache lines to a *write pending queue* (WPQ) within an iMC. Once data is in the WPQ, it is in the *Asynchronous DRAM Refresh* (ADR) Domain, which is guaranteed to be persisted, even on power loss. The WPQ then issues the write to the correct PMem device. On Intel CPUs, available instructions to flush a cache line are:

» `clflush` (cache line flush): Flushes the cache line and invalidates it, i.e., the next access to this cache line must fetch data from PMem. This is mainly a

legacy instruction that should not be used. Instead, developers should use `clwb`. `clflush` can be called via, e.g., `_mm_clflush(addr)`.

- `clwb` (cache line write back): Flushes the cache line but does not invalidate it, i.e., data is written to PMem but it still remains valid in the cache hierarchy. The next access to this cache line can be answered from cache if it was not evicted otherwise in the meantime. This is the recommended instruction to write data to PMem. `clwb` can be called via, e.g., `_mm_clwb(addr)`.
- `ntstore` (non-temporal store): For data with low or no temporal locality, i.e., it will not be accessed in the near future, Intel also offers a non-temporal store. This completely bypasses the cache hierarchy, offering better performance than the other instructions, which perform additional cache-related operations. This instruction is useful, e.g., for large sequential writes or logging. A non-temporal store can be issued via, e.g., `_mm512_stream_si512(destination, data)`.

A read request to PMem is posted to a read pending queue (RPQ) at cache line granularity. Following a request, data is also returned from the DIMM at cache line granularity, irrespective of the underlying physical granularity, e.g., 256-byte for Optane. While WPQs and RPQs are Intel-specific, they are based on the NVDIMM-P standard that describes a write buffering mechanism. Future PMem is likely to work similarly.

Communication via the DRAM memory bus (DDR) is synchronous [75]. This is not the case for a shared DRAM and PMem memory bus, as PMem is slower than DRAM. To overcome the varying latency in Optane, Intel uses a modified DDR4 protocol called DDR-T to support asynchronous communication between the WPQ and Optane. While DRR-T is Optane-specific, synchronous memory buses are a general problem that vendors need to solve when supporting different memory types. Compute Express Link (CXL) is emerging as an industry-wide solution to handle larger memory capacity with slower access [30]. We discuss the implications of PMem research for future CXL-aware designs in Chapter 6.

2.1.4 Atomicity and Durability

An important distinction between memory-mapped files on disk and PMem is that traditionally data is copied to a page cache in DRAM, which is then modified and flushed back. When memory mapping PMem, it is accessed directly (via DAX) and not copied to a DRAM page cache. Any modification to the data, if flushed correctly, is directly performed in *persistent* memory. This changes the failure granularity of

data modification compared to file-backed memory. For traditional files, developers must consider various failure cases, such as torn writes, but at the granularity of, e.g., a 4 KiB page write at an explicit point in the application. For PMem, developers must explicitly control data persistence and handle low-level crash consistency for *every* write to PMem.

As data in the caches is not persisted, developers must issue explicit flush instructions (see Section 2.1.3). However, in addition to explicit flushes, data might be randomly evicted from the cache, resulting in unexpected data persistence. Even if the developer does not issue an explicit flush, some data modifications might be written to PMem. As current CPUs provide only 8-byte atomic writes, random 64-byte cache line evictions may cause an inconsistent state after a crash for modifications larger than 8 byte. Thus, programmers must carefully design fine-grained PMem data access to ensure application correctness.

Additionally, programmers have to ensure correct store ordering. Modern compilers and CPUs may re-order instructions to improve performance, e.g., through better pipelining. However, this may lead to re-ordering of persist instructions, resulting in correctness bugs [99]. We show an example of this in the code below.

```
void insert_item(vector& vec, int x) {
  vec[vec.size] = x;
  // Flush cache line to PMem
  _mm_clwb(&vec[vec.size]);
  vec.size++;
}
```

In his example, we append an integer to a vector and then increase the current size of the vector. Depending on the compiler, this may first store a copy of the size in a register r, update the vector's size, then write x to the position stored in r, and finally flush it. For data in DRAM, this execution order makes no difference to the user as all data is lost after a crash. For data in PMem, this order may not be correct. Due to out-of-order execution in the CPU, the change to the size may be observed by later instructions, which then assume that data has been written and, e.g., report a successful operation to the user. However, if the application crashes before x was actually flushed to PMem, the execution was not successful. When restarting after the crash, x cannot be recovered and the application is in an inconsistent state.

To avoid such reordering, programmers must explicitly issue memory fences, e.g., via an `sfence` instruction on x86 [63]. This ensures that all modifications issued before the fence are globally visible before later modifications become visible, i.e., they can be observed by other threads. A common pattern when programming for

PMem is to *i)* perform the data modification, *ii)* flush it, and *iii)* perform a *store fence* to ensure that no later instruction is executed before the flush is completed. After the store fence, *iv)* metadata is updated to indicate the modification's validity. In our example, this would be done as shown below.

```
void insert_item(vector& vec, int x) {
  vec[vec.size] = x;           // i) Modify
  _mm_clwb(&vec[vec.size]);   // ii) Flush
  // Ensure that vec.size
  // is not modified before
  // x is flushed
  _mm_sfence();               // iii) Fence
  vec.size++;                 // iv) Metadata
}
```

Correctly moving data from CPU caches to PMem burdens programmers due to these correctness issues. It also incurs performance penalties due to additional CPU instructions. However, solutions exist to mitigate the correctness issues and performance penalties. Intel's 3rd Generation Xeon processors (Ice Lake) introduce an *enhanced* ADR (eADR) [61]. This includes all caches in the ADR, i.e., ensuring the persistence of all cached data in case of power loss (see Figure 2.3). This new design removes the necessity of explicit flushing. However, it still requires store fences for correct ordering and data can still be randomly and/or partially evicted, requiring careful data structure designs. A recent study finds that missing flushes are a common mistake in various PMem applications and libraries [116]. An eADR protects the user from this class of bugs.

2.1.5 Programming Interfaces and APIs

As PMem is commonly mapped into the application's virtual address space, developers can interact with it the same way they interact with DRAM. However, we show that they must pay attention to flushing, store ordering via fences, and crash consistency to write correct applications. This complexity results in numerous bugs that are hard to detect [116]. To aid developers with PMem programming, various PMem libraries have been created under the Persistent Memory Development Kit[1] (PMDK) [124]. These contain, among others, general purpose utilities for persistent memory development (libpmem[2]), transactional objects and memory allocations

1 Due to Intel discontinuing Optane, most projects in PMDK have been deprecated as of 2023.
2 https://pmem.io/pmdk/libpmem/

(libpmemobj[3]), C++ bindings (libpmemobj-cpp[4]), or logging utilities (libpmemlog[5]). To have more explicit control over PMem access, we do not rely on PMDK libraries in this thesis.

3 https://pmem.io/pmdk/libpmemobj/
4 https://pmem.io/libpmemobj-cpp/
5 https://pmem.io/pmdk/libpmemlog/

3 Benchmarking Persistent Memory Access

The majority of this chapter has been published in [15].

3.1 Introduction

Both research and industry have awaited the arrival of persistent memory (PMem) as a new layer in the storage hierarchy for many years. PMem promises byte-addressability and persistency at DRAM-like speed with SSD-like capacity. These characteristics have the potential to cause a major performance increase in storage systems, such as databases and key-value stores. Thus, research on system design incorporating PMem was published long before real PMem hardware was available, based on simulations [7, 123, 141]. Now that byte-addressable, persistent memory is finally available commercially, Intel's Optane DC Persistent Memory has received a lot of attention in initial performance evaluations [31, 47, 130, 147]. These evaluations provide valuable insights into the general performance and unique characteristics of first-generation Optane.

Research on data structures [25, 97, 101] and storage systems [14, 26, 91] that incorporate these insights often have to perform additional hardware-specific micro benchmarks to understand the specific nuanced PMem behavior for their expected workloads. Initial research shows that Optane's performance is highly dependent on the workload with major differences between read and write behavior.

Due to limited availability and high prices, researchers often have access to only one PMem server. Thus, new systems built for PMem are designed, implemented, and optimized on a single server with a single combination of PMem, DRAM, and CPU. However, many factors impact PMem performance that are not yet well understood, e.g., the DIMMs' size and power budget or the number of DIMMs in the server. As PMem is a new technology, it is unclear how well these initial designs generalize across PMem configurations. On top of various configurations, with the availability of second-generation Optane, new performance characteristics are introduced.

Based on the configuration space and workload-tailored micro-benchmarks of previous work, we identify the need for a comparable workload-driven analysis of PMem. We propose PerMA-Bench, a configurable benchmark framework

that analyzes the bandwidth, latency, and operations per second for customizable database-related PMem access. In PerMA-Bench, we pre-define various workloads that cover the maximum achievable performance of core access patterns (sequential/random reads/writes), as well as a wide range of realistic, database-related access patterns, such as updates, lookups, and scans in tree and hash indexes. These complex patterns include pointer-chasing loads, mixed read/write access, and hybrid PMem/DRAM access. Additionally, PerMA-Bench allows users to run custom workloads tailored toward their design choices. With PerMA-Bench, we propose a tool that provides insight into the performance of PMem at a general and workload-specific level. Users can explore the performance of new access patterns but also validate existing designs. Based on these findings, users can validate their design choices without having to write their own benchmark application and find areas of improvement in existing designs.

Based on PerMA-Bench, we perform the first extensive evaluation of Optane for database workloads across various DIMM sizes of the first and second generation. We compare the performance of all three DIMM sizes of 100 Series Optane and one DIMM size of the 200 Series. Additionally, we show the impact of varying the number of DIMMs, DIMM power budgets, and memory bus speeds.

We validate our results with existing implementations and show that they do not fully utilize the performance improvements across Optane generations. We show that the choice of persist instruction has a high performance impact and that avoiding explicit flushes in eADR does not always yield the best results. Based on our results, we identify and discuss eight aspects that future work should take into account when designing PMem-aware systems. With the availability of more PMem hardware, research has to consider more than one setup to achieve general PMem-optimized designs.

In addition to PMem's performance, its price-performance is important to determine whether PMem is suitable for users' needs. In this chapter, we perform a price-performance comparison of various server configurations. Our comparison shows that PMem's price-performance is competitive with that of DRAM and is often even better. Thus, in addition to providing persistence, PMem can act as a larger, cost-effective general memory when used correctly. In summary, we make the following contributions:

1) We propose PerMA-Bench, a configurable benchmark framework to analyze bandwidth, latency, and operations per second for customizable database-related PMem access.

2) We perform an extensive evaluation of PMem performance across four PMem

servers and additional per-server configurations to show the impact of individual server setups on bandwidth utilization and latency.

3) We compare the price-performance for key workloads across all servers and show that while there are large differences across Optane, PMem is generally competitive with DRAM.

4) We discuss eight general and implementation-specific aspects that influence the performance of PMem and need to be taken into account for the design of future PMem-aware systems.

The remainder of the chapter is structured as follows. In Section 3.2, we introduce the PerMA-Bench framework. In Section 3.3, we present PerMA-Bench results on various hardware configurations, which we then use in Section 3.4 to discuss the price-performance of PMem. Finally, we discuss our findings (Section 3.5) and related work (Section 3.6), before concluding in Section 3.7.

3.2 Introducing PerMA-Bench

In this section, we introduce PerMA-Bench, a benchmark framework for persistent memory access. When designing new systems or database components, it is important to know the performance of the underlying memory access. This understanding allows users to tune their system towards better PMem utilization. PerMA-Bench supports *basic* and *complex* memory access patterns to evaluate the performance of PMem. Basic access patterns determine the maximum achievable bandwidth utilization and latency by repeatedly executing the same operation, i.e., a simple read or write. Complex patterns allow users to evaluate specific designs via chained read/write access from/to DRAM and PMem with varying persist instructions and access sizes. Based on these complex patterns, users can model, e.g., new index structure designs and gain insight into their memory performance before implementing them.

We present the runtime of PerMA-Bench in Section 3.2.1. Then, we present options for workload customization in Section 3.2.2 and briefly discuss supported memory store semantics in Section 3.2.3.

3.2.1 Runtime

PerMA-Bench is designed as a standalone benchmark executable. Users interact with PerMA-Bench via configuration files and command line arguments. Based on

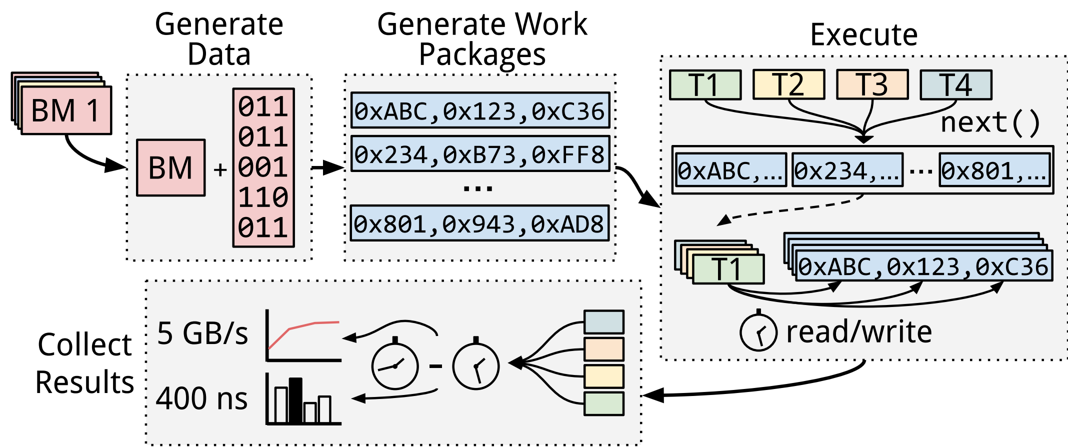

Figure 3.1: Execution cycle of a benchmark in PerMA-Bench.

these specified configuration parameters, individual benchmarks are created. We show the execution cycle of PerMA-Bench in Figure 3.1. For each benchmark (BM) that is created, PerMA-Bench performs four steps.

First, all data files are prepared and filled with random data. The files can be located in PMem or DRAM to allow for hybrid setups, which are common in current PMem research. Next, individual work packages are generated, which contain a pointer for each operation that is to be executed on the data. Work packages of, e.g., a random read benchmark with 100 operations contain 100 pointers to random offsets in the data file. For sequential access, packages contain 100 pointers to contiguous addresses. This allows for execution in a tight loop instead of requiring logic per benchmark type. For raw performance benchmarks, all work packages are pre-generated to avoid the overhead of generation during execution.

During the execution, N threads are spawned ($N = 4$ in Figure 3.1). Each thread then continuously pulls a new work package from a shared queue and executes the requested operations on that work package. PerMA-Bench adopts this work package approach for two reasons. First, to avoid stragglers when statically assigning work to threads. During our evaluation, we observed that hyperthreading often leads to very unbalanced execution times, skewing the final results. Second, as the general concept of work-stealing is employed in many databases exactly to avoid execution skew, PerMA-Bench represents a common execution model where workers operate on small work packages, e.g., via morsels [88]. To avoid long-running packages and the skew this entails, work packages contain 64 MB worth of operations by default, which execute in less than 100 ms in most cases. All threads are synchronized via a barrier before the execution starts to ensure concurrent execution.

Listing 3.1: Example config YAML file.

```
1 hash_index_update:
2  matrix:
3   number_threads: [ 1, 4, 16 ]
4  args:
5   custom_ops: "r_256,w_64_cache_128,w_64_cache_-128"
6   total_memory_range: 10G
7   number_operations: 100000000
```

We find that results are not impacted by a warm-up phase within workloads, as they exceed cache and queue sizes. However, as pre-faulting pages before writing to them avoids kernel page zeroing during execution [68], we provide a "warm-up" pre-fault flag.

After all work packages have been processed, PerMA-Bench collects the results of all threads to calculate the final benchmark results. PerMA-Bench determines the total execution time as the time between all threads' earliest begin timestamp and latest end timestamp. This captures the entire execution duration but may underestimate the actual performance slightly, as some threads are already idle while others are finalizing their work. However, in PerMA-Bench, we perform workload-driven performance evaluation and from a higher-level perspective, this approach captures the total time it takes to complete a given workload. Based on the total number of processed bytes or operations and the total execution time, PerMA-Bench calculates the overall throughput in GB/s or operations/s. If specified by the user, PerMA-Bench also samples the latency of individual operations. The sampled values are added to a histogram and presented in the form of minimum, maximum, average, and multiple percentile latencies.

3.2.2 Custom Workloads and Configuration

Besides the pre-defined workloads, custom benchmarks can be configured via YAML files and command line arguments. In this section, we present configuration options provided by PerMA-Bench with which users can express their specific workloads' access patterns.

Configuration Files. Benchmarks in PerMA-Bench are configured via YAML files. This format allows users to specify workloads manually and programmatically. We show an example configuration in Listing 3.1. Each configuration file consists of two main parts, the matrix arguments and the general arguments. The `matrix`

block (Lines 2–3) describes which dimensions should be evaluated in the benchmark. Each matrix argument is provided as a list, from which PerMA-Bench creates a benchmark for each combination in the cross product, i.e., three benchmarks in this example. The `args` block (Lines 4–7) describes which general arguments should be used for every combination. In this example, we configure a hash index update workload and evaluate it for 1, 4, and 16 threads.

Custom Operations. In Line 5, we show the definition of a custom operation. These model complex, pointer-chasing memory access patterns instead of simple, independent reads or writes. They are created in a chain in which each operation op is responsible for calling the next operation op' once complete. When op has read the random data d, it passes d to op', which then determines the next address based on d. By requiring data from op in op', PerMA-Bench ensures that op' is not executed before op was completed.

In the example, PerMA-Bench reads 256 Byte (`r_256`), e.g., a hash bucket, at a random location r_a within the allocated data range. Then, two 64 Byte *Cache* write instructions (`w_64_cache`) are executed. The first is performed with an offset of 128 Byte (`_128` $= r_a + 128$), e.g., to store data in a hash bucket. The next write operation jumps back 128 Byte to the start of the bucket (`_-128` $= r_a$) to update metadata. This pattern of storing data in a node and updating metadata afterwards is common in PMem data structures [14, 97, 101, 123]. As 64 Byte cache line flushes are combined to 256 Byte in Optane, it is important to model adjacent writes correctly instead of simulating them with writes to the same cache line while supporting different persist instructions. Varying these sizes also gives users insight into the impact of prefetching in PMem. Additionally, PerMA-Bench supports mixing DRAM and PMem for hybrid access, as used, e.g., in PMem B-Trees [25, 97, 123, 148].

Benchmark Parameters. PerMA-Bench currently offers 19 configuration parameters that allow users to define a wide range of individual benchmarks without having to write C++ code for each of them. Users can specify, e.g., PMem/DRAM memory ranges, access size, sequential/random execution, number of partitions and threads (for data parallelism), custom operations, work package size, runtime, and file pre-faulting.

Other Features. PerMA-Bench supports running different workloads as task-parallel benchmarks. Concurrent workloads might impact each other as one benefits from caching, while the other fills the cache with unwanted data. Users can also specify *NUMA-aware execution* of benchmarks on *far* or *near* CPUs to explore how data placement impacts their workloads and whether NUMA must be considered in their design. PerMA-Bench additionally allows users to run all benchmarks in DRAM as a performance reference.

3.2.3 Persist Instructions

PerMA-Bench supports four persist instructions, *Cache*, *CacheInvalidate*, *NoCache*, and *None*. *Cache* represents a temporal store (`clwb`), *CacheInvalidate* represents a temporal store that invalidates the cache line (`clflushopt`), *NoCache* represents a non-temporal store (`ntstore`), and *None* performs no explicit flush instruction. Temporal refers to the inclusion of data in the cache hierarchy with the assumption of future access, i.e., temporal locality is likely. When temporal locality is unlikely, non-temporal instructions can bypass the cache completely, avoiding cache pollution. Not explicitly flushing is useful when persistence is not required, e.g., when storing intermediate results in PMem or when eADR ensures persistence. For *Cache*, *CacheInvalidate*, and *NoCache*, we add a store fence (`sfence`) afterwards to guarantee correct write ordering. PerMA-Bench uses Intel's AVX512 extension to write an entire cache line, i.e., 64 Byte or 512 Bits, in one instruction using `SIMD`-registers [63].

3.3 PerMA-Bench Results

In this section, we present the results of various PerMA-Bench workloads on multiple PMem server configurations. Our results give insight into both raw and workload-specific PMem performance to better understand PMem's use in database-inspired workloads. We evaluate various configurations to show how comparable previous results are across PMem setups, as they are often run on only one configuration, e.g., on one DIMM size or with a partially stocked server. These configurations allow us to draw more general conclusions about PMem as well as provide insight into how previously published systems and results apply to other setups.

We describe our evaluation servers in Section 3.3.1. We then show the bandwidth and latency results of PerMA-Bench's raw performance workloads in Section 3.3.2 and Section 3.3.3 to gain an understanding of the maximum performance of current PMem hardware. In Section 3.3.4, we discuss the results of database-related workloads and index structures to gain insight into the performance of PMem for more complex access patterns in actual systems and implementations. Finally, we investigate the impact of configurations affecting a single server in Section 3.3.5, i.e., by varying the number of DIMMs or the memory bus speed, as well as by disabling the prefetcher.

Table 3.1: Evaluated servers (single socket). Apache/Barlow refer to the code names of 100/200 Series Optane. All CPUs are Intel Xeon, all PMem is Optane.

Name *(Plot Label)*	**CPU**	**PMem**	**DRAM**	**OS**
Apache-128 *(A-128)*	Cascade Lake 18 Cores (2.7 GHz)	6x 128 GB 100 Series @ 2666 MT/s | 15 Watt	6x 16 GB	Ubuntu 20.04 (5.4 kernel)
Apache-256 *(A-256)*	Cascade Lake 18 Cores (2.6 GHz)	6x 256 GB 100 Series @ 2666 MT/s | 18 Watt	6x 16 GB	Ubuntu 20.04 (5.4 kernel)
Apache-512 *(A-512)*	Cascade Lake 24 Cores (2.4 GHz)	6x 512 GB 100 Series @ 2666 MT/s | 15 Watt	6x 64 GB	Ubuntu 20.04 (5.4 kernel)
Barlow-256 *(B-256/B-D)*	Ice Lake 32 Cores (2.2 GHz)	8x 256 GB 200 Series @ 3200 MT/s | 15 Watt	8x 32 GB	Ubuntu 20.04 (5.4 kernel)

3.3.1 Setup And Methodology

We perform our evaluation on the four server configurations presented in Table 3.1. We refer to the servers as named in the table or via their label, e.g., Apache-128 or A-128. Apache Pass is the code name for the first generation/100 Series Optane DIMMs. Barlow Pass is the code name for the second generation/200 Series Optane DIMMs. All servers are equipped with Optane DC Persistent Memory DIMMs. All Optane DIMMs are configured interleaved, i.e., striped in 4 KB blocks and accessed in App Direct mode. All measurements are performed on a single socket. To avoid measuring zeroing of requested pages by the kernel, files are pre-allocated and pre-faulted by default before running the benchmark, as recommended [68]. Unless stated otherwise, we generate a fixed amount of random data for each benchmark, depending on the benchmark configuration. In all experiments, we use 1 GB = 2^{30} Byte.

The A-256 server is configured with an 18 Watt average power budget per DIMM, the other 100 Series servers are configured to 15 W. While Optane allows the power budget to be set from 12 to 15 W (A-128, B-256) or 18 W (A-256/512), it is set by the vendor on the evaluated servers and cannot be reconfigured. Analyzing the power range allows us to show that even within the same generation, server configuration has a large performance impact.

When drawing performance conclusions, we also consider official performance numbers provided by Intel [60, 61] and previously reported numbers in research [31, 47, 71, 130, 147]. To provide a reference to well-known performance numbers, we also evaluate all experiments in DRAM. We show the results of the DRAM runs on Barlow-256 (shown as *B-D* in the plots). The DRAM performance in the Apache

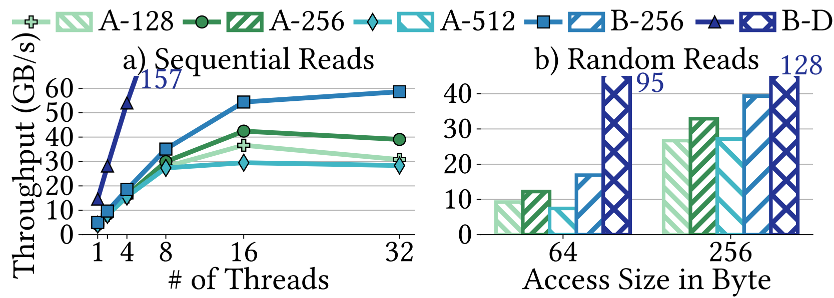

Figure 3.2: Sequential and random read bandwidth.
a) Fixed to 4096 Byte Access | b) Fixed to 16 Threads

servers is lower, so we omit them for space reasons. The full results can be found in our repository[6].

3.3.2 Raw Performance Workloads – Bandwidth

In this section, we present the bandwidth results of PerMA-Bench's raw performance workloads. We investigate the bandwidth utilization of all servers for sequential and random reads and writes. As the first part of our evaluation, these workloads provide insight into which performance can be achieved with current PMem hardware. Based on this knowledge, users can make decisions about the feasibility of PMem-specific implementations and their expected performance range in bandwidth-heavy applications.

Sequential Reads

We first discuss the throughput of sequential reads across all servers, as they are a core database access pattern. In this benchmark, we perform a sequential read workload on 50 GiB of randomly generated data with 4096 Byte access size and a varying number of threads. We show our results in Figure 3.2a. Within the first generation, we observe a difference of up to 44%, ranging from 29 to 42 GB/s. According to the official product sheet, A-128 and A-256 have the same read bandwidth under equal power budgets [60]. So the 24% difference between A-512 and A-128 is based on the DIMMs, while the additional 15% improvement from A-128 to A-256 is based on the higher power budget.

B-256 achieves ~40% higher bandwidth than its first-generation counterpart A-256 with 58 GB/s and a 60% improvement over A-128. B-256 is stocked with

6 https://github.com/hpides/perma-bench

8 DIMMs per socket instead of 6 DIMMs as in the 100 Series, leading to a 33% higher expected performance. Beyond this, we observe only a small improvement compared to the 18 W budget in A-256. But compared to A-128, which has the same read bandwidth as A-256 under equal power budgets [60], we see an additional 30% improvement. So for common 15 W setups, there is a notable performance increase between generations.

Our results show higher variance in throughput once hyperthreading is used and PMem limits are reached. This is observable for A-128 and A-256 with 32 threads, as both have only 18 physical cores. A-512 is more consistent, as it has 24 cores and B-256 even improves until 32 threads, which is the number of its physical cores. To achieve stable performance across servers, it is important to not exceed the number of physical cores when scanning data.

As a reference, B-DRAM's bandwidth reaches 98/145/157 GB/s for 8/16/32 threads, which is still significantly higher than PMem's. In the second generation of Optane DIMMs, the gap between PMem and DRAM even increases, from 2.3 to 2.7×. So while the bandwidth improved, there is still a clear advantage for DRAM in bandwidth-heavy applications. On the other hand, future systems must be able to process ~60 GB/s of data when reading from PMem, which is a major challenge when considering the cost of, e.g., random access data structures used in aggregations or joins. Thus, for most data-intensive applications, the bandwidth of sequentially accessing data stored in PMem is sufficient and does not constitute a bottleneck, unlike alternative secondary storage [31].

Random Reads

As indexes are core database components and essential to query performance, we investigate an index-inspired workload consisting of small, random, read-only operations, as commonly performed in hash or tree indexes. The bandwidth for uniform 64 and 256 Byte access across 10 GiB of random data is shown in Figure 3.2b. These access sizes represent the internal access granularity of Optane as well as standard cache-line-sized reads.

When considering PMem as random access memory as seen by the CPU, we see that it cannot achieve the same random to sequential ratio as DRAM. For 64 Byte, B-DRAM achieves 60% of the peak sequential performance, while the PMem servers achieve only 25%. Using the access granularity of Optane at 256 Byte, PMem achieves 67 – 92% and B-DRAM achieves 81%. Configurations with lower sequential performance achieve higher percentages in random access, reducing the gap from 40 to 20%. The second generation improves by 40% over A-256 and 80% over A-128.

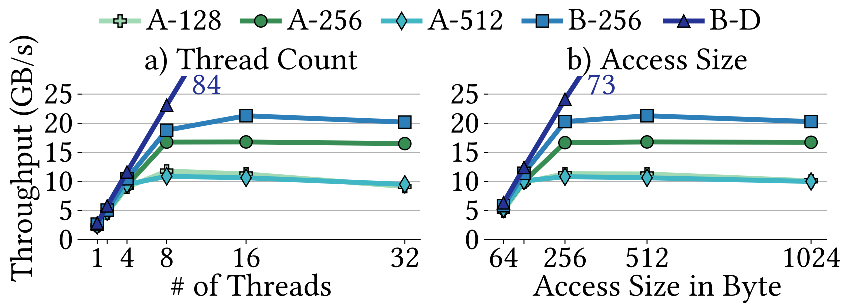

Figure 3.3: Thread and access size impact on sequential writes. *a) Fixed to 512 Byte Access | b) Fixed to 16 Threads*

Similar to sequential access, we see little improvement over the higher-powered DIMMs but large gains compared to the lower wattage DIMMs.

When designing for PMem, it is important to keep the read amplification of small reads and the access granularity in mind, as 256 Byte access achieves more than two-thirds of the peak sequential performance. Compared to volatile DRAM, the performance is still significantly lower. But for applications that need fast access to small persistent records, e.g., point lookups in a key-value store or persistent index operations, 17+ GB/s of random 64 Byte access and 32+ GB/s of random 256 Byte access allow future systems to re-think the cost of persistence, especially when considering other bottlenecks such as network or alternative secondary storage.

Sequential Writes

Inspired by logging workloads in databases, we show a sequential write benchmark of 30 GiB in Figure 3.3. We investigate varying the number of threads and write size. We use *NoCache* writes, as logged data does not require temporal locality.

In Figure 3.3a, we evaluate the bandwidth for 512 Byte sequential writes. Within the 100 Series, we observe a large difference between A-128/512 and A-256. A-128/512 achieve around 12 GB/s sequential write bandwidth, as shown in previous work [31, 130, 147]. A-256, on the other hand, achieves close to 17 GB/s, due to the higher power budget. This is 40% higher than previously published results for 100 Series Optane. We verify this bandwidth utilization in VTune to confirm that there is a significant difference even within the first generation DIMMs. Thus, it is highly beneficial to configure PMem with a higher power budget of 18 W for write-heavy applications.

We observe a large improvement from the regular-powered 100 to the 200 Series. At its peak, B-256 achieves 21.6 GB/s, which is 75% higher than A-128/512 and

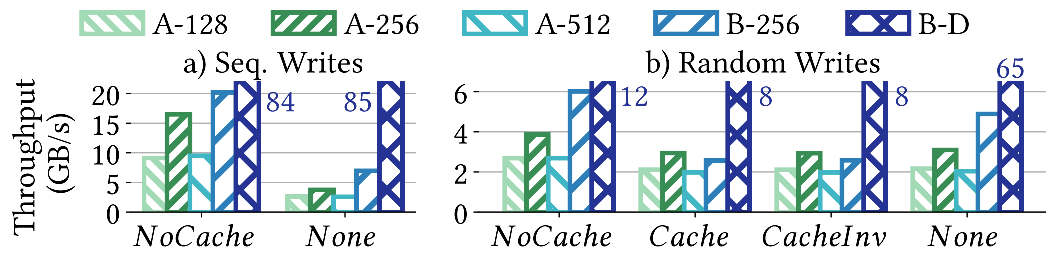

Figure 3.4: Impact of persist instruction on write bandwidth.
32 Threads | a) Sequential 512 Byte Write | b) Random 64 Byte Write

goes beyond the expected 33% increase due to more DIMMs. The improvement for sequential writes is also higher than that of sequential reads. However, compared to the high-powered A-256, we observe only a 30% improvement, i.e., none beyond the extra DIMMs. Nonetheless, for common configurations used in previous research, there is a large increase that encourages utilizing PMem even more for sequential writes, e.g., when logging or using log-based storage systems. When persistence is not needed, PMem performs significantly worse than DRAM, which achieves more than 40/80 GiB/s for 16/32 threads, i.e., a difference of 4×.

When scaling the access size for 16 threads, as shown in Figure 3.3b, we observe that all servers require at least 256 Byte to achieve peak bandwidth. However, for A-256 and B-256, this is more important than for A-128/512. The latter two servers perform close to their maximum with 128 Byte, while the other two servers improve by at least 70% from 128 to 256 Byte. B-256 even improves from 256 Byte to 512 Byte, before dropping again slightly for larger sizes.

We also observe that all configurations decrease slightly when increasing the number of threads beyond a certain point. This point is at 16 threads for A-256 and B-256, while it is at 8 for A-128 and A-512. The ideal configuration of threads and access size depends on the server and differs across generations. While all 100 Series servers in our evaluation peak at 512 Byte access, B-256 peaks with 256 Byte access and 32 threads. These slight performance differences across all servers indicate that fine-tuning for the individual server yields higher performance and cannot be easily generalized.

Persist Instruction

We evaluate the impact of different persist instructions on the bandwidth for sequentially writing 30 GiB and randomly writing 10 GiB. The results are shown in Figure 3.4.

With Ice Lake CPUs, Intel offers persistence for all data in the eADR (see Section 2.1.4), making explicit flushing optional. However, we see that for sequential write access, not flushing data strongly decreases bandwidth utilization by up to 4× compared to explicit stores. Randomly evicted cache lines impair write-combining within the DIMMs, resulting in random-access-like write performance. Thus, explicitly flushing is beneficial for sequential writes, even with the second generation server and eADR. For B-DRAM, on the other hand, there is no difference between both options, as there is no write amplification when randomly evicting data from cache.

For random 64 Byte writes, we evaluate all four persist options, i.e., *NoCache*, *Cache*, *CacheInvalidate*, and *None*, as shown in Figure 3.4b. Explicitly bypassing the cache via non-temporal stores achieves the highest bandwidth in all servers. Non-temporal stores also surpass explicit flushing in DRAM, which shows that there is a ~25% overhead of passing stores through the cache hierarchy.

Issuing no flush (*None*) is only marginally better than explicit temporal stores for Apache servers, but nearly 2× better for B-256. Thus, in eADR servers, users benefit from reduced code complexity and higher bandwidth when not explicitly flushing. Based on the different performance characteristics of flushes in the 100 and 200 Series, future work should re-evaluate flushes in existing PMem-optimized index structures. Significant work has been done to reduce the number of flushes and to decide which instructions to use [97, 101, 123, 148, 156], but it is unclear whether the choices apply to future Optane or are tailored only towards 100 Series.

3.3.3 Raw Performance Workloads - Latency

In this section, we evaluate the latency of raw PMem access across all servers. Understanding latency allows users to evaluate the feasibility of PMem-specific implementations in latency-critical applications and gain insight into the expected performance.

Operation Latency

In Figure 3.5, we show the average latency of five operations in PMem: a single 256 Byte read, and four variants of a 256 Byte read followed by a 256 Byte write to the same location with the supported persist instruction (*NoCache*, *Cache*, *CacheInvalidate*, *None*). We perform 100 million operations on 10 GiB of data and sample every 5000th operation. The results show that read latency is consistent across servers. While their read bandwidth differs significantly, there is no difference in

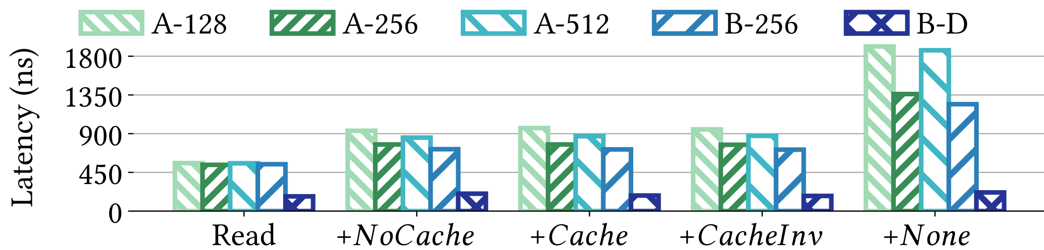

Figure 3.5: 256 Byte random read + write latency. *16 Threads.*

latency. However, the latency is still 3× higher than in B-DRAM, which is also approximately the factor between both for random read bandwidth.

When following the read with a write operation, latency is not equal on all servers. We observe that A-256 and B-256 have lower latency than the other servers across all flush operations. While read latency is bound by the latency of physical media access, write latency is more nuanced, as writes do not need to be flushed to the medium to be considered complete. Flushing to a full write pending queue blocks the caller and has higher latency. Due to B-256's higher bandwidth, more writes are flushed, freeing up space in the queue. The latency across all explicit persist instructions is consistent within each server, with *NoCache* having a slightly lower latency. Not flushing when writing 256 Byte of data has the highest latency (and lowest bandwidth), as randomly evicted cache lines cause high write amplification, blocking the write pending queues.

When running the same experiment with 64 Byte access, the latency is nearly identical for all instructions except *None*. For *None*, omitting the flush for consecutive memory addresses prevents efficient write combining. But for 64 Byte writes, write combining cannot be performed, so there is no disadvantage. For latency-critical applications, the choice of persist instruction is not relevant from a performance perspective. However, when writing more than 64 consecutive Bytes, an explicit flush should be used to benefit from write combining, even in eADR servers such as B-256.

While we observe a bandwidth increase across generations, latency has not improved. Additionally, we notice a correlation between higher available bandwidth and reduced write latency. However, most research focuses on bandwidth as a limiting factor of PMem compared to DRAM. Our results raise the question of whether future designs should shift their focus towards avoiding latency instead. Especially in latency-bound applications, new approaches may sacrifice bandwidth

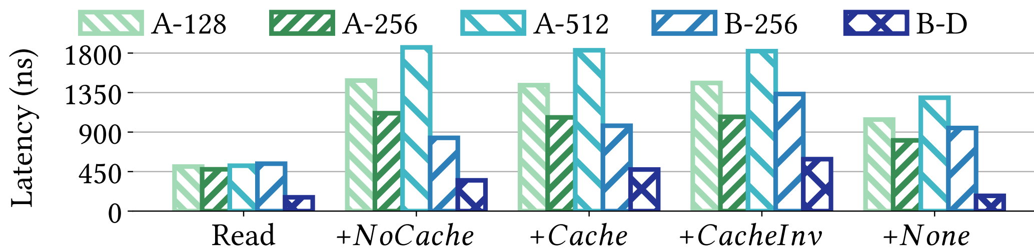

Figure 3.6: Double-flush latency. *64 Byte read + 2× 64 Byte write.*

to reduce latency, e.g., by writing additional data, which in turn reduces additional random lookups.

Double Flush Latency

Current PMem systems often store metadata for tree nodes, hash buckets, or storage pages in a single cache line and repeatedly update, e.g., counters, bitsets, or locks in that cache line [14, 101, 123]. While this practice is often used, some authors discourage it due to high latency [78, 138]. In this section, we evaluate the impact of different persist instructions when flushing the same cache line twice (*double-flush*). We perform 100 million operations on a 10 GiB range and sample every 5000th operation.

In Figure 3.6, we see that the double-flush latency for A-128 and A-512 is 2× of the single flush latency shown in Figure 3.5. In comparison, the double-flush latency of A-256 and B-256 is only marginally higher than the single flush. Under high load, these servers achieve higher bandwidth, which results in less pressure on the write queue and, in turn, reduces the latency of individual writes.

By comparing *Cache* and *CacheInvalidate*, we see that in the second generation Optane DIMMs there is actually a difference between the used persist instruction. The 2nd generation Xeon CPUs in the Apache servers do not fully implement `clwb`, internally mapping it to `clflushopt` instructions instead. As B-256's CPU supports true `clwb`, we observe that invalidating flushes (via `clflushopt`) have a higher latency due to the required memory read between the writes. While the latency of B-256 is slightly higher than that of A-256 for *None* and *CacheInvalidate*, it is lower for *Cache* and *NoCache* stores. Thus, we conclude that using non-invalidating flush operations on the same cache line is preferable for 200 Series Optane, as it does not include the penalty of invalidating the cache line, which occurred in the 100 Series.

3.3.4 Database-Related Workloads

In this section, we present the results of database-related workloads modeled in PerMA-Bench. The memory access patterns in these workloads are based on actual implementations of PMem index structures. Expressing complex memory access patterns in PerMA-Bench allows users to gain insight into their design choices at a memory-performance level before having to implement numerous options. This also helps to understand where performance is lost and where operations are close to the underlying memory performance. Our results show that both existing systems that were designed for a specific server configuration and systems that were designed pre-Optane do not fully utilize the underlying performance improvements of second-generation Optane.

First, we discuss the performance of PMem-aware index-inspired workloads compared to a DRAM-only version in Section 3.3.4. Then, we evaluate the performance of actual PMem-aware implementations based on our findings in PerMA-Bench. We show that current designs often cannot fully utilize the performance improvements of 200 Series Optane and avoiding explicit flushes in eADR does not always yield the highest bandwidth. To provide more general solutions, future work on PMem-aware systems must expand beyond designs evaluated on a single setup and reconsider design choices that may have been altered by newer characteristics of 200 Series Optane.

Database Index Operations

In this section, we cover a wide range of database-related index workloads in PerMA-Bench, run with 32 threads. When designing PMem systems, a DRAM-based version is often used as a comparison to show the efficiency of the chosen design. To cover this, we model our access based on PMem but run the experiments in both PMem and DRAM. We use the throughput of all access in DRAM as a baseline and show the factor × by which the throughput of the same access in PMem is lower. From our evaluation in the previous section, we know that DRAM's random read bandwidth is 5 – 10× higher than PMem's for 64 Byte access and 3 – 4.5× for 256 Byte access (see Section 3.3.2). DRAM's random write bandwidth is up to 2 – 6× higher for 64 Byte with explicit flushes, and 3 – 6× higher for 256 Byte writes. Unflushed writes have an up to 30× higher bandwidth. This shows the strong imbalance between DRAM and PMem for random access data structures, especially if only tiny amounts of data are changed, e.g., an 8 Byte pointer in an index. Understanding these differences is important to optimize access to each

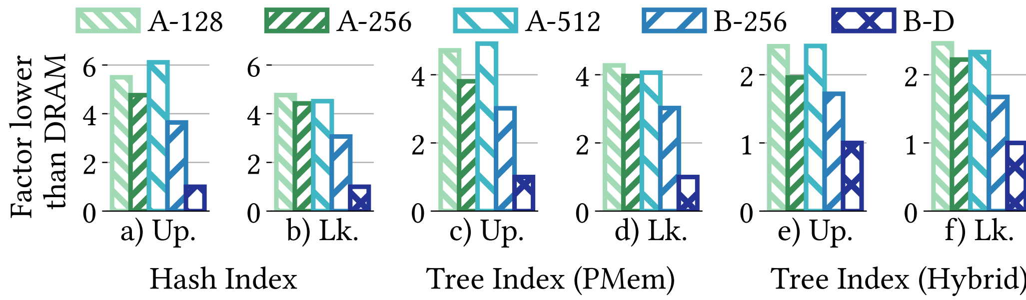

Figure 3.7: Factor × lower throughput than DRAM of updates *(Up.)* and lookups *(Lk.)* in PMem index workloads. *32 Threads. Access modeled after Dash (Hash), FAST+FAIR (Tree PMem), and FPTree (Tree Hybrid).*

memory type accordingly. We perform 100 million operations across 10 GiB in all workloads.

We model the access patterns for a hash index like Dash [101]. For lookups, its access consists of a 512 Byte read, representing two adjacent 256 Byte buckets, followed by two 64 Byte cache flushes for updates. In Figures 3.7a and b, we see how the improved random access performance of B-256 closes the gap to DRAM. As the access pattern of a hash index is $O(1)$ by design, the improvement should apply directly to real workloads. However, PMem still performs significantly worse than DRAM, as small updates to the index have a high write amplification, e.g., 16× for 16 Byte updates.

We model a PMem-only tree index after FAST+FAIR [58]. Based on the authors' implementation, we issue 3× 512 Byte random reads for a lookup and 4× 64 Byte cache flushes for an insert, as 50% of a node has to be moved on average when inserting a value into a leaf in FAST+FAIR. We see an average performance of around 4× in the 100 Series and 3× in the 200 Series. As this design operates on 512 Byte nodes, we observe the expected ~4× higher DRAM bandwidth for lookups. Updates perform slightly worse, but better than the raw DRAM bandwidth would suggest. However, four flushes per update (on average) are expensive, even in DRAM, as each one entails a memory fence instruction that clears all write buffers.

We represent a hybrid DRAM-PMem tree through FPTree [123]. PerMA-Bench issues 2× 2048 Byte DRAM reads and 1× 1024 Byte PMem read for lookups, followed by 3× 64-byte cache flushes for updates, based on the node sizes in the implementation that we use [52]. We see that the hybrid tree is closer to DRAM in relative performance, as most of the random lookups occur in DRAM. Thus, we see an overhead of 1.7× for placing the leaves in PMem on B-256. The Apache servers

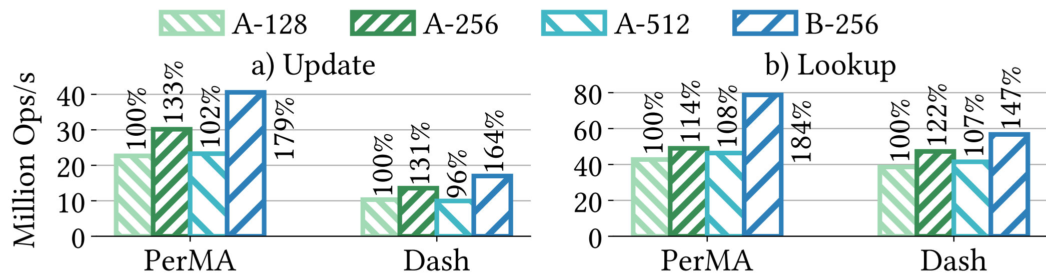

Figure 3.8: Hash index in PerMA and Dash. *16 threads.*
PerMA: 512 Byte lookup + 2× 64 Byte Cache update. Dash: 8/8 Byte key/value.

have a higher overhead, as their random read PMem bandwidth is lower and their DRAM is generally slower.

Overall, we see that random access patterns in PMem index structures perform significantly worse than in DRAM. Especially with small random writes, performance drops compared to PMem. We also note that these patterns are not optimized for DRAM, meaning that without the explicit flushes, even higher performance is observed. In the case of the hash index, we observe a 1.3× increase in DRAM when omitting the flush. But while PMem-based indexes cannot achieve the raw performance of DRAM, they offer (full) persistence and recovery with an average performance drop of only 4×. In addition, the price per GB of PMem is 4 − 6× lower than DRAM, striking a balance in price-performance.

Hash Index

In this section, we compare the PMem-aware hash index Dash [101] and the corresponding operations' memory access pattern in PerMA-Bench. We prefill Dash with 100 million entries before performing 100 million operations using the benchmark tool provided by the authors. The results are shown in Figure 3.8.

PerMA-Bench provides a good upper bound estimate of performance based on solely on memory access. For all Apache servers, the relative performance in raw access transfers directly to the relative performance of Dash. The insert performance is more complex in Dash, as it includes regular inserts, displacement, overflow buckets, and resizing. As these depend heavily on the implementation, it is not possible to model all in one custom operation. For our comparison, we assume an idealized insert without resizing and displacement and, thus, overestimate the insert performance. Developers can model all operations individually and run the benchmarks separately. This provides a good overview of the individual operations'

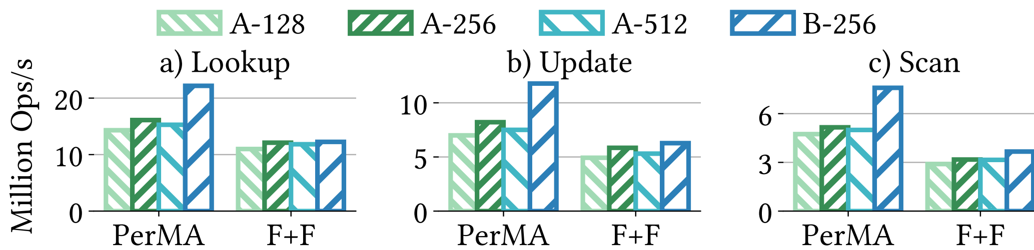

Figure 3.9: Tree index in PerMA and FAST+FAIR. *16 threads.*

performance and the results can be combined to determine the overall performance depending on the configuration parameters of the desired implementation. If a displacement takes X μs, depending on the ratio of inserts to displacements, e.g., 3:1, we can add $X/3$ μs to each insert to combine both operations.

PerMA-Bench also provides insight into potential areas of improvement. Dash achieves close-to-raw performance for lookups, which indicates that there is not much room for optimization in designs that access at most two buckets to retrieve an entry.

Finally, we observe that Dash underperforms on B-256. Unlike the 100 Series servers, Dash's relative raw lookup performance in PerMA-Bench is 40% higher than the actually achieved throughput. When investigating the performance in more detail, we see that Dash spends nearly 20% of all cycles on *machine clears* caused by memory ordering violations [59], which do not occur on the Apache servers. While we use Dash in this experiment, this problem is not Dash-specific but a general issue in current systems designed for PMem. Due to the high price, researchers often have access to only one server, resulting in current research focusing on a single configuration during development. Our results show that its performance is not yet understood well enough to generalize from one server to all, especially across generations. Now that the second generation Optane DIMMs are available, it is beneficial to consider more than one server to develop more general PMem-aware solutions in the future. In the following section, we show that performance limitations occur also in other index structures.

Tree Index

In Figure 3.9, we show the FAST+FAIR BTree implementation [58] and its modeled memory access in PerMA-Bench. FAST+FAIR is a popular PMem-only BTree implementation that was designed pre-Optane. We prefill 100 million records before performing 100 million operations using the benchmark tool provided by the

authors. The ideal-insert assumption as in Dash also applies to this benchmark. Across all operations, we see that the raw performance of B-256 is significantly higher than that of the 100 Series servers. However, this does not translate to FAST+FAIR, as its performance improves only marginally across generations.

As it was designed pre-Optane, it does not include various optimizations made in later designs. When taking a closer look at the execution, we see that 30% of all cycles are consumed by bad speculations, front-end stalls, and computation. These 30% are reflected in the performance difference between PerMA-Bench and FAST+FAIR. Compared to more recent work, FAST+FAIR also makes use of heavy-weight locking instead of atomics or hardware transactional memory. This overhead prevents FAST+FAIR from scaling with the higher performance of the newer Optane DIMMs.

These results indicate that general implementations without explicit knowledge of the underlying PMem technology do not scale well with better hardware. The memory access in FAST+FAIR is not optimized towards Optane, e.g., by requiring many flushes for updates due to sorted nodes. In another experiment, we observe better scaling results for the pre-Optane FPTree [123], as we used a version that was re-implemented more recently on Optane [52]. Adding to our insights on the Optane-tuned hash index Dash in Section 3.3.4, we conclude that it is also not viable to rely only on general PMem assumptions, as done in pre-Optane designs. It is beneficial to tune PMem-aware systems across a range of current hardware to capture the intricacies of Optane without optimizing solely for one server configuration. Especially with increasing PMem performance as in B-256, bottlenecks may shift from PMem access to, e.g., CPU or DRAM, requiring a balance between them.

Impact of eADR

In 200 Series Optane, eADR guarantees the persistence of data that resides in the cache, thus, making explicit flushes unnecessary. In this section, we evaluate the impact of omitting explicit flushes and issuing only `sfence` instructions[7] on various PMem-aware key-value storage designs. In this evaluation, we also include LB+Tree [97], a hybrid DRAM-PMem B+Tree optimized highly and explicitly for Optane, and Viper (Chapter 4 and [14]), a hybrid DRAM-PMem log+index key-value store that is designed for operations on larger items than 8 Byte index entries. This gives us a broader overview of PMem storage designs. We show the results of 32 and 64 thread runs on B-256 for FPTree, LB+Tree, Dash, and Viper in Figure 3.10.

7 `sfence` is still required to ensure correct ordering.

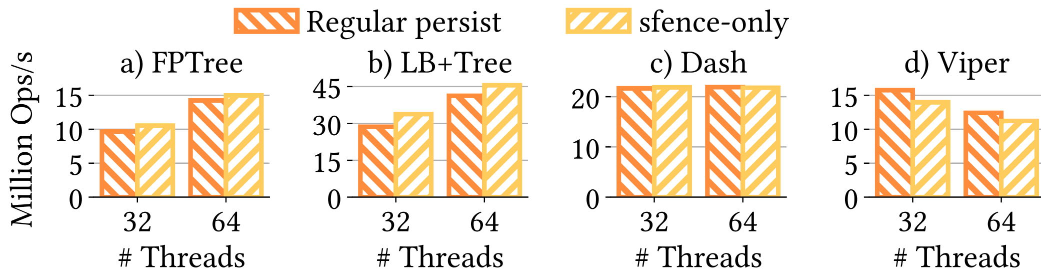

Figure 3.10: Impact of eADR on write performance of different PMem key-value storage designs *(Barlow-256). 32/64 threads.*

For the three index structures, we use 8/8 Byte key/values. For Viper, we use 16/200 Byte key/values. We evaluate FPTree with pibench [90]. For LB+Tree, Dash, and Viper we use the respective benchmark tools provided by the authors.

Our results show that for the tree-based designs, removing explicit flushing improves performance. However, for Dash, we observe no improvement and even a slight decrease for 64 threads. When storing larger records in Viper, we observe that not explicitly persisting reduces performance by 10%. Viper is designed to leverage sequential PMem writes, which are lost through random cache line eviction when omitting flushes (cf. Section 3.3.2).

Our results show that eADR is not a silver bullet for future PMem-system design. Developers must still understand their access patterns and evaluate whether they benefit from explicit flushes or not. Based on this, we encourage future work that explicitly compares low-level flush performance in PMem index/storage designs to provide an overview of benefits and downsides in this space.

3.3.5 Single Server Performance

In this section, we investigate configurations and settings that impact a single server. First, we show the impact of Intel's hardware prefetchers on memory bandwidth and discuss the implications this has for general PMem-aware system design (Section 3.3.5.) Next, to provide insight into the performance of partially stocked servers with older or lower-end components, we evaluate the performance of a single server with varying configurations. For economical reasons, users may not always choose fully stocked servers (one DIMM per available slot) with the highest configuration and latest components. To this end, we investigate the impact of the memory bus speed on PMem bandwidth (Section 3.3.5), followed by the impact of the number of DIMMs (Section 3.3.5).

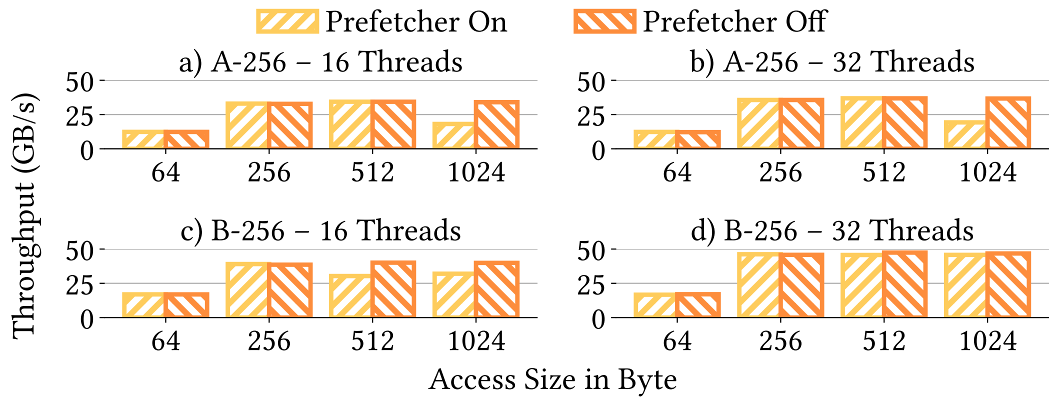

Figure 3.11: Impact of prefetcher on random read bandwidth.

Prefetcher

Throughout our evaluation, we observe workloads for which PerMA-Bench achieves lower performance than expected due to Intel's hardware prefetching behavior. This observation has been made in previous work [31] and we investigate it further in Figure 3.11. We actively disable all hardware prefetchers and measure the bandwidth utilization of 200 million random reads across 10 GiB with varying sizes. In the top row, we see that Apache-256 performs worse when the prefetcher is active for 1024 Byte reads with both 16 (a) and 32 threads (b). On the other hand, Barlow-256 performs worse for 512 and 1024 Byte but only with 16 threads (c). With 32 threads, the prefetcher impacts the performance only marginally (d). For the impacted runs, we observe higher bandwidth utilization in VTune, which indicates that the prefetcher is mistakenly fetching unnecessary data and thus reducing the effective bandwidth.

To transfer these insights to a real system, we run a micro-benchmark on Barlow-256 with the key-value store Viper. In this, we observe that disabling the prefetcher for 200 Byte values results in a 40% performance increase for get requests with 32 threads. But for 64 threads, a disabled prefetcher reduces performance by 30%. So while it is not generally advisable to disable the hardware prefetchers, its impact should be taken into account when designing, profiling, and optimizing systems that operate on larger data chunks, e.g., buffer managers or storage engines.

Memory Bus Speed

Optane modules share the memory bus with regular DRAM DIMMs, so they must run at the same memory bus speed. While DRAM often supports higher speeds

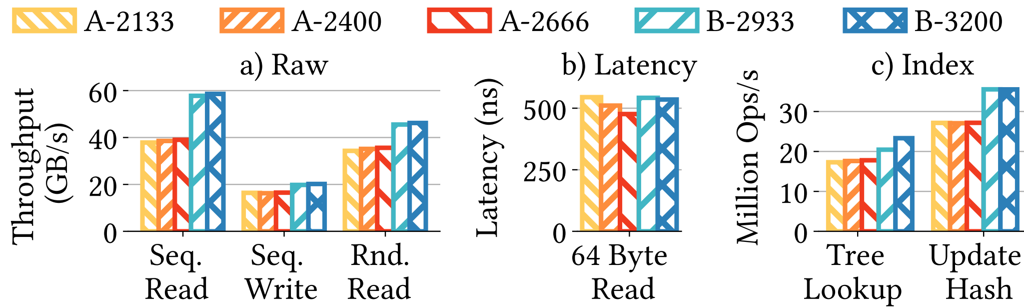

Figure 3.12: Performance-impact of varying memory bus speeds (in MT/s) on *Apache-256* and *Barlow-256. 32 threads.*

than Optane, this is not always the case, e.g., when using older DRAM modules, requiring users to reduce the speed of their PMem. To investigate which impact this has on PMem performance, we configure the memory bus in Apache-256 and Barlow-256 to different speeds. In Figure 3.12, we show the performance of sequential reads and writes, random reads, as well as read latency, and custom hash and tree index operation throughput. We use the same configurations as in the previous benchmarks. Apache-256's DRAM supports up to 2933 MT/s but is limited by PMem at 2666 MT/s, which we choose as the baseline. We also configure the bus speed to 2400 and 2133 MT/s to artificially slow down the server. Barlow-256's DRAM and PMem both support 3200 MT/s and we compare this to 2933 MT/s[8].

Our results show that PMem read bandwidth is impacted only marginally by reduced memory speed and not at all for write bandwidth. With a bus speed of 2666 MT/s, the theoretical bandwidth limit is ~20 GiB/s (= $2666 \times 10^6 \times 8$ Byte). In a server with six DIMMs, this allows for a theoretical maximum of ~120 GiB/s. Reducing the bus speed to 2133 MT/s results in a limit of ~16 GiB/s per DIMM and ~96 GiB/s across all DIMMs, i.e., a drop of 20%. However, PMem cannot supply data at this rate and stays significantly below the limit. The marginal difference in performance across the configurations is a result of slightly increased access latency due to fewer transfers per second. For DRAM, on the other hand, we observe a 20% bandwidth drop as it can provide data at the maximum frequency. Overall, we observe little to no performance drop and conclude that the selected memory bus speed is negligible for current Optane PMem. This also holds for 200 Series Optane, where it may be more common to have DRAM that limits the bus speed, as 3200 MT/s is also the current speed supported by DRAM.

8 The server's BIOS does not allow configurations below 2933 MT/s.

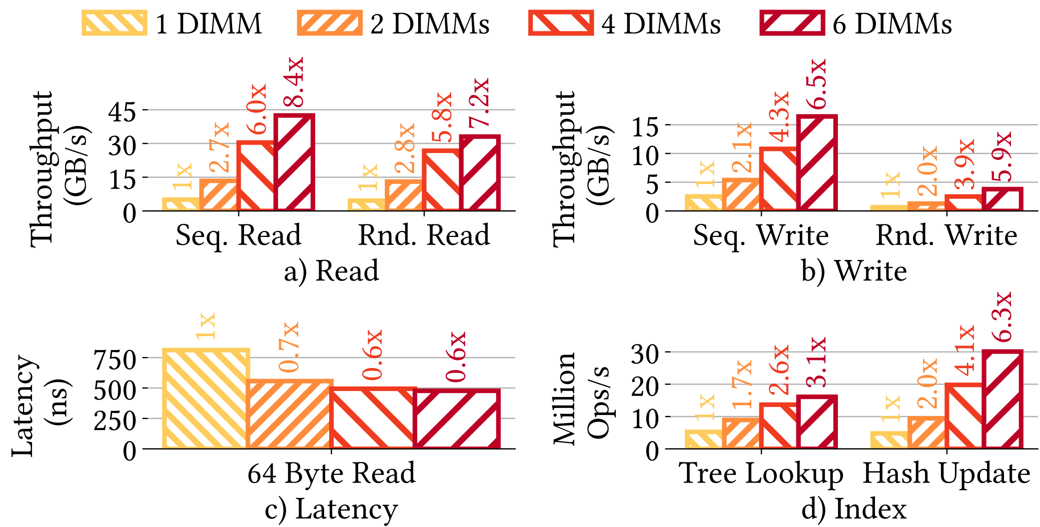

Figure 3.13: Impact of number of DIMMs in the server *(Apache-256)*. 16 threads. Sequential/random reads with 4096/256 Byte access size. Sequential/random writes with 512/64 Byte access size and *NoCache*. Tree with 512 Byte nodes. Hash with 256 Byte buckets.

Number of DIMMs

To provide insight into the performance of servers with only partially filled PMem slots, we evaluate PMem with a varying number of DIMMs. This allows us to both draw conclusions about how to stock a PMem server and also to make results of recent studies [82, 97, 144], which ran experiments on partially stocked servers, comparable to a full server. All experiments are run on a single socket of Apache-256 with 16 threads, we do not measure cross-socket performance. We run the experiment with all supported configurations, i.e., with 1/2/4/6 DIMMs. We physically remove the unused PMem DIMMs but keep all six DRAM DIMMs, following the official memory population guide [37]. In Figure 3.13, we show the bandwidth utilization of sequential and random read and write workloads, as well as random read latency and operations per second for tree index lookups and hash index updates.

In Figures 3.13a and b, we show the absolute bandwidth and relative improvement over the *1 DIMM* configuration. For reads and writes, we observe two different patterns. For write bandwidth, we observe a near-perfect linear scale. Each DIMM is fully saturated and constitutes a bottleneck. By adding more DIMMs, we evenly distribute the load across all available DIMMs until we have reached the maximum bandwidth. For sequential writes, we see a slightly super-linear scale. With fewer

DIMMs, the load on the individual DIMMs is higher and the write combining buffers receive more requests. They cannot combine adjacent stores as efficiently, resulting in increasing write amplification. When using 32 threads, this effect is even stronger, as the buffers are overloaded with requests. In this case, we observe an 11.9× increase from one to six DIMMs.

Read bandwidth utilization shows super-linear scaling from one to two and from two to four DIMMs. On currently supported CPUs, configuring a server with fewer than six PMem DIMMs results in an *unbalanced* configuration. While these unbalanced setups are supported, they are not recommended [121]. Memory controllers cannot optimize the memory layout and must create multiple interleave sets, resulting in worse performance. Additionally, the server must run in single-channel mode when using a single DIMM, which further reduces performance. For sequential reads with six DIMMs, PMem achieves 42 GiB/s or 7 GiB/s per DIMM. With 1 DIMM, it achieves only 5 GiB/s, which is ~30% worse.

Read latency decreases with more DIMMs, as the contention on each is reduced. Once the load is distributed, latency is not affected as much. This translates directly to the tree and hash index operations shown in Figure 3.13d. The tree lookup is impacted more by latency, so its performance does not improve as much as raw bandwidth. The hash index updates reflect the scaling of random writes with an influence of increased preceding read bandwidth.

Overall, we see a predictable performance pattern when using more than one DIMM, which allows us to transfer the results of previous work by approximately scaling the used number of DIMMs to six. Our results show that Optane PMem should be configured fully stocked and *balanced* to achieve maximum performance. However, if this is not possible, prefer multiple smaller DIMMs, e.g., 4× 128 GB = 512 GB, over fewer larger ones, e.g., 1× 512 GB to achieve a *near-balanced* configuration and reduce the load on the individual DIMMs, which otherwise quickly become over-saturated.

3.4 Server Price-Performance

A major selling point of Intel Optane is its lower price for higher capacity than DRAM. However, there is very little actual price-performance analysis in existing research. To provide insight into this, we perform a price-performance comparison across all evaluated servers. As major cloud vendors do yet not offer PMem, we base our analysis on the price as listed by Dell when configuring a server with Optane [135]. We note that the actual price of PMem differs slightly depending

Table 3.2: PMem price-performance comparison in Euro (€). See Table 3.1 for server info.

	Apache-128	Apache-256	Apache-512	Barlow-256	B-DRAM
€ per DIMM	1180	2750	8500	3270	1900
€/System *(GB capacity)*	7080 *(768)*	16500 *(1536)*	51000 *(3072)*	26160 *(2048)*	15200 *(256)*
€/GB capacity	**9.21**	10.74	16.60	12.77	59.37
€/GB/s seq. read	0.25	0.25	0.56	**0.22**	0.38
€/GB/s rnd. read	0.34	**0.33**	0.61	**0.33**	0.46
€/GB/s seq. write	0.78	0.64	1.52	**0.60**	0.70
€/GB/s rnd. write	3.43	2.78	6.18	2.12	**0.91**
€/100ns latency	**46.34**	52.38	83.94	64.49	80.61
€/update hash index	0.39	0.36	0.71	**0.27**	0.46
€/lookup tree index	0.56	0.60	0.96	**0.42**	0.84

on the source, country, and currency[9], but the relative difference between them is consistent. As such, our focus is not on the exact monetary values but rather on the relative difference between the servers. We base the performance-related values on the price per GB to explicitly exclude the price of higher capacity. As the Apache-128 server does not support 18 Watt, a comparison against the 18 Watt Apache-256, which supports this improvement, does not yield unfair results. However, an 18 W 512 GB server may achieve better price-performance than in our evaluation. To the best of our knowledge, this is the first extensive price-performance comparison of PMem across various configurations.

We show the price-performance results in Table 3.2. We first compare the PMem servers and then draw an overall comparison to DRAM. The price per GB capacity increases with the DIMM size, resulting in an up to ~80% difference within the 100 Series. Across generations, i.e., Apache-256 to Barlow-256, the price per GB increases by ~20%. The price for sequential and random read throughput differs

9 We checked dell.de, dell.com, and hpe.com in February 2022 and December 2021.

only slightly between the various PMem servers. However, Apache-512 is an outlier, as it offers the lowest performance (cf. Section 3.3.2) but the highest price, even when considering a 20% performance improvement through an 18 W configuration.

We see a wider range in the price-performance for both sequential and random writes. They differ by up to 1.5× and 2.9×, respectively. Due to significantly higher write bandwidth (cf. Section 3.3.2), it becomes apparent why both Apache-256 and Barlow-256 achieve up to 25/40% lower prices than the cheapest server (Apache-128) for sequential/random writes.

The price difference for hash index updates (normalized to 1 million operations) is a mix of the random read and write prices. Due to the low random read variance, the price variance across servers is not as significant as for pure random writes. The normalized 1 million tree lookups represent a random read workload and their relative price-performance ratio does not differ significantly from the random read ratios.

Regarding bandwidth and latency, DRAM outperforms PMem significantly (see Sections 3.3.2 and 3.3.3). However, DRAM's price per GB is up to 6.4× higher than PMem's. Our results show that PMem is competitive with DRAM in most of the raw access patterns, i.e., sequential/random reads and sequential writes. DRAM outperforms PMem for random writes, as PMem's bandwidth is significantly lower for such workloads. This read/write split also extends to more complex access in data structures, where DRAM outperforms PMem for write-intensive operations but offers little to no benefit for read-only access.

To optimize the price-performance of a server for a given workload, users have to choose which and how many DIMMs to buy. While this choice is heavily workload-dependent, our results show that users can use the following rule of thumb: *maximize the number of DIMMs for a target capacity*. If the workload fits into DRAM, there is no need for PMem, as this requires special CPUs and increases the overall cost. If the workload exceeds DRAM, users should use *n* DIMMs of the smallest size that offer the needed capacity, i.e., users should prefer 4× 128 GB over 2× 256 GB over 1× 512 GB. Our results in Section 3.3.5 show that the performance scales almost linearly, so while 256 GB DIMMs have a better individual performance than the 128 GB DIMMs, two 128 GB DIMMs outperform one 256 GB DIMM while providing the same capacity at a lower overall price. Thus, we suggest to use larger DIMMs only when the capacity is needed, as the price grows disproportionately higher for larger capacity.

3.5 Discussion

In this section, we present key takeaways from running PerMA-Bench and PMem-aware systems on various server configurations.

PMem Configurations. Our results show that the exact server configuration has a large impact on PMem performance. We identify four aspects that have not yet been studied in detail.

1) **DIMM size:** We show that the choice of DIMM size does not only impact capacity but also performance, especially as only the 256 and 512 GB DIMMs support higher power budgets.

2) **Power budget:** The 18 Watt power budget of Apache-256 improves write bandwidth by up to 40%, which is a major improvement considering PMem's otherwise limited write bandwidth. If possible (only for 256 and 512) and supported by the server, users should increase the power budget of their DIMMs.

3) **Number of DIMMs:** Varying the number of DIMMs has a predictable, close-to-linear impact on performance unless only a single DIMM is used. This causes an imbalanced memory configuration and a fallback to single-channel execution. For maximum performance, fully stocked servers should be chosen.

4) **Memory bus speed:** While DRAM and PMem must run with the same memory speed, we show that this does not impact PMem. The theoretical limits exceed PMem's performance, so users can reduce the speed if needed without losing performance.

Future PMem Research. We identify four additional aspects that impact PMem-aware implementations. With the increasing performance of PMem, previous bottlenecks may shift away from PMem to, e.g., the CPU, requiring more advanced and specialized implementations. As PMem is a new and evolving memory technology, a detailed understanding and optimization level known from DRAM must still be developed for it.

5) **Hardware utilization:** We observe that existing indexes do not fully utilize the performance improvements of the second generation. With more Optane configurations available, it is essential to tune future designs across a wider range of servers to achieve more stable performance.

6) **Persist instruction:** While the choice of persist instruction for random writes impacts bandwidth by only 30% in the 100 Series, it makes a difference of up to

2.5× in the 200 Series. Future work has to consider this and re-evaluate which choice of persist instruction is best-suited for different designs. Especially, now that the 200 Series allows for two different cache flushes, non-temporal stores, and no stores via eADR.

7) **eADR:** We show that omitting flushes due to eADR does not always yield the best performance. It remains important to understand when explicit flushes improve bandwidth utilization and latency, and when they do not.

8) **Prefetcher:** The prefetcher has an unexpected negative impact on certain workloads. While it should not be disabled, developers have to be aware that their system may be influenced by it.

Price-Performance. Within the first Optane generation, we identify 512 GB DIMMs to have the worst price-performance by a large margin. But overall, we show that PMem's price-performance is generally competitive with DRAM or even better. This allows PMem to be used as both explicit persistent memory or as cheaper and larger volatile memory, potentially even allowing for in-memory processing of workloads that previously did not fit into DRAM.

3.6 Related Work

In this section, we briefly discuss related work around PMem.

Persistent Memory Analysis. Various studies on the performance of PMem have been conducted. Earlier work focuses on performance assumptions and latency ranges to evaluate PMem in the context of various applications [7, 123, 141]. More recently, the performance of Intel's Optane DC Persistent Memory is investigated in more detail [19, 31, 47, 71, 138, 147]. These studies provide insight into the performance details of individual servers. In this work, we evaluate and compare the performance of PMem across various setups and show that this is needed to gain a better understanding of overall PMem behavior. These early benchmarks and existing tools such as `fio` [9] often run hard-coded queries or cannot represent complex access patterns and varying persist instructions, which are both essential to understand the performance of PMem for database components. To represent access patterns of current PMem systems, PerMA-Bench offers customizable, mixed PMem-DRAM pointer-chasing with locality-aware store instructions.

Persistent Memory Applications. The use of PMem is widely studied in index structures [27, 56, 87, 97, 101, 113, 123, 156], key-value stores [14, 26, 91], database systems [7, 8, 106, 122, 129], and filesystems [76, 117, 146]. We extract common

access patterns from this work and define the workloads in PerMA-Bench based on them.

3.7 Conclusion

In this chapter, we propose PerMA-Bench, a configurable benchmark framework that allows users to evaluate the bandwidth, latency, and operations per second for customizable database-related PMem access. We perform an extensive analysis across four PMem servers of the first and second Optane generation, with varying configuration options, such as DIMM power budget, memory bus speed, and number of DIMMs per server. We show which impact these configurations have on performance and raise awareness for the overall configuration space of PMem. We validate our results with existing implementations and show that they do not fully utilize the performance improvements across Optane generations. We show that the choice of persist instruction has a high performance impact and that avoiding explicit flushes in eADR does not always yield the best results. Finally, we perform a price-performance comparison across all evaluated servers. While there are great differences between Optane DIMMs, PMem is generally competitive with DRAM. This allows PMem to be used as both explicit *persistent* memory or cheaper and larger *volatile* memory.

PMem is still a new and evolving technology and research into PMem-aware databases is still in its infancy compared to DRAM. We present directions for future designs, implementations, and evaluation of PMem solutions that are needed to fully understand and utilize the hardware. We make our evaluation results available and with PerMA-Bench, we hope to lead the way to a common understanding of PMem performance by gathering and comparing various existing configurations and future PMem hardware.

4 Viper: An Efficient Hybrid PMem-DRAM Key-Value Store

The majority of this chapter has been published in [14].

4.1 Introduction

Persistent key-value stores (KVSs) have become a widely used alternative type of data store next to classical relational database management systems (RDBMSs). Different to RDBMSs, KVSs store schema-less data (*value*) retrievable through a given *key*. KVS workloads also differ from classical RDBMS workloads in that they are write-heavy and nearly exclusively operate on single records [91]. These workload characteristics allow for a variety of KVS applications, ranging from storage engines in SQL systems [32], over state-storage for stream processing engines [21, 150], to caches for web applications [128]. On a large scale, these use-cases all require high performance and strong persistence guarantees.

To ensure data persistence, current KVSs write their data to devices with a block-based interface, i.e., SSDs or HDDs. However, the emergence of persistent memory (PMem) promises byte-addressable data persistence with close-to-DRAM speed [44, 71, 138, 147]. Thus, leveraging PMem for KVSs and removing disk access has a large potential to improve KVS performance. It also supports the storage of arbitrary data structures without the need for record de-/serialization, which is required in traditional string-based KVSs.

To improve the performance of write-heavy workloads, most traditional persistent KVSs such as RocksDB [38] or LevelDB [43] optimize their inserts to avoid expensive write amplification on block-based devices. They employ log-structured trees [120] to collect records in-memory that are then written to disk in a single block-sized chunk. This approach requires additional disk-based write-ahead logging to ensure data persistence, as well as sophisticated merging logic for the disk-writes. Additionally, most disk-based KVSs require string or byte keys and values to store arbitrary data. This comes at a high de-/serialization cost for each access, significantly impacting the overall performance [40, 104].

Previous PMem research either focuses on how to adapt existing systems or develop new ones to harness PMem's potential. Various hybrid PMem-DRAM data structures have been proposed that leverage the speed of DRAM with the

persistence of PMem for better overall performance. Most research focuses on the design of index structures, e.g., B-Trees [123, 148], LSM-Trees [98], or hash maps [101, 114]. Other research integrates PMem into larger systems, e.g., for database buffer management or recovery [7, 129]. Some simulated-PMem KVSs have also been proposed [98, 145].

However, as PMem has only recently become publicly available, the majority of previous PMem research uses simulations to estimate PMem performance in which key characteristics were assumed incorrectly [147]. These incorrect assumptions limit the effectiveness of proposed solutions as the optimal utilization of PMem requires knowledge of the underlying storage access patterns and characteristics. Recent research shows that Intel's Optane DIMMs [69] behave differently than DRAM and SSD [31, 147]. Thus, simply replacing disk-based storage with an identical PMem-based one does not yield the best performance. Benchmarks also show that sequential write latency to PMem is much closer to DRAM's performance, whereas there is a higher penalty for random reads than expected [44, 147]. This breaks one main assumption previous research built upon, that writes are slow and should be avoided and reads are fast and can be random.

To overcome the central performance issues of disk-based KVSs and incorrect assumptions of previous PMem research, we propose three PMem-specific access patterns for efficient data storage, *direct PMem writes, DIMM-aligned storage segments,* and *uniform thread-to-DIMM distribution*. We implement these patterns in *Viper*, a hybrid PMem-DRAM KVS whose persistence is built on PMem, thus avoiding expensive disk accesses. Viper consists of a volatile index and persistent data, to perform most of the random operations in fast DRAM while optimizing the storage layout for efficient writes to PMem. In summary, we make the following contributions:

1) We propose PMem-specific access patterns to efficiently store and retrieve data directly to and from PMem in a hybrid PMem-DRAM environment.

2) We implement these access patterns in Viper, a hybrid PMem-DRAM KVS that persists its data directly in PMem.

3) We evaluate Viper against state-of-the-art KVSs and show that it outperforms them for core KVS operations. Viper exceeds existing PMem-only, hybrid, and disk-based KVSs by 4–18x for write workloads, while matching or surpassing their *get* performance.

The remainder of this chapter is structured as follows. In Section 4.2 we cover some technical background on key-value stores. In Section 4.3 we introduce Viper

and its core design principles. We show Viper's core functionality in Section 4.4, followed by a detailed evaluation in Section 4.5. We end this chapter with an overview of related work in Section 4.6 and our conclusion in Section 4.7.

4.2 Background

In this section, we briefly introduce key-value store terminology and concepts as used in this chapter. Key-value stores (KVSs) are a class of storage systems that handle data as ⟨ key, value ⟩ pairs. The basic operations KVSs implement are *put*, *get*, *delete*, and optionally *update* [24, 38, 91]. To access KVSs, two designs have emerged, *KVS servers* and *embedded KVSs*. A server-based KVS stores and synchronizes state that can be globally accessed by multiple applications running on different machines. It communicates with the applications via a network client/server API. Popular KVS servers are *Redis* [128] and *memcached* [109]. If the KVS is used by a single application, embedded KVSs provide a more lightweight alternative to server-based ones. These KVSs are embedded in the application and accessed using library function calls. Popular embedded KVS are RocksDB [38] and FASTER [24].

A main advantage of KVS servers is that they are self-contained systems. This provides them with full system control, i.e., among others, they manage their own threads, concurrency, and I/O queues. However, this control entails an abstraction cost via, e.g., a network-based interface. On the other hand, embedded KVSs are controlled by the user within an application, which results in less communication overhead compared to network-based access and allows more fine-tuning. Yet, this control comes at the risk of incorrect usage, which might impact correctness and performance. To provide good performance and control, embedded KVS must design their interfaces as simple as possible without requiring the user to strictly follow patterns or complex procedures in case of, e.g., partial failures or system restarts. In this work, we focus on the design of such an embedded KVS and simple interface to allow the user to fully utilize PMem without high network overhead. We present such a design and implementation in Viper.

4.3 Viper: A Hybrid Key-Value Store

In this section, we present *Viper*, a hybrid PMem-DRAM KVS that leverages PMem-specific access patterns for efficient data storage and retrieval. Viper avoids expensive disk access by persisting data in PMem while keeping an in-memory index to harness DRAM's lower random access latency over a fully PMem-based approach.

We first discuss our hybrid design in Viper in Section 4.3.1 followed by a description of Viper's core components in Section 4.3.2.

4.3.1 Hybrid Design

To fully utilize both DRAM's and PMem's strengths, we propose a hybrid storage approach in Viper. Viper consists of a volatile hash index located in DRAM and persistent data blocks located in PMem. While Optane DIMMs can act as a drop-in replacement for SSDs to achieve data persistence, to fully leverage the performance of PMem, we need to understand its storage layout and beneficial access patterns. All data is durably stored in persistent memory and the hash index contains only references to the storage location.

Hybrid storage models have also been proposed in previous work on index structures [123, 148] with the concept of *selective persistence*. The idea behind selective persistence is to store only the data required to rebuild the entire system state in persistent memory and keep a dynamic recoverable state in volatile memory. Viper is designed to be an embedded KVS similar to RocksDB [38] or FASTER [24] and not a KVS server. Thus, users interact directly with the database in the same process without any network interface.

PMem Access Patterns. Initial studies on real PMem show complex performance characteristics, which often lead to low bandwidth and high latency [71, 147] In Viper, we propose three core design choices for PMem-specific access patterns that significantly impact its performance on real hardware:

1) **Direct PMem writes.** As sequential PMem writes are faster than previously assumed in simulations, Viper writes all data directly to PMem without an intermediate DRAM buffer.

2) **Uniform thread-to-DIMM distribution.** Viper minimizes the thread-to-DIMM ratio for inserts by assigning threads to different memory regions.

3) **DIMM-aligned storage segments.** Viper stores data in DIMM-boundary aligned pages (*VPages*) to balance DIMM contention with parallelism. Smaller pages result in more threads accessing the same DIMM and larger pages result in a single thread accessing multiple DIMMs, both leading to a worse, and thus disadvantageous, thread-to-DIMM ratio [147].

We demonstrate the impact of these design choices in Figure 4.1 (see Section 4.5.1 for our system setup). We perform 64 Byte stores followed by `clwb` and `sfence` with a varying number of threads in PMem and DRAM. Figures 4.1a and b show that

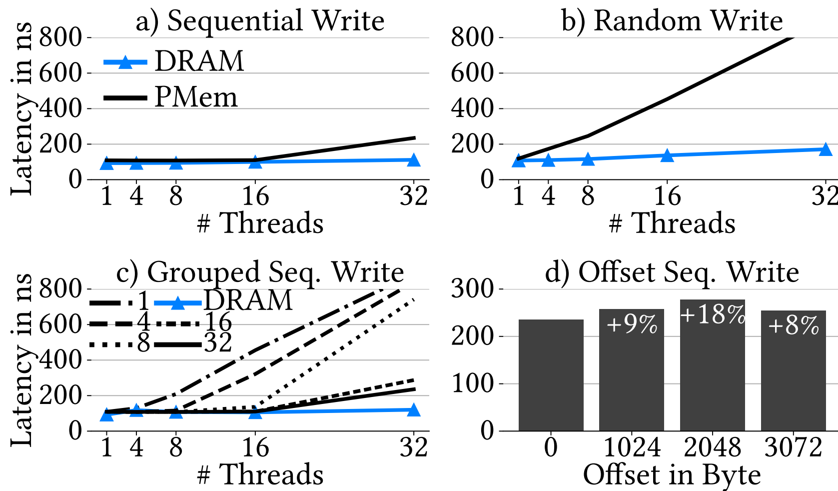

Figure 4.1: Write latency for various write patterns to DRAM and PMem.

sequential writes have a similar latency for PMem and DRAM (maximum 2x higher for 32 threads), while random writes perform significantly worse on PMem even for low thread counts. This is due to Optane's internal write-combining buffer, which combines adjacent writes to reduce expensive media flushes but cannot combine small random writes, causing high write amplification. From this observation, we derive our *direct PMem writes* design.

Figure 4.1c shows the importance of an even distribution of threads across all DIMMs. We distribute the threads across k memory regions (1 GB each), representing log files, to which they write sequentially. Using 1 log file (denoted as 1 in the plot), all threads write adjacent cache lines, i.e., thread 1 writes bytes 0–63, thread 2 writes 64–127, and so on. When using the same number of threads and logs, each thread has its own disjoint memory region. With more logs, fewer threads share a memory region and evenly distribute across the DIMMs. The poor performance of 1 log is caused by all threads operating on a single DIMM (32 × 64 Byte = 2048 Byte) and thus, disregarding the inherent parallelism of interleaved PMem. We see a performance increase when using more logs as the threads profit from PMem's parallelism by writing to varying locations evenly distributed across DIMMs. From this observation, we derive our *uniform thread-to-DIMM distribution* design.

Finally, Figure 4.1d shows the impact of storage-aligned access. In this benchmark, we let each thread write 4 KB sequentially and alter the alignment of the writes. We see that 4 KB aligned writes (offset = 0) achieve the lowest latency, while a 2 KB offset has an 18% higher latency. This is again caused by the necessity of accessing

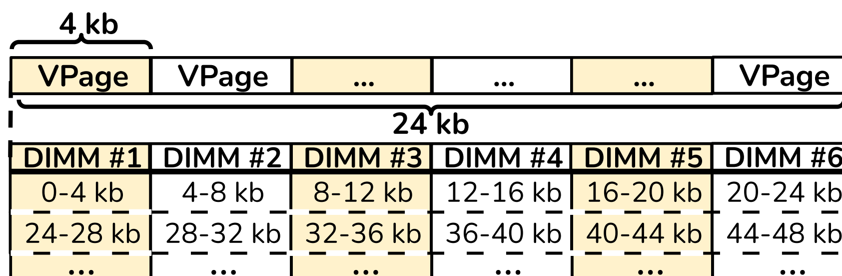

Figure 4.2: Viper's storage aligned with 4 KB PMem layout.

two DIMMs to write 4 KB instead of only one. From this observation, we derive our *DIMM-aligned storage segments* design.

Volatile Index. Our evaluation of real PMem hardware shows that random operations have a significantly higher latency than sequential ones and achieve lower bandwidth (see Figure 4.1 and Chapter 3.3.2). Thus, we avoid (possibly multiple) expensive random operations to the hash index by locating it in DRAM. Additionally, the efficient design and implementation of hash maps in DRAM are widely studied [85, 94, 105, 133], allowing us to fully take advantage of these concepts. Persistent hash maps, on the other hand, have only recently been introduced [101, 113, 114, 131] and show lower performance than DRAM-based ones. Furthermore, due to the persistence of every operation in the map, complex logic is required to avoid concurrency and memory issues, e.g., persistent memory leaks, invalid pointers, and blocked persistent locks. For our implementation of Viper, we build on CCEH [113] and use it in DRAM instead of PMem. CCEH uses an extendible hashing approach, thus allowing for dynamic resizing without an expensive full table rehashing. As we use the volatile index to store offsets to PMem locations, we refer to it as *Offset Map* in the remainder of this work.

Persistent Data. As our goal is to persist all data in Viper, we need to store all key-value pairs on a durable storage medium. We choose Intel's *Optane DC Persistent Memory* [69] in our implementation. In Viper, we write all records directly to PMem-based storage segments (design choice 1). Viper's main storage segments are called *VPages* and contain the individual key-value records as well as some metadata. Figure 4.2 shows how we align VPages with the layout of the underlying PMem DIMMs (design choice 3). We assume a system configuration with six DIMMs per socket. However, Viper is configurable to work on any number of DIMMs. We use Optane DIMMs in the interleaved mode to achieve a higher degree of parallelism [147]. In the interleaved mode, data is striped across all DIMMs in 4 KB pages. We exploit this striping by aligning VPages to the 4 KB page boundaries.

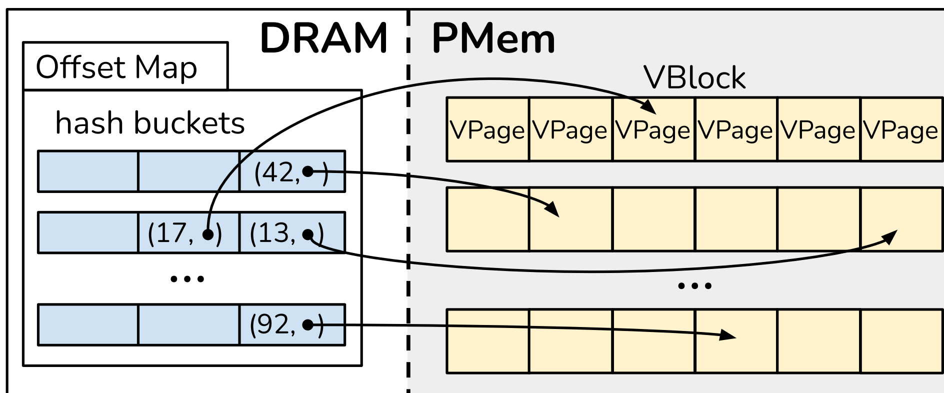

Figure 4.3: Viper's architecture. VPages store key-value records in PMem (right). The Offset Map stores (key, record-offset) entries in a volatile hash index (left).

This allows us to access exactly one DIMM per VPage, thus reducing contention on the DIMMs during parallel access (design choice 2).

4.3.2 Architecture

Viper consists of three main components, persistent *VBlocks* and *VPages*, as well as an in-memory *Offset Map*. We show Viper's core components in Figure 4.3. On the right-hand side, we see Viper's persistent storage segments (VPage) grouped into VBlocks, located in PMem. On the left-hand side, we see Viper's volatile Offset Map, which acts as an index by storing the key and persistent storage location of each record. In the remainder of this section, we describe the design of the three core components in detail for fixed- and variable-sized records. We first describe the VBlock and Offset Map, as these are identical for both variations, followed by the fixed-sized VPage design and the variable-sized modifications.

Common Components

In this section, we present components that are identical for fixed- and variable-sized records: the *VBlock* and *Offset Map*, as well as Viper's metadata management.

VBlock. In Viper, we align VBlocks to the boundaries of the underlying interleaved set of DIMMs, spanning exactly 24 KB. Each VBlock contains a fixed number of VPages, one VPage for each DIMM, stored in an in-place array for efficient access. VBlocks contain no logic themselves but simply act as a grouping of VPages to reduce the bookkeeping overhead in Viper. Each VPage is 4 KB (DIMM-aligned) and contains some metadata plus the actual key-value records stored in *slots*. They

are the actual storage units in Viper. To support larger key-value pairs, Viper scales VPages to multiples of 4 KB and VBlocks to multiples of 24 KB, ensuring the same 4 and 24 KB alignment. For simplicity, we assume 4 KB VPages and 24 KB VBlocks in the remainder of this work.

Offset Map. The *Offset Map* is the core volatile index that Viper uses to keep track of all records. In Viper, the Offset Map is an in-memory, concurrent hash map. When a record is inserted into Viper, it is first persisted in a VPage and then the *offset* of the record is stored as the value in the Offset Map for the given key. The *offset* consists of three parts: the VBlock id, the VPage id, and the record position in the VPage. The record position depends on fixed- or variable-sized records. With these three parts, Viper can uniquely locate any given record. Viper stores the offset in a 64-bit `Offset` object, where the most significant 45 bits represent the block id, the following 3 bit represent the page id, and the next 16 bit are used for the record position. The bit-assignments may be modified in case the user has specific knowledge of the expected workload, e.g., very large records or the number of DIMMs varies significantly.

Analogously to previous work, we use *fingerprinting* in order to store keys larger than 8 Byte in the Offset Map [101, 123]. Instead of storing the actual key in the map, Viper stores the hash of that key and checks for equality only if the hash matches. This significantly reduces the number of expensive comparisons with the keys in PMem, as very few collisions are expected for 64-bit hashes.

Metadata Management. To grow, Viper allocates VBlocks in PMem and maps them into the virtual memory space via `mmap` [112]. To keep track of the virtual addresses, Viper stores a pointer to each VBlock in a list in DRAM. This allows for easy access to an arbitrary VBlock by its implicit id, which is equal to the offset in the list. Once the available VBlocks reach a certain configurable filling degree, Viper allocates additional VBlocks and adds them to the list. Viper supports PMem allocation from both *devdax* or an *fsdax* directory. Data is allocated in increasing memory order (devdax) or increasing file names (fsdax) to guarantee ordering, thus maintaining the VBlock order after a restart. To reduce the number of memory allocations, large chunks (or files) are allocated, which contain 43690 VBlocks by default (1 GB). Metadata recovery and mapping all data back into Viper's virtual memory space takes only a few milliseconds, as it mainly consists of `mmap` calls.

Fixed-Sized Records

We now present the VPage design for fixed-sized records, as shown in Figure 4.4 (a).

VPage Data. Viper stores the actual key-value records in VPages. Both the key and the value are stored together in a single *slot*. The slot id is used as the third

byte

0	1	4	204	404	604	3804	4004 4096
lock	free slot bitset	$slot_0$	$slot_1$	$slot_2$	···	$slot_{19}$	pad
0	1111001...000	12|abc	51|xyz	13|def	···	|	

(a) 200 Byte fixed-sized records.

byte

0	1	9	10	249	309 4096
lock	next insert pos	deleted	data		
0	ptr to 300	0	size: 8|227 rec: k1|ab	size: 4|52 rec: k2|x	

(b) Variable-sized records.

Figure 4.4: VPage layout with example entries. Key-value records are stored consecutively.

part of the Offset Map entry (*record position*) for fixed-sized records. When using the term key-value *record*, we refer to both the key and value together. The number of slots per VPage depends on the record size, where larger records require more space and thus fewer fit into the available 4 KB. Viper uses nearly all of the 4 KB to store data, as only a few bytes are needed for metadata. We describe the calculation for the number of slots with the metadata size below.

VPage Metadata. The metadata is stored in the first few bytes of the VPage. It consists of a version lock byte and a bitset indicating which slots are free or populated. Both concepts are also used in previous research on PMem data structures, e.g., in tree nodes [25, 123] or in hash buckets [101]. We use a lock byte to handle concurrent access to the VPage, allowing only one thread to concurrently modify its data. The lock is acquired and released via atomic compare-and-swap operations (CAS). We thus avoid the use of heavy-weight mutexes at this point. Even though there are persistent CAS implementations [143] that ensure correct persistence-semantics, Viper uses regular in-memory CAS operations with less overhead. The lock is only relevant during active use and is reset after a crash.

The bitset contains k bits, one for each slot in the VPage. A set bit indicates that the slot is occupied and contains data. An unset bit, in reverse, indicates that the slot is free. This allows Viper to efficiently delete a record by setting the bit at its slot position to 0.

VPage Slot Count and Metadata Size. The exact size of the metadata depends on the record size, as Viper requires one bit per slot in the bitset. To determine the metadata size, we first calculate the number of slots per page by dividing the VPage size by the record size, $\lfloor size_p / size_r \rfloor = num_{slots}$. This is rounded down to the nearest integer as we cannot have partial slots. To avoid an over-allocation of the

VPage, we need to check if the metadata still fits. The metadata size is calculated as $1 + \lceil num_{slots}/8 \rceil = size_m$ bytes, for the lock + bitset. We round up the bitset size, as the underlying system cannot work on individual bits but requires full bytes. If the data plus metadata is too large for the VPage ($num_{slots} * size_r + size_m > size_p$), we reduce the number of slots by one. All unused space at the end of the VPage is left as padding to keep the 4 KB alignment.

Variable-Sized Records

To support variable-sized records, the VPage-design needs to be modified, as shown in Figure 4.4 (b).

VPage Data. Records are not stored in fixed slots, as their size is unknown a priori. Thus, Viper uses all non-metadata bytes in the VPage as a log. Each record is consecutively written to this log together with the respective key and value length. The sizes are stored in a single 32-bit value (15 bits for key, 16 bits for value) to allow for atomic updates. The least significant bit of the value indicates whether the record is set (= 1) or deleted (= 0). The offset in the log is used as the third part of the Offset Map entry (*record position*) for variable-sized records. For key-value pairs larger than 4 KB, Viper dynamically uses an entire VBlock as a single VPage. For even larger records, Viper writes the record across multiple large VPages and marks these as *overflow* pages. Thus, large records do not impact the design of Viper, as it still has unique VBlocks per client and equal distribution of client threads to DIMMs.

VPage Metadata. As the VPage does not contain any slots, the free slot bitset is removed. Instead, each VPage now contains a pointer to its next insert position, i.e., the tail of the log, and an 8-bit integer to track how much data has approximately been deleted (i.e., metadata bit = 0) and needs to be compacted. The metadata size is fixed for variable-length records at 10 Byte, allowing for 4086 Bytes of records per VPage.

4.4 Key-Value Store Operations

In this section, we discuss the common KVS operations *put* (Sec. 4.4.2), *get* (Sec. 4.4.3), *update* (Sec. 4.4.4), and *delete* (Sec. 4.4.5), as well as space reclamation (Sec. 4.4.6) and the recovery of an existing database (Sec. 4.4.7). Before discussing the operations, we present the *Viper client* through which users interact with Viper (Sec. 4.4.1).

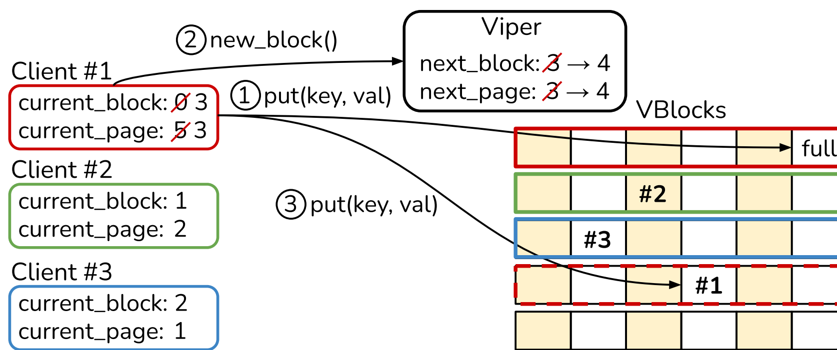

Figure 4.5: Client requests new VBlock. Client #1 requests a new VBlock after a *put* to a full one and then writes to the new VBlock.

4.4.1 Viper Client

Commonly in embedded KVSs, a database handle is created for a given file, which either creates a new database if the file does not exist or opens the existing database. This handle can be used by multiple threads to interact with the KVS, by issuing, e.g., *get* or *put* requests. However, when the KVS does not control or own the threads, the handle has to control external concurrent access. Examples for such embedded KVSs are RocksDB [38] or LevelDB [43]. For *put* requests, this means providing a new insert location for each request. This central synchronization point quickly becomes a bottleneck, which we avoid in Viper by introducing a *Viper client*.

As Viper is an embedded KVS, the client does not contain any network logic as common in KVS servers. It is a light-weight object that exposes the KVS-operation interface to the user and contains information on where to write future records to reduce synchronization within Viper. In Figure 4.5 we show clients interacting with Viper. In our example, three clients have been created. Each client is initialized with its own VBlock (#1 $\rightarrow$ VBlock_0, #2 $\rightarrow$ VBlock_1, #3 $\rightarrow$ VBlock_2), i.e., no two clients write new records to the same VBlock. To indicate that a VBlock is currently "owned" by a client, an owned_bit is set in the version lock of the first VPage. This bit is used for space reclamation and recovery. The client then writes data to its current VBlock/VPage and progresses the VPage until all VPages are full. Once a client cannot *put* data into its VBlock because it is full ①, it requests a new VBlock from Viper ②. Viper then returns the next block to the client and atomically updates the `next_block` and the `next_page` counters. Viper stores the `next_block` and `next_page` counters in a single 64-bit variable that can be updated atomically with a compare-and-swap operation. The `next_page` counter is chosen

randomly to achieve a uniform distribution across DIMMs. When the client receives its next block, it updates its references and inserts the record into the new VPage ③. This approach significantly reduces the coordination overhead within Viper, as it does not need to issue a new write location for each *put*. If a VBlock fits, e.g., 100 records, the overhead is reduced by 100x, as a client only needs a new location every 100 writes.

As Viper supports space reclamation (see Section 4.4.6), it also keeps track of free blocks in a concurrent queue. If a free block is present in the queue, it is given to a client for re-use rather than allocating a new client. In that case, the block is removed from the queue and the `next_block` counter remains unchanged.

4.4.2 Put

To insert data into Viper, clients must issue a `put(Key, Value)` request. The pseudo-code for this is presented in Listing 4.1. The client first acquires the VPage lock for its current VPage in a blocking call (Line 1). To acquire exclusive access, the version lock is atomically compare-and-swapped with a +1 increment to an odd-number, e.g., from 0 to 1. If a client encounters an odd-numbered lock, it retries its operation. Once the client has exclusive access to the VPage, it searches for the next free slot (Line 2). If the VPage is full, the client releases the lock, updates its page and block information, and retries the *put* operation (Lines 3-6). To update the page and block information, it either progresses to the next VPage in its current VBlock or it requests a new VBlock from Viper.

If there is a free slot in the current VPage, the client stores the record in the free slot and persists it (Lines 8-9). To write the current cache line to PMem, the `Persist` method issues a `clwb` call to the underlying system followed by an `sfence` call. The `sfence` enforces correct ordering guarantees, i.e., after the call, the data is guaranteed to be persisted. Only after the data is persisted does the client update and persist the bitset (Lines 10-11). The order here is important, as the bitset becomes the ground truth for recovery [101, 123]. If the bitset indicates a populated slot but the data is not properly stored, Viper is in an inconsistent state.

Once the data is persisted, the client inserts the new offset into the Offset Map. If the Offset Map contained an entry for the key, the old value is overwritten and the client must ensure that the record at the old location is deleted by setting the corresponding bit to 0 (Line 15, see Section 4.4.5). As the Offset Map handles concurrency, it guarantees that in the event of concurrent writes to the same key, one client will see the value added by the other client as an old offset, thus deleting the other client's value. Finally, the client releases the lock on the VPage and returns a Boolean indicating whether a new key was inserted or an existing one

Listing 4.1: Viper's `put(Key k, Value v)`

```
1 AcquireVPageLock(v_page);
2 free_slot_idx = FindFreeSlot(v_page.slot_bitset);
3 if (free_slot_idx == max_bitset_size) {
4   ReleaseLock(v_page); GetNewVPageOrVBlock();
5   return Put(k, v);
6 }
7
8 v_page.slots[free_slot_idx] = {k, v};
9 Persist(v_page.slots[free_slot_idx]);
10 v_page.slot_bitset[free_slot_idx] = 1;
11 Persist(v_page.slot_bitset);
12
13 offset = {block_num, page_num, free_slot_idx};
14 [is_new, old_offset] = offset_map.Insert(k, offset);
15 if (!is_new) DeleteOldRecord(old_offset);
16
17 ReleaseLock(v_page);
18 return is_new;
```

was overwritten (Lines 17-18). The lock is released by atomically storing another +1 increment, thus, making the lock even-numbered again.

Crash Consistency If a crash occurs between persisting the bitset and the deletion of an old record, Viper contains two values for the same key. To guarantee a deterministic recovery and thus ensure atomic writes, Viper selects the greater ⟨ block_id, page_id, slot_id ⟩ in case of a conflict. We note that this is not necessarily the newer value, as "old" block ids are reused after reclamation but it constitutes a deterministic tie-break during recovery. To ensure that the new value is not read until it is guaranteed to be deterministically recoverable, clients hold the VPage lock until the old record is deleted. In rare cases, this may lead to a deadlock, as two clients might need to lock the same two VPages in reverse order. If a deadlock is detected, i.e., the lock cannot be acquired in x tries, the client adds the offset it needs to delete O to a global list. All clients in the deadlock continuously check this list for offsets O' that match their current VPage, delete the record at O', and remove it from the list. If a client notices that O was deleted from the list, the deadlock is solved and it can return after unlocking its VPage. In a micro benchmark with 50 million mixed operations, Viper encounter only two such deadlock-like scenarios.

Variable-Sized Records. Inserting variable-sized records follows a similar procedure as shown in Listing 4.1, but the actual writing of the data is different. To insert a variable-sized record, the client first retrieves the `next_insert_position`

from the VPage metadata. It then writes the record to PMem at the given location followed by a `Persist` call. Only then does it write the record's metadata in front of the record. This order guarantees that if the metadata is present, the record is persistently stored. This is identical to persisting the bitset after the slot for fixed-sized records. When a record does not fit into a page, the client checks if the key without the value fits. If it fits, the value is written to the next page, and only then is the key written with metadata indicating a value length of 0, which tells Viper that the value is stored on the following page to ensure the same persistency guarantees as above. If the key does not fit, the record is written to the next page and the metadata is set to an invalid configuration on the current page, indicating that no more data is present after this marker. After inserting the record, the `next_insert_position` is updated to reflect either the end of the page or a new position.

4.4.3 Get

To retrieve individual records from Viper, the client issues a `get(Key)` request. To efficiently scale for read-heavy workloads, Viper uses lock-free reads [25, 101]. First, the client searches for the key in the Offset Map and returns an error if no entry was found. The client then atomically reads the version lock of the VPage that contains the record into $l1$. If $l1$ is odd-numbered, another client currently holds an exclusive lock and the entire read is retried, as a VPage modification might have altered the retrieved offset. In an unlocked state, the client reads the value at the given offset. The pointer retrieval is a lookup in Viper's VBlock list for the offset's block id, followed by direct accesses into that VBlock's page list at the page id and the VPage's slots at the slot id. We note here that the VPage array within a VBlock and the slots within a VPage are known at compile-time, thus allowing the compiler to combine the latter two lookups into simple pointer arithmetic on the VBlock pointer from the initial lookup. Before returning the value, it again atomically loads the version lock into $l2$. If $l1 = l2$, the VPage was not modified and the value can be safely returned. If $l1 \neq l2$, the entire read operation is retried, as a conflict might have occurred.

Retrieving variable-sized records follows the same steps, but the actual record lookup differs slightly. Instead of reading a record from a given slot, it first reads the record length at the given offset in the VPage log and then retrieves the value according to its size.

4.4.4 Update

In order to update a value in Viper, the user can call `update(Key, UpdateFn)`, where *UpdateFn* is an arbitrary function that receives a value and modifies it atomically. As Viper does not copy the values, modifications are made in-place in PMem. To avoid partial update anomalies, only atomic updates can be performed. However, this allows the user to modify up to 8 Byte (or 16/32/64 with modern AVX-512 CPUs) of a value in-place. This is useful to, e.g., update counters or other individual fields in the value [20]. Updating in Viper is similar to *get* but instead of returning the value if no version conflict occurred, the client acquires an exclusive lock and applies the *UpdateFn* to the value. Thus, any subsequent operations are aware that a modification was performed.

For non-atomic updates, the value must be re-inserted. To achieve this, the user *gets* the value, creates a copy, modifies it, and finally calls *insert* for the same key with the new value. This is a common approach in many KVSs [38, 43, 84, 102] and Viper always falls back to this approach if in-place modifications are not possible. This is also the approach for variable-sized records, as modifications in them might change their size. In Viper, records are tightly packed in the log and do not allow for any subsequent size variation.

Two main advantages of in-place updates over conventional copy-on-write are avoiding serialization and fewer cache line flushes, i.e., only one `Persist` call is needed in Viper as no metadata is updated. Also, recent work shows that in-place modification is preferred over copy-on-write for PMem [90, 130].

4.4.5 Delete

To delete a record, the client issues a `delete(Key)` request. The client first looks for the key in the Offset Map and returns false if it is not found. If it was found, the client retrieves the VPage from the offset information and acquires its lock to block other modifying access. In Viper, the actual record is not erased, but the corresponding free slot bit is set to 0 and the bitset is persisted to make the deletion durable. Then, the key is removed from the Offset Map before releasing the page lock and returning a successful deletion. For variable-sized records, the deletion bit is not set in the bitset but rather in the record metadata in the log. The size information is not modified, as it is required to skip the deleted record when scanning the VPage during recovery or compaction.

4.4.6 Space Reclamation

After various records have been deleted or re-inserted, the VPages contain many free slots or tombstoned records in the log. In order to reuse this free space, Viper runs a periodic background space reclamation process. In this reclamation, Viper scans the bitsets of the VPages to see how many free slots are available. If the number of free slots in a VBlock is higher than a configurable threshold and the VBlock is not currently "owned" by a client, the VBlock is compacted into a new VBlock, marked as free, and added to the free block queue. Compacting a VBlock is equivalent to re-inserting each record in that VBlock. Thus, when compacting many VBlocks, the records are tightly packed again. If a client reads a record that is currently being compacted, it either reads the stale offset and retries because the version lock of the compacted VPage has changed or it reads the new offset. Each VPage is locked for the entire duration of its compaction to avoid modifications throughout.

For variable-sized records, Viper checks the metadata of each VPage for the approximate free space on this page. If the VBlock reaches a configurable threshold, it gets compacted as in the fixed-sized process. After the compaction of a VBlock, it is marked as free with a *free* bit in its first VPage's lock byte. This allows Viper to recognize free VBlocks during a recovery. This process can also be used to deallocate VBlocks at the tail of the VBlock list and thus reduce its PMem footprint after many records have been deleted.

4.4.7 Recovery

A persistent KVS needs to be able to recover from a crash or be re-opened after a regular shutdown. In Viper, we handle both scenarios identically, as all required metadata is continuously persisted during its normal operational mode. Viper stores a small amount of metadata in PMem to keep track of the number of allocated VBlocks, the number of used VBlocks, and the total memory-mapped size. Every time new VBlocks are allocated in PMem, the metadata is updated to reflect the total number of allocated blocks. Additionally, every time a new VBlock is assigned to a client, the number of used blocks is incremented in the metadata.

When Viper is opened with an existing database, it checks this metadata and prepares for a recovery based on it. Viper maps existing VBlocks into its virtual address space and stores pointers to each VBlock, as described in Section 4.3.2. After mapping all VBlocks, Viper checks for the number of used blocks and scans those to retrieve the records in them. For each VPage, Viper checks which slots are set (fixed-sized) or scans the log for non-deleted records (variable-sized) and inserts the

offsets into the map. This can be parallelized by assigning disjoint VBlock-ranges to different threads. After scanning all VBlocks, the `next_block` counter in Viper is updated to the highest used block_id + 1, so that new clients receive fresh VBlocks (see Section 4.4.1).

4.5 Evaluation

In this section, we present the evaluation results of our implementation of Viper compared against other KVSs. In Section 4.5.1 we describe our setup, followed by an introduction of the other systems in Section 4.5.2. We present our Micro-Benchmark results in Section 4.5.3 and our YCSB evaluation in Section 4.5.4.

4.5.1 Setup and Methodology

We run all experiments on an Intel Xeon Gold 5220S CPU server and pin all threads to one socket to avoid cross-socket data access. The CPU has 18 cores (36 logical cores via hyperthreading). The socket is connected to 750 GB PMem, in six 128 GB Intel Optane Persistent Memory DIMMs, and to 96 GB DRAM. To access the Optane DIMMs directly, we use *devdax* mode. We prefill the stores with 100 million records before performing the benchmark operations and use 16 Byte keys (e.g., a UUID) and 200 Byte values, as these represent common sizes in real-world KVSs [20].

We implement our prototype of Viper in C++, compiled with GCC 9.3 on Ubuntu 20.04. We use and modify the CCEH map [113] for the offset map and low-level libpmem (v1.10) [124] calls to persist data in PMem. Our code is open-source and available on Github[10].

4.5.2 Other Systems

We evaluate Viper against six other systems to show the impact of various design choices in Viper: *FASTER*, *pmem-rocksdb*, *Dash*, *pmemkv*, *μTree*, and *Cross-Referencing Logs*. *FASTER* [24] (v1.8.0) is a state-of-the-art embedded hash-based KVS, which we run backed by PMem instead of SSD, making it a hybrid DRAM-PMem system. We initialize FASTER's hash index identically to the authors' evaluation with $\sim \#keys/2$ hash buckets, resulting in a 2 GB index. We set the log size to 6 GB, which is ~1/4 of the total raw data size. *pmem-rocksdb* [103] is a modified version of RocksDB to work explicitly with PMem by optimizing SSTables for and placing the WAL on it. We run pmem-rocksdb with the same configuration as the

10 https://github.com/hpides/viper

authors. These comparisons show the need for new PMem-aware designs instead of drop-in replacements and minor modifications.

We also compare Viper against two PMem-only setups to show the benefit of a hybrid design. As proposed in previous work, index structures can be used together with a persistent allocator as a KVS [101, 123, 148]. *Dash* [101] is a state-of-the-art PMem-optimized hash index that we pair with PMDK's persistent allocator [124]. A second PMem-only system we evaluate is Intel's hash-based *pmemkv* [125] (v1.4), which we run with the *cmap* backend [70, 126].

We also evaluate Viper against two hybrid PMem-DRAM systems. *μTree* [25] is a state-of-the-art hybrid BTree implementation that natively supports large values, making it suitable for a KVS use-case. We note that the performance of a BTree is expected to be slightly lower for single record operations, due to sorting overhead for additional range-query support. *Cross-Referencing Logs* (CRL) [57] were proposed to bridge the gap between volatile and persistent KVSs by persisting cross-referencing logs between two KVSs, one in DRAM and one in PMem. As CRL is not publicly available, we implement it (*CrlStore*) with Intel's volatile TBB concurrent hash map as the DRAM KVS [70] and the persistent map as the PMem KVS [126], which both fulfill the per-record locking requirements of CRL. As CRLs require front- and backend threads, we use a 1:1 mapping for all write operations, limiting our results to 18 threads in the plots. We do not employ a dynamic mapping, as proposed by the authors, because the backend threads constitute the bottleneck in our experiments. For *get* requests, we use only frontend threads.

4.5.3 Micro Benchmarks

In this section, we evaluate Viper's performance through various micro benchmarks. To this end, we discuss the performance of the four core KVS operations, the impact of different record sizes and variable-length records, followed by the systems' memory consumption. We then evaluate Viper-internal design choices by showing the impact of in-place updates and of data placement on DRAM or PMem, followed by an operation breakdown, space reclamation impact, and recovery performance.

Key-Value Store Operations

To understand the throughput of Viper, we compare it against the other KVSs for the core KVS operations *insert*, *get*, *update*, and *delete*. We initially fill each KVS with 100 million 216 Byte records (16 Byte key, 200 Byte value), before performing 50 million individual operations on them. Each client inserts consecutive keys from a disjoint range. For update, get, and delete, we uniformly choose a random

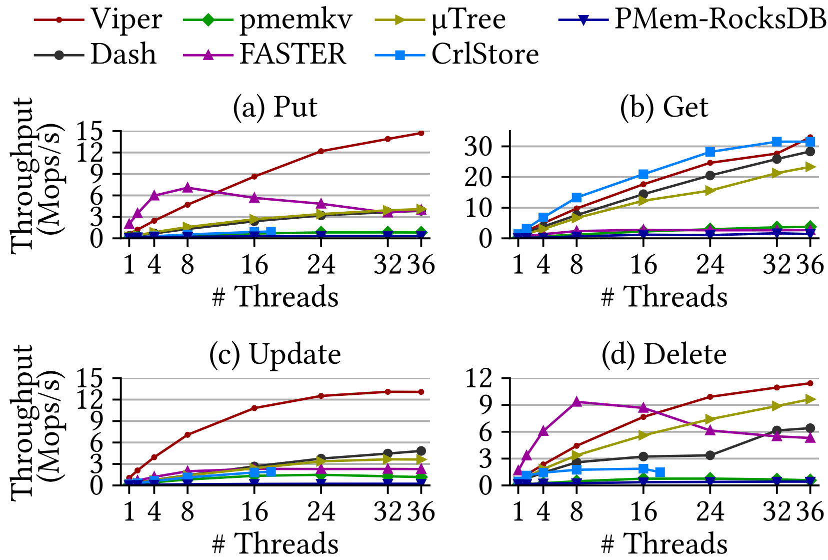

Figure 4.6: Core KVS operations. Viper scales better than the other persistent KVSs for (a) *put* due to efficient data layout and thread distribution, (b) *get* due to direct storage access, (c) *update* because of in-place modifications, and (d) *delete* due to minimum writes.

key in each call. We use a fresh KVS for each operation to avoid unintentional caching effects. For all *get* operations, we explicitly read the value to ensure that it is accessed from the underlying medium and not just pointed to.

The results are shown in Figure 4.6. In (a), we see that Viper's insert throughput scales well with the number of threads due to its efficient sequential access across multiple DIMMs via the Viper clients, reaching a peak of 15 million puts/s with 36 threads. The PMem-optimized Dash and µTree also scale but achieve only ~4 Mops/s. Both are limited by the record allocation outside of the actual index structure, which shows the need for a more structured insert mechanism. FASTER performs better than Viper for few threads, as the data is initially written to DRAM and is not persisted. However, after 8 threads its performance decreases. Once FASTER's DRAM-based log is full, it writes old segments to PMem to free space. This becomes a bottleneck, as the log needs to wait until the segment was copied and flushed before it can allocate a new segment to write to. The other systems do not scale well and achieve fewer than 1 million inserts/s due to unoptimized random hash map operations performed in PMem.

Retrieving records (b) is split into two groups. FASTER, pmemkv, and RocksDB do not perform well for random *get* request due to inefficient lookups in PMem,

all peaking below 4 Mops/s. FASTER and RocksDB are optimized for access from disk-storage, disregarding random access capabilities PMem, while pmemkv is built for PMem but with an unoptimized hash index. The other group of systems are optimized for PMem and achieve peaks between 25 and 35 Mops/s. This shows that *get* performance heavily depends on the chosen (hash-) index implementation and that the DRAM-based index in Viper does not significantly outperform PMem-based Dash. We plan to investigate the use of different index types in Viper in future work, as a recent study shows that, e.g., Dash achieves significantly higher lookup rates than CCEH [55]. In Figure 4.11, we show that DRAM-based Viper achieves ~50 Mops/s, indicating that CrlStore is limited by the TBB concurrent map in this evaluation.

In real world use-cases, record updates are often small modifications, e.g., 8 Byte counter updates [20]. In such a workload (c), we see that Viper outperforms all other systems due to its atomic PMem-aware in-place modification compared to the read-modify-write semantics of the other systems. We discuss the different update semantics in Viper in more detail in Section 4.5.3.

Deleting (d) records behaves similarly to updating in Viper, as it performs a key lookup followed by a small write, i.e., invalidating the slot. Other systems' delete performance is higher than their respective update performance, as many use a tombstone invalidation without the need to insert a new entry.

Our evaluation shows that for inserts, a PMem-specific sequential write pattern considerably improves the performance over batched disk-based approaches or random PMem allocations by 4–18×. Also, the update performance of Viper is superior, as it can perform in-place updates in persistent storage, which other systems cannot. For data retrieval, Viper performs on par with comparable systems. As PMem-RocksDB performs worse than all other systems, we omit it from future evaluation due to limited space.

Key-Value Record Size

To understand the impact of record sizes, we evaluate all systems with varying key and value lengths. We evaluate the impact of very small records (8 Byte key, 8 Byte value), more common sizes (16 B, 100 B) and (32 B, 500 B), as well as large records (100 B, 900 B). We define a fixed prefill data size of 20 GB, which we divide by the record sizes to get the number of records to prefill each system with, i.e., $20\ GB/16\ B$ = 1.25 billion 16 Byte records and 92/37/20 million 216/532/1000 Byte records. We then insert 10 GB in the same manner, i.e., exactly half as many records as the prefill. In a second workload, we issue 50 million *get* requests on a prefilled

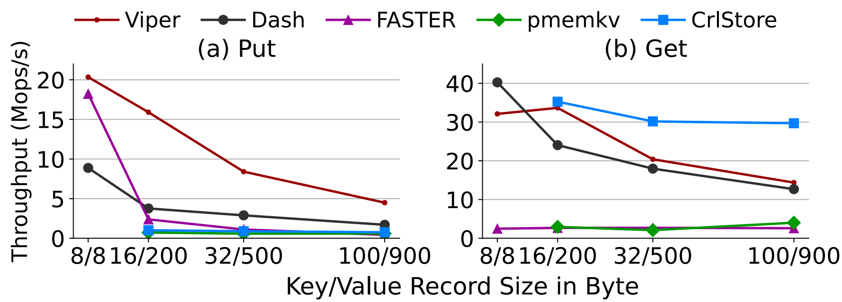

Figure 4.7: Key-value size impact.

KVS. All runs are performed with 36 threads. We omit μTree, as it does not support large keys.

The results are shown in Figure 4.7. For 16 Byte records, Viper achieves ~20 M puts/s and decreases linearly with an increasing record size, as it becomes PMem bandwidth-bound. FASTER also achieves nearly 20 M puts/s for 16 Byte records, as many of them fit into the DRAM-based log and are not persisted. With increasing record sizes, FASTER's performance drops to under 2 Mops/s as fewer records fit into the log, requiring more frequent PMem flushes. From this result, we see that efficient access patterns to PMem, as employed in Viper, have a higher impact on the overall performance than simply reducing the number of PMem flushes, as done in FASTER, via a DRAM buffer followed by a large PMem flush. We note that especially for larger records, the impact of a single additional metadata flush decreases, as multiple flushes are required for the record alone.

Dash benefits from 16 Byte records, as it does not require an extra memory allocation outside of the index. However, its insert performance is only about 50% of Viper's, which demonstrates the high overhead of random writes to PMem over sequential ones. For larger records, random memory allocations become the bottleneck in Dash. Both pmemkv and CrlStore cannot insert the 1.25 billion 16 Byte records as they run out of memory. We note that this behavior is expected, as explained in the PMDK documentation: "allocations of a size less than 64 Bytes [are] extremely inefficient and discouraged."[11] Thus, both pmemkv and a default allocator KVS are not suitable for small records, and for larger records, they are limited by their inefficient PMem writes.

The *get* performance trend of Viper is similar to the insert performance, where access to larger records is bandwidth-bound. Surprisingly, 16 Byte *gets* are less efficient than 216 Byte, as CCEH performs better with fewer entries. Dash retrieves

11 https://pmem.io/pmdk/manpages/linux/v1.8/libpmemobj/pmemobj_alloc.3

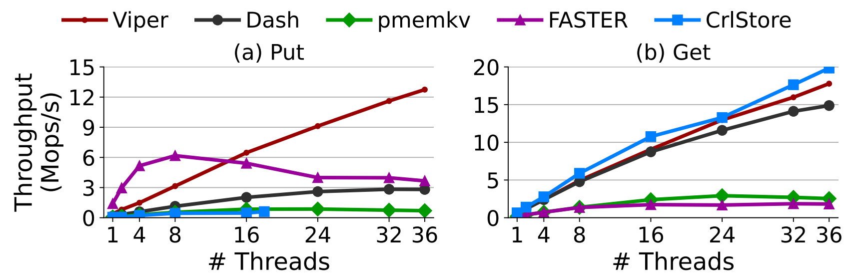

Figure 4.8: Variable-sized ~216 Byte records.

16 Byte records very efficiently, as the values are stored directly in the map without indirection. For larger records, its performance is also bandwidth-bound. CrlStore exhibits a consistently high *get* performance, as all requests are answered from DRAM without PMem access. Both FASTER and pmemkv show the same low performance as in the previous section due to inefficient access.

Variable-sized Records

In this benchmark, we evaluate the impact of variable-sized records on the performance of the systems. To this end, we prefill 100 million records of about 216 Byte, with a normal distribution around 16 Byte for the key and 200 Byte for value size. We then perform each 50 million puts and gets and measure the throughput.

The results shown in Figure 4.8 are in line with those of the core operations (see Figure 4.6). For *puts*, Viper clearly outperforms the other systems due to its efficient VPage design. Record retrieval also follows the same trend of fixed-sized records discussed above. However, both *put* and *get* achieve lower overall throughput compared to fixed-sized records. For fixed-sized records, the compiler generates SIMD `mov` instruction, while regular `mov` instructions are used for variable-length. The *get* performance is also lower for variable records, as they additionally require more data reads than fixed records. Viper must read the size metadata before retrieving the actual value, while fixed records require only pointer arithmetic due to known offsets at compile time.

Memory Consumption

We evaluate the total DRAM and PMem consumption to better understand the systems' resource requirements. We fill each system with the default 100 million

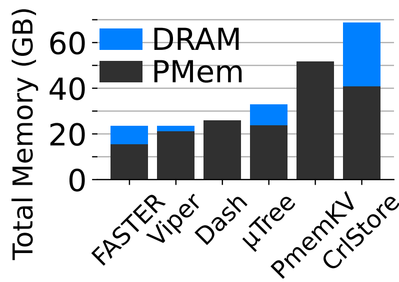

Figure 4.9: Total memory.

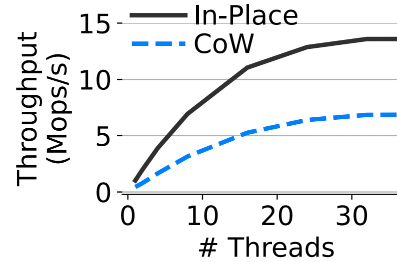

Figure 4.10: Update strategy.

records, i.e., 20.1 GB raw data (1GB = 2^{30}B). We measure the DRAM and PMem consumption with Intel's *VTune*[12], `pmap`[13], and `pmempool`[14].

The results are shown in Figure 4.9. Viper consumes 21.2 GB of PMem and 2.3 GB of DRAM. The DRAM consumption is attributed nearly completely to the offset map. FASTER consumes slightly less memory overall but significantly more DRAM due to its volatile log, which holds a large part of the data. Dash and µTree both require 23.8 GB for the data via the allocator, being ~10% less efficient than Viper. However, Dash requires only an additional 2.1 GB PMem for its index while µTree requires nearly 9 GB of DRAM for its tree index. pmemkv is very inefficient in its memory consumption, requiring more than twice the raw data size in PMem at 52 GB. In our implementation, CrlStore requires 28 GB of DRAM and 41 GB of PMem, as it needs to store each record twice.

DRAM is a scarce and expensive resource compared to PMem, with a capacity of only about 1/8× on our server and a 9× higher $/GB ratio [3, 69]. Viper's DRAM-PMem ratio is ~1/10 for 216 Byte records and lower for larger keys due to fingerprinting, i.e., the DRAM consumption depends solely on the number of records. Thus, Viper efficiently manages DRAM and supports larger configurations.

Update Strategy

A recent study by Facebook shows that certain workloads consist of many small updates, e.g., 8 Byte counter updates [20]. For these workloads, efficient in-place modification significantly reduces read- and write-amplification. In Figure 4.10 we show the advantage of in-place updates over copy-on-write (*CoW*) updates. When atomically updating only 8 Byte of a value, Viper achieves more than 2× updates/s

12 software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html
13 linux.die.net/man/1/pmap
14 pmem.io/pmdk/manpages/linux/v1.8/pmempool/pmempool-info.1.html

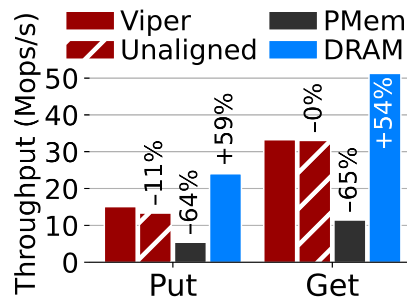

Figure 4.11: Viper versions.

compared to CoW. If an atomic update is not possible, Viper still outperforms the other systems when reading, modifying, and re-inserting the value (cf. Fig 4.6). Recent work [90, 130] has also shown the advantage of in-place updates, thus, supporting larger in-place modifications poses an interesting challenge for future work.

Viper Versions

In Figure 4.11, we evaluate four Viper versions to understand the impact of data placement in our design, i.e., by placing data + index in *PMem* or *DRAM*, by placing the data in PMem and the index in DRAM (*Viper*), and by using unaligned VPages in PMem shifted by 2048 Byte (*Unaligned*). We run the experiments with 36 threads. This evaluation supports our design choice of DIMM-aligned storage, as unaligned writes reduce *put* performance by 11%, due to a worse thread-to-DIMM distribution. Random *gets* are affected less than 1%, as they are point lookups that rarely cross DIMM borders.

Placing all data in PMem achieves only ~1/3× performance of the hybrid approach, clearly showing the advantage of a hybrid design when aiming for higher throughput. In Figure 4.12, we see that 60% of a *put* are already spent in PMem. Adding the index to PMem increases the absolute time spent on *Offset Map* operations and decreases PMem bandwidth due to inefficient access.

On the other side, hybrid Viper achieves ~2/3× of a DRAM-only Viper. The 1.4 µs spent in PMem for *put* are now approximately halved (cf. Fig 4.1a), reducing the *put* duration to ~1.6 µs, allowing for ~22 Mops/s. Similarly, the time spent fetching data from PMem is reduced by 2.5×, allowing for ~50% more ops/s. This evaluation shows that a hybrid approach significantly outperforms a PMem-only one, while the cost of data persistence is only about 33%. To further close the gap between

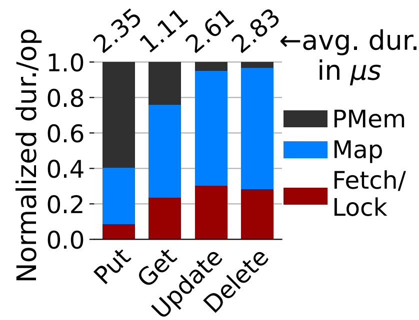

Figure 4.12: Operation breakdown.

PMem- and DRAM-based storage, we plan to investigate caching strategies in DRAM in future work.

Operation Breakdown

To better understand the individual operations, we break them down into common sub-parts. We prefill Viper before performing 50 million operations with 36 threads. We split the operations evenly into a mixed 25%-each workload. We normalize the runtime of each operation to 1 and present the time spent on PMem access, Offset Map access, and VPage fetching/locking.

The results are shown in Figure 4.12. For *put*, we see that most of the time (~60%) is spent on writing the record to PMem. Due to its VPage and client design, very little time is spent on locking and fetching the VPage, as it is cached in the client. As PMem-write speeds are close to those of DRAM, Viper makes good use of the time spent on inserting. However, adding random PMem writes, e.g., in a persistent index, might significantly impact the performance benefits gained by the sequential VPage writes.

Both *updates* and *deletes* require ~30% of the operation time to initially fetch the required VPage and lock it. The majority of the time is then spent in the map, which also includes fingerprint lookups in PMem, to retrieve the correct record offset. The final record update/invalidation is only a small part of the operation.

Retrieving data is similar to updates and deletes, in that it initially requires ~20% to fetch the VPage (but not lock it). Again, the majority (> 50%) is spent in the map lookup and fingerprint resolution. Finally, compared to updates/deletes, 20% is spent on retrieving the actual record and copying it to DRAM.

This breakdown shows that Viper efficiently handles the core operations. The

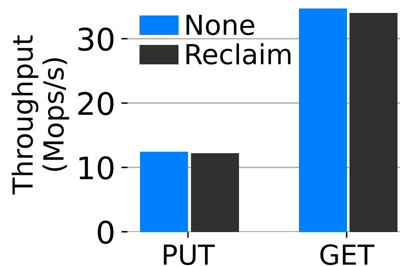

Figure 4.13: Space reclamation.

DRAM-based index access takes up a significant portion and Viper might benefit from different index designs, which we plan to investigate in future work.

Space Reclamation

To evaluate the impact of space reclamation on insert and get workloads, we prefill Viper with 216 Byte records and randomly delete 33% of the records without space reclamation. We then manually trigger a compaction of all VBlocks and start 32 parallel threads that *put* or *get* records.

In Figure 4.13, we show that running space reclamation in the background has only a marginal impact on the performance of read workloads, i.e., ~2% and no impact on write workloads, as each client inserts records independently and Viper reuses existing VBlocks without new allocations. Thus, space reclamation should be used to reduce the PMem footprint if free CPU resources are available. If Viper is not run at capacity, reclamation can be parallelized to reduce its runtime or a higher threshold can be set to avoid reclaiming every deleted record.

Recovery

As a persistent KVS should be able to restart after a crash or shutdown, we evaluate Viper's recovery performance. We prefill 100 million 216 Byte records and recover using a varying number of threads. A single thread requires 38 seconds to fully restart Viper. More threads reduce the recovery time to 19/10/5/4 seconds with 2/4/8/16 threads. 36 threads recover Viper in 2.3 s.

A disadvantage of a hybrid KVS is that the volatile index needs to be rebuilt when restarting. For a very large KVS, e.g., 1 TB, this can take up to 2 minutes. In Viper, we optimize for the average case of a running database, i.e., improve *put*/*get* performance instead of the worst case, i.e., a crash. However, recovery time

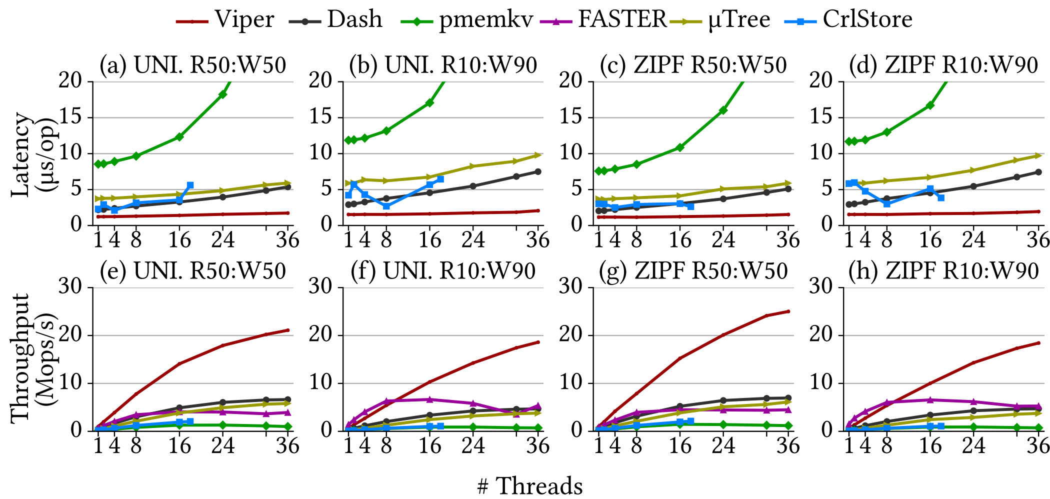

Figure 4.14: YCSB latency and throughput for mixed read/write (read-ratio:write-ratio) uniform (UNI.) and Zipfian (ZIPF) workloads.

is an important aspect of KVSs and we plan to investigate the trade-off between operational and recovery performance in future work.

4.5.4 YCSB

In this section, we evaluate Viper and the other systems with the widely used Yahoo Cloud Serving Benchmark (YCSB) [29]. We discuss latency and throughput as both are important metrics depending on the exact application, as well as mixed workloads. We split our evaluation along three axes, *i)* latency and throughput (top/bottom row), *ii)* uniform and Zipfian distribution (left/right half), and *iii)* 50:50 and 10:90 read:write workloads (left/right quarter). As YCSB is Java-based and Viper does not offer a network interface, we generate the workloads (8 Byte keys, 200 Byte values) using YCSB and then map them into our C++ benchmark for execution. We show the average latency in microseconds measured with *HdrHistogram* [51] and the throughput in million operations/s.

The results are shown in Figure 4.14. We first look at the latency measurements in the top row, i.e., (a) – (d). We see that Viper has a very low average latency for all four workloads. It increases from 1.2 µs with one thread to a maximum of 2 µs with 36 threads. The Zipfian workloads show a slightly lower latency, due to better caching effects. Dash and µTree have similar latency for all workloads, which is 3–5× higher than Viper's and is mainly caused by the random record allocation. CrlStore also shows low latency, as writes return as soon as they are persisted in

the log and frontend KVS. However, while the average latency is low, the 99.9^{th}-percentiles of Dash/µTree/CrlStore in the uniform workloads reach 150/110/240 µs compared to only 25 µs in Viper. pmemkv has significantly higher latency than the other systems in all workloads and peaks at ~50 µs. For all systems, we see a slightly lower latency in the 50:50 workloads compared to the write-heavy workloads as *get* requests perform better in all systems. As FASTER is inherently asynchronous and request completion intervals must be tuned by the user, we omit its latency as it is not directly comparable.

The throughput of all systems follows the trend of the respective latency. For Viper, we see slightly lower maximum throughput (~20 Mops/s) in the uniform workloads than expected compared to the average of the individual *put* and *get* operations as shown in Figure 4.6, which would reach ~24 Mops/s. In the realistic YCSB workload, there is more mixed access to PMem, which decreases the bandwidth [147], compared to our isolated micro benchmarks. The throughput of the other systems is also similar to the numbers shown in Figure 4.6. However, all systems are severely limited by inefficient insert operations. Dash and µTree peak at ~8 Mops/s and the other systems reach fewer than 5 Mops/s.

YCSB shows that Viper consistently outperforms existing KVSs with an average latency below 2 µs/op and a maximum throughput of over 19 Mops/s for both write-heavy and mixed workloads. Overall, Viper's throughput is significantly higher in all workloads compared to the other systems, ranging from 3× to 27×, making its design choices a good fit for real-world workloads.

4.6 Related Work

Viper builds on many techniques from prior KVSs, concurrent hash maps, pure PMem data structures, and hybrid PMem-DRAM structures. In this section, we briefly discuss related work.

Traditional Key-Value Stores. Popular in-memory KVSs such as Redis [128], memcached [109], or MICA [95] optimize for a purely in-memory cache-like use case for maximum performance. They do not persist the data in order to avoid expensive disk access at the cost of data-loss after a system shutdown or crash.

Prior research in persistent KVSs is extensive and focuses mainly on avoiding expensive read- and write-amplification to either SSDs, HDDs, or both [102, 110, 132]. Popular stores such as RocksDB [38], LevelDB [43], and Cassandra [84] use log-structured merge trees with an in-memory table for insertions to reduce write-amplification. To ensure the persistence of the data in the in-memory table, they often employ file-based Write-Ahead-Logging (WAL), which quickly becomes a

bottleneck. *FASTER* [24] is a modern KVS that uses an in-memory hash index and a hybrid log to store records on disk with a volatile "tail" that allows for in-place updates. Data in the volatile tail may be lost during a crash. While this approach works very well in some use-cases, we aim for a stronger storage model in Viper, in which data-persistence is guaranteed. With Viper, we propose a persistent KVS that leverages PMem instead of disk to allow for efficient operation without central log-based bottlenecks.

PMem-Based Key-Value Stores. Recent research also focuses on PMem-based KVSs. *RStore* [91] is a hybrid PMem-DRAM KVS that focuses on reducing tail-latency via asynchronous message passing and log-structured storage. FlatStore [26] also employs a hybrid design based on record batching and cross-core stealing from RDMA-connected request buffers. As RStore and FlatStore are designed as a KVS server, their core design decisions are tightly coupled to networking, include controlling their own threads, and reducing network overhead through user-space networking. Viper's design as an embedded KVS is significantly different, as it does not require any network interaction and more importantly, it does not control its own threads. *HiKV* [145] proposes a hybrid index for a KVS, where a hash index is stored in PMem and a B-Tree is located in DRAM for efficient range queries. However, as only the B-Tree is located in DRAM, all point queries are performed on PMem essentially making it a PMem-only KVS compared to Viper. *LibreKV* [96] also builds a hybrid index where data is initially inserted into a DRAM-based hash map and later merged into a PMem-based hash map once it reaches a certain filling degree. However, LibreKV does not offer consistency as all data in DRAM is lost during a crash. *NVLevel* [98] is an LSM tree-based KVS that uses multiple PMem-based memtables and compacts these into SSTables on disk once they are full. NVLevel uses disk as its storage medium, thus being limited similarly to other disk-based KVSs. In Viper, we propose PMem-specific access patterns for real hardware to efficiently store and retrieve data directly in and from PMem.

PMem Data Structures. Several (hybrid) PMem data structures have been proposed that introduce concepts used in Viper. *Dash* [101], *NVTree* [148], and *FPTree* [123] use a lock-per-node approach in their hash map and B-Tree structures, which we leverage in our VPages. Various work has focused on the advantages of using a hybrid DRAM-PMem approach [25, 97, 123, 129, 156], from which we derive our hybrid index-storage model. Lersch et al. [90] show that in-place updates are preferred over copy-on-write for PMem and that fingerprinting is an effective mean to reduce PMem lookups.

PMem Programming. Yang et al. [147], Izraelevitz et al. [71], and van Renen et al. [138] show how access patterns affect the performance of PMem, which we rely

on in Viper. PMDK [124] is the de-facto standard toolkit to interact with persistent memory. We make use of its low-level methods in our implementation.

4.7 Conclusion

In this chapter, we present Viper, a hybrid PMem-DRAM key-value store that leverages PMem-specific access patterns to efficiently store and retrieve data while providing full data persistence. We propose three key design choices for hybrid PMem-DRAM systems based on efficient PMem access patterns for real hardware, *direct PMem-writes*, *uniform thread-to-DIMM distribution*, and *DIMM-aligned storage segments*. We also discuss how to implement core KVS operations in such a system with regard to correct persistence guarantees. Our evaluation shows the efficiency of our design choices, as Viper significantly outperforms existing PMem-only, hybrid, and disk-based KVSs by 4–18× for write workloads, while matching or surpassing their *get* performance. For future work, we propose to investigate alternative index designs and as PMem shows similar performance characteristics to DRAM for certain access, we suggest to investigate moving parts of the index to PMem. With Viper, we provide a foundation for future work on PMem-aware storage systems and hybrid PMem-DRAM designs based on real PMem hardware characteristics.

5 Darwin: Scale-In Stream Processing

The majority of this chapter has been published in [16].

5.1 Introduction

Today's large-scale Internet companies use stream processing engines (SPEs) to process up to terabytes of incoming data per second [5]. However, recent studies show that widely used SPEs such as Apache Flink and Spark Streaming do not fully utilize the underlying hardware and are resource inefficient [151, 152]. Thus, companies must *scale-out* their analytics jobs to millions of cores and tens of thousands of commodity servers. At this scale, large infrastructure teams are necessary to optimize analytics pipelines to maintain such a high level of processing capacity.

We briefly illustrate the required scale with an example from Alibaba's 2020 Singles' Day. At its peak, they processed 4 billion events per second in a Flink cluster with 1.5 million CPUs [5]. Running a general purpose VM (e.g., ecs.g6.4xlarge) with 16 cores on Alibaba's cloud currently costs $1/hour [4]. Scaling this to 1.5 million cores with 93,750 VMs totals at $93,750/hour or $2.25 million for the entire day, just in nominal infrastructure cost.

To overcome resource inefficiency, new *scale-up* SPEs were proposed that, e.g., use query compilation [45], optimize for NUMA-awareness [153], or utilize GPU-CPU co-processing [80], promising up to hundreds of millions of events per second on a single node. To showcase the stark contrast of scale-up to scale-out system performance, we assume a scale-up system with 100 million events per second in our Alibaba example. With this system, the workload could run on 40 VMs with 52 cores each (e.g., ecs.g6.26xlarge at $6/h) for a total of $5760/day [4], resulting in a nearly 400× reduction in price. While this calculation is simplified, it clearly shows the huge gap between what is achieved with current scale-out SPEs and what is possible with proposed scale-up systems.

However, the price to pay for this high performance and reduced infrastructure cost lies in a reduced feature set. While scale-up systems utilize the hardware more efficiently, they lack support for larger-than-memory state and crash recovery, which limits their use in production setups. When the server or application crashes,

all state is lost and must be reprocessed. For workloads with small windows, reprocessing includes only a few hours of old data. For unbounded or large-window streaming jobs with global state, reprocessing may span days or weeks of old data, which quickly becomes infeasible.

Regardless of scale, various business workloads require high availability and cannot afford a full reprocessing after all in-memory data is lost due to a crash in scale-up systems. This forces users to chose inefficient scale-out systems over highly tuned scale-up SPEs, as production-grade scale-out systems support persistent, larger-than-memory state and crash recovery. However, scale-out SPEs rely on slow secondary storage for this, further decreasing overall system performance. Recent developments in storage technology significantly improve the performance of persistent storage devices, allowing us to reduce the gap between high-performance and persistent state in SPEs. Persistent memory (PMem) offers byte-addressability at close-to-DRAM speed with SSD-like capacity [31, 147]. Efficiently incorporating PMem into streaming applications has the potential to radically shift the way SPEs interact with persistent state, which is why we investigate integrating it for efficient SPE state management in this chapter.

In our example, additional problems arise that limit the use of scale-up systems. Even with modern high speed networks, it is currently not possible to ingest TB of data per second into a single server. To overcome this limitation, large-scale pipelines must build on scale-out systems, accepting the performance penalty they entail. Thus, we see a huge performance gap between workloads that fit onto a single server and can run in scale-up systems and ones that do not fit onto a single server and must retreat to scale-out SPEs.

Following both efficient durable state management and hardware utilization, we observe that systems support either one or the other, but not both. In this space, we identify three key challenges current SPEs face: *state management*, *resource inefficiency*, and *overall system optimization*. Overcoming these challenges heavily impacts SPE performance and constitutes an important step towards industry adoption and system maturity.

Based on these challenges, we propose *scale-in* stream processing, a new paradigm that adapts to varying application demands by achieving high hardware utilization on a wide range of hardware setups, reducing overall infrastructure requirements. In contrast to scaling-up or -out, it focuses on fully utilizing the given hardware instead of demanding more or ever-larger servers. Scale-in processing combines scale-out and scale-up concepts to efficiently process streaming data without sacrificing larger-than-memory state or crash recovery. To scale-in, we adapt common scale-up approaches that optimize for the underlying hardware and common scale-out approaches that enable large state management. Compared to the current

performance drop when switching from scale-up to scale-out SPEs, scale-in allows for graceful scaling when workloads exceed single server by optimizing for both single- and multi-node setups.

To this end, we introduce *Darwin*, our scale-in SPE prototype. Darwin leverages modern storage and query compilation to handle large recoverable state and fully utilize the underlying hardware. In summary, we make the following contributions.

1) We propose *scale-in* stream processing, a new paradigm that adapts to varying application demands by achieving high hardware utilization on a wide range of hardware setups, reducing overall infrastructure requirements.

2) We present Darwin, a scale-in SPE prototype that supports recoverable larger-than-memory state while optimizing for high overall hardware utilization.

3) Based on Darwin's design principles, we highlight the potential of PMem storage engines for SPEs compared to traditional disk-based ones.

The remainder of this chapter is structured as follows. We briefly discuss some background on stream processing in Section 5.2. In Section 5.3, we present current challenges in SPEs. In Section 5.4, we present scale-in processing and its opportunities, with a focus on state management with persistent memory. We introduce our scale-in SPE prototype Darwin in Section 5.5 before concluding in Section 5.6.

5.2 Background

In this section, we briefly present some background on stream processing concepts. Compared to traditional database workloads that answer continuously incoming queries on a snapshot of data (*data-at-rest*), stream processing engines (SPEs) invert this model and answer a set of pre-defined or standing queries on incoming data (*query-at-rest*). Data is ingested by, e.g., scanning remote file directories [1] or by reading from message queues such as Apache Kafka [83]. Once the incoming data is processed, the results are again written to, e.g., files, external databases, or another message queue. This chaining allows developers to create complex processing graphs between SPEs and other systems.

SPEs commonly do not store all historical data but only the data that is relevant to answering the current queries. Once data is no longer needed, it can be removed to free up resources. As data is continuously ingested at high rates and queries may span days, weeks, or months of data, it is important for SPEs to recover quickly after a crash. If this is not supported, all data up until that point must be reprocessed to

recover the lost state, which quickly becomes prohibitively expensive, as state can grow into the terabytes [33].

As aggregations on unbounded data streams is not always possible, SPE queries often contain *windows*. These windows split the unbounded stream into a stream of bounded chunks [13, 22, 93, 137]. On such a chunk, all aggregations are computable. For example, calculating the median of an infinite stream is not possible, while calculating the median of the past 60 minutes is. In line with our description above, if the content of a window is no longer needed, it can be discarded. A common window type is the *tumbling* window, which is defined via a window length l. Every l time units, the window is completed and the next window of length l starts. An simple example use of a tumbling window is to calculate the average stock price of every hour, i.e., $l = 60$ minutes. Depending on the application, more complex window types can be used.

5.3 Current SPE Challenges

Recent work in stream processing focuses either on scale-up or scale-out concepts. Scale-up systems optimize for high system utilization while scale-out systems focus on application stability and efficient large state management. Both areas show promising advancements, but combining them has received little attention. In Section 5.3.1, we compare nine SPEs to see how they offer resource efficiency and state management. From this comparison, we observe that numerous challenges remain in the intersection of scale-up and scale-out systems. In Sections 5.3.2 to 5.3.4, we present three major challenges that current SPEs face: *state management*, *resource inefficiency*, and *overall system optimization*.

5.3.1 Focus of Existing Systems

In this section, we briefly discuss the feature sets of common SPEs with regard to state management and resource efficiency. We show the comparison in Table 5.1. For most features, we observe a clear distinction between scale-up and scale-out systems (see *Scale* column in the table). While scale-out systems support recoverable, larger-than-memory state with persistent or distributed state management, all scale-up systems support only in-memory state without recovery. Instead, scale-up systems offer optimizations for higher hardware resource utilization on a single server. These include query compilation, NUMA-awareness, lock-free data sharing, and GPU co-processing. Except for NUMA-awareness, none of these are exclusive to large servers and are applicable optimizations also in commodity machines. They

Table 5.1: Feature set of existing SPEs with regard to state management, crash recovery, and hardware resource efficiency.

System	Scale	Lang.	State	Recovery	Hardware Efficiency
Flink [21]	out	JVM	in-memory + persistent	✓	–
Spark Streaming [150]	out	JVM	in-memory + persistent	✓	–
Drizzle [139]	out	JVM	in-memory + persistent	✓	–
Hazelcast Jet [42]	out	JVM	distributed in-memory	✓	cooperative multi-threading
Briskstream [153]	up	JVM	in-memory	–	NUMA-aware scheduling
Saber [80]	up	JVM	in-memory	–	CPU/GPU co-processing
Trill [23]	up	C#	in-memory	–	query compilation, row/column layout
StreamBox [111]	up	C++	in-memory	–	NUMA-aware, lock-free
Grizzly [45]	up	C++	in-memory	–	query compilation, NUMA-aware

represent a large area of improvement for scale-out systems, which currently offer very little hardware optimization.

Finally, all selected scale-out systems target the JVM with managed languages. The group of scale-up systems is not as homogeneous, spanning managed languages and C++. The recent scale-up SPE Grizzly demonstrates large performance gains by using a system language such as C++, outperforming other scale-up and JVM-based systems by orders of magnitude [45]. Thus, using a system-level language is highly advantageous to achieve high utilization when targeting the underlying hardware.

Overall, we observe distinct characteristics for scale-up and scale-out systems. While none of the scale-up systems offer recoverable state management, the scale-out systems neglect hardware optimizations. To achieve high performance and resource efficiency, future SPEs must focus on combining these features. Scaling-up should not come at the price of data loss and scaling-out should not come at the price of poor hardware utilization.

5.3.2 State Management

Recent scale-up SPEs focus on maximizing hardware utilization through, e.g., query compilation [45], CPU-GPU co-processing [80], or NUMA-awareness [153]. Yet, none of these systems support larger-than-memory state or crash recovery, both important features for industry adoption. We identify efficient state management as a largely uninvestigated topic in scale-up SPEs compared to numerous computational improvements.

Common scale-out SPEs such as Apache Flink use persistent state backends (e.g., RocksDB) to handle larger-than-memory state. However, general-purpose key-value stores do not always fit stream-specific state access patterns [77]. They treat state as a black box, while many streaming-specific patterns are known in advance. Also, currently used general-purpose stores are not optimized for emerging storage technology. Research on modern storage-aware systems shows significant performance gains compared to traditional approaches [14, 89]. Storage-aware and streaming-specific state management presents a wide range of research challenges to improve the overall performance of modern SPEs.

5.3.3 Resource Inefficiency

Recent studies show that widely used scale-out SPEs do not fully utilize the underlying hardware [151, 152]. When designing future SPEs, overcoming this resource inefficiency has great potential to reduce cost and improve performance. Higher resource utilization leads to higher system performance, i.e., higher throughput or lower latency. However, when processing smaller data volumes, scalability is not the primary concern for many users. Efficient server use allows users to reduce the number of required servers while still satisfying their performance needs. This not only reduces infrastructure cost but also overall system complexity.

5.3.4 Overall System Optimization

Database systems show that optimizing the overall system brings large performance benefits. Databases are commonly implemented in system languages such as C and C++, which compile to machine code. They are highly tuned towards the underlying system for maximum performance and offer, e.g., hardware-conscious joins and indexes, CPU-optimized scans, or NUMA-aware scheduling.

On the other hand, many widely used SPEs are written in high-level languages such as Java and Scala, targeting the JVM. Especially memory-management has a high performance impact due to, e.g., garbage collection overhead. Also, common

SPEs often do not optimize internal operators at the level known from databases. Overall, we observe a major gap between optimization levels in SPEs and database systems. With the increasing maturity of SPEs, reducing this gap is essential to improve the performance of future applications. Fortunately, many operations are similar in databases and SPEs, allowing us to benefit from database optimization research.

5.4 Scale-In Stream Processing

To overcome the current challenges in stream processing and acknowledge the fact that real-world setups have drastically varying performance and availability requirements, we propose *scale-in* processing. Scale-in processing is a new paradigm that adapts to varying application demands by achieving high hardware utilization on a wide range of hardware setups, reducing overall infrastructure requirements. In contrast to scaling-up or -out, it focuses on fully utilizing the given hardware instead of requiring more or ever-larger servers. We identify six opportunities for scale-in SPEs, based on the challenges discussed in Section 5.3. In Section 5.4.1, we present three opportunities to improve state management in scale-in systems: emerging persistent storage media, crash recovery, and streaming-specific access patterns. In Section 5.4.2, we discuss how query compilation and specialized network communication aid in overcoming general resource inefficiency. To increase the overall system performance, we discuss CPU-aware optimizations in Section 5.4.3.

To achieve high performance, scale-up systems commonly optimize for large high-end servers and scale-out SPEs commonly add more commodity servers. Solving performance limitations by adding more machines results in neglected individual server performance and poor resource utilization in scale-out systems. On the other hand, current scale-up systems are confined to a single server and require ever-larger machines to overcome performance issues. By combining both scale-up and scale-out concepts, scale-in SPEs treat every machine as a server that requires optimization, even if it contains only off-the-shelf components. To this end, scale-in systems adapt their execution to the underlying hardware and the specified queries. For large workloads, scale-in systems achieve the raw performance known from scale-up systems and for medium or small pipelines, they reduce hardware requirements while still offering key functionality such as crash recovery.

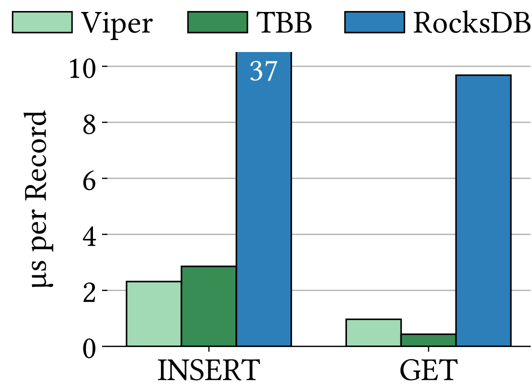

Figure 5.1: Insert and get performance.

5.4.1 Opportunities for State Management

In this section, we present three opportunities to improve state management in scale-in systems: emerging persistent storage media, crash recovery, and streaming-specific access patterns.

Persistent Storage

Scale-out systems use persistent storage to handle larger-than-memory state. This entails a large performance decrease as storage access is significantly slower than DRAM access. However, new persistent storage technology is closing the gap between slow secondary storage and fast volatile memory. Recent work on PMem storage systems shows that persistency can be achieved with less than 2× performance decrease [14]. Also, fast NVMe SSDs are used in modern database systems to extend storage capacity while still offering close-to-DRAM performance [89].

We show the performance of modern storage systems in Figure 5.1. We choose Intel's TBB concurrent hash map as a representative of in-memory state management, as it is used in recent scale-up SPEs [45, 111]. We choose RocksDB as a representative of a classical generic byte-based key-value store, as it is used in Apache Flink. Finally, we choose Viper as a representative of modern storage-aware key-value stores, which is based on a hybrid DRAM-PMem index and log structure (Chapter 4 and [14]). Viper stores records directly in a PMem log without intermediate buffering in DRAM. To leverage the higher random access performance of DRAM compared to PMem for index updates, its index is located in DRAM. This hybrid design achieves 4× higher insert rates than PMem-only stores.

We prefill 100 million 50 Byte records before measuring another 100 million

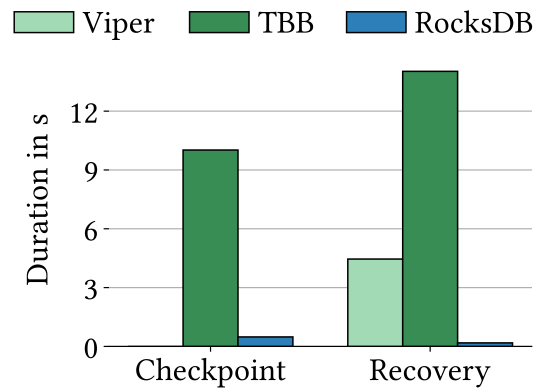

Figure 5.2: Checkpoint and recovery duration.

inserts/gets with 32 threads. Viper slightly outperforms TBB for inserts and is only ~2× slower for gets. We note that recent work shows TBB to not be the fastest concurrent in-memory storage system [24], but it is used in common scale-up SPEs [45, 111] and serves as an in-memory reference. The clear gap between Viper/TBB and RocksDB shows the major shift in persistent storage performance that systems can leverage. Unlike existing scale-out systems, storage-aware scale-in systems do not need to trade performance for persistence. Fast storage enables both efficient recoverability and high overall throughput.

Recovery

Considering server-local state is necessary when restarting an application after a crash and highly beneficial for regular application restarts. Scale-up systems run on high-end servers that contain hundreds of GBs of state. For specialized hardware, replacing the server is not always possible and transferring its state to another server quickly becomes a recovery bottleneck. In this case, state recovery must occur on the same server. Additionally, current scale-out SPEs use server-local state when restarting an application on the same node without a crash, e.g., when re-scaling or deploying a newer version. For both recovery and restarting, it is essential to have persistent state that outlives the application.

We show the advantage of using modern storage technology to achieve efficient checkpointing and recovery in Figure 5.2. In this microbenchmark, we store 200 million 50 Byte records, i.e., 10 GB raw data, in three different storage instances. As representatives of their respective system classes, we again use TBB, Viper, and RocksDB. To persist TBB's data, we store all entries in a tightly packed byte array in a file stored on SSD. We see that the persistent systems RocksDB and Viper

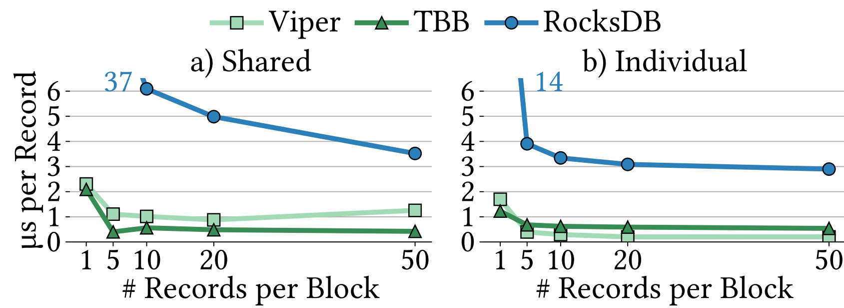

Figure 5.3: Grouped state access performance for a) *shared* and b) *individual* storage instances.

perform a checkpoint very efficiently. RocksDB must only flush its volatile write buffer and Viper must persist only metadata. TBB takes significantly longer, as all in-memory data is converted and copied to secondary storage, which is I/O-bound. For recovery, we see that RocksDB performs best, as it reads only metadata and immediately accepts requests. Viper must recover its volatile index, which depends on the number of entries. TBB reads all data from storage and re-creates its in-memory state, which takes significantly longer than the other two systems.

This experiment shows a large difference in the recovery performance of volatile and persistent state. Expanding on this is a key element of scale-in stream processing, as it impacts both runtime performance while checkpointing and start-up time after a crash. With state in the order of TBs, re-creating in-memory state from secondary storage becomes infeasible.

Streaming-specific State Access

In addition to storage-aware state management, streaming-specific access patterns improve performance even on slow storage media. We demonstrate this based on current behavior in Apache Flink. In Flink, windowed operations store incoming events in a list of records belonging to a given window. When using the RocksDB backend, the operator gets the current value, deserializes it, appends the new value, and serializes the updated list back to its byte representation. This incurs an unnecessarily high overhead for each record. As the records are not needed immediately and are accessed only as a list, they can be buffered in small in-memory lists before writing to RocksDB.

We show the effects of buffering records in Figure 5.3. In this experiment, we prefill 100 million 50 Byte records before performing another 100 million inserts with 32 threads. We distinguish between a shared instance, in which all threads

operate on the same store, and individual instances, in which each thread has its own store. We store the records in small blocks, consisting of up to 50 records. We observe that Viper outperforms TBB for individual instances but performs worse for the shared one. This difference is important when designing streaming state, as it can either be shared across operator instances, e.g., in Grizzly [45], or partitioned by key, e.g., in Flink. Our results show that depending on the underlying storage and chosen system, one or the other is more beneficial. More importantly, RocksDB performs between 14–20× worse than TBB when storing individual records, showing the high overhead of persistent storage. However, compared to individual records in TBB, 50 grouped records are only 2–3× slower in RocksDB. This shows that streaming-specific access significantly improves state performance, even for low-end storage media.

5.4.2 Opportunities for Resource Inefficiency

In this section, we discuss how query compilation and network communication aid in overcoming general resource inefficiency.

Query Compilation. At the core of scale-in processing, query compilation allows for hardware-conscious optimization on each server. This has many advantages, which are clearly demonstrated in previous work on SPEs [45] and database systems [118]. Compiling queries allows the compiler to optimize the execution for the given CPU without pre-compiling the runtime engine for all possible systems. This enables compiler features such as auto-vectorization and architecture-aware tuning without any development overhead.

To highlight this advantage, we run a short experiment with CPU-specific optimization enabled and disabled. We execute a query in-memory based on Nexmark Q3, containing an auction-like setup with a join and aggregation. We first compile the query generically, i.e., no CPU-aware compiler flags. This represents the general case in which we do not target the underlying hardware and use a generic implementation for all servers. We then enable CPU-optimizations via `-march=native`. On an Intel Xeon Gold 5220S CPU, adding the CPU optimizations achieves a 12% higher throughput without any changes to the code. This shows that even without explicitly writing CPU-optimized code, automatically targeting the underlying hardware is very beneficial.

Additionally, query compilation allows us to integrate query information as compile-time information, enabling additional optimizations. A push-based execution model improves data locality and reduces the number of virtual method calls compared to interpreted execution by compiling the entire execution into a tight loop [118]. Overall, query compilation allows us to tailor the execution exactly to

the given query, system, and user requirements. It is the foundation for numerous other optimizations that we describe below, such as CPU optimization and state management.

Network Communication. An important component of scale-out SPEs is network communication between the servers. Recent work shows that the network constitutes a performance bottleneck and causes resource inefficiency on the servers [79]. In their study, the authors show that scale-out systems like Flink reach network limits long before they reach the actual saturating data rate. When performing a distributed aggregation across multiple nodes with 1 GBit LAN, Flink sustains only 1.2 million events/s, which amount to only 40 MB or 1/3 of the network bandwidth. While this constitutes the main bottleneck in 1 GBit LAN, today even medium-sized cloud instances have 10 or more GBit/s network connections, matching or surpassing SSD storage bandwidth. Managing this bandwidth with techniques like late merging [151] to reduce data shuffling or user-space networking to reduce TCP/IP overhead [91], enables higher effective bandwidth utilization even for smaller workloads.

For large-scale workloads, advances in network technology drastically improve cross-server communication performance via high bandwidth Infiniband and RDMA connections of up to 200 GBit/s per network card [108]. Current research demonstrates that SPEs benefit from RDMA for data ingestion [151] and that RDMA-based message passing achieves very high throughput with low latency [136]. These two findings show that there is a large potential for optimizing SPEs through fast RDMA connections. Especially in combination with modern byte-addressable storage, such as PMem, this opens new opportunities for more efficient checkpointing, state migration, and recovery approaches.

5.4.3 Opportunities for System Optimization

In this section, we discuss CPU-aware optimizations to increase the overall system performance.

CPU-aware Optimization. Poor CPU utilization is a major contributor to resource inefficiency in current scale-out SPEs [152]. To overcome this, scale-in SPEs target the system's CPU to achieve higher overall utilization. Recent work shows the potential of adapting OLAP queries towards a given workload and system setup [46]. Exploring different computation modes, such as compiled or vectorized execution, is heavily researched in databases. However, they have received little attention in SPEs so far. Transferring these concepts to SPEs has the potential to further increase the overall system performance. For example, storing network-buffered records in a row or column format depending on the data and query allows

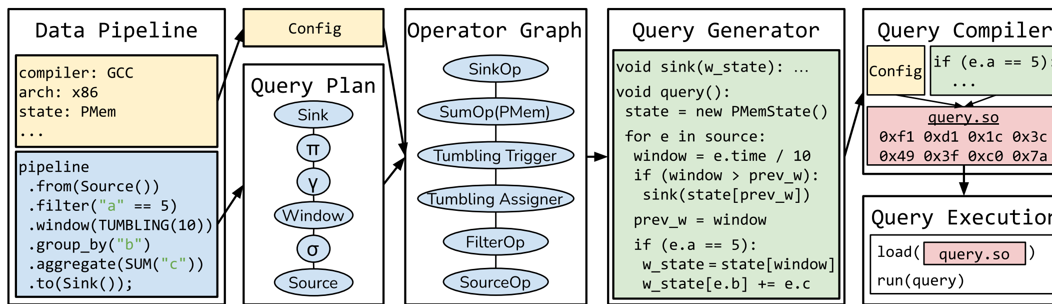

Figure 5.4: Darwin's architecture and execution flow.

for a performance trade-off between processing time and ingestion rate, while also enabling scalar or vectorized execution modes.

Another optimization is based on simultaneous multithreading (SMT). Depending on the workload, using SMT hides memory access latency while not using it improves cache locality. When multiple operators have low CPU consumption, they are placed on the same core to achieve better utilization. When CPU utilization is high, features such as explicit SIMD instructions achieve higher throughput with the same utilization. While query compilation generally targets the underlying CPU, explicit optimizations and domain knowledge additionally improve performance.

5.5 Introducing Darwin

In this section, we present Darwin, our scale-in SPE prototype. Darwin treats each server it runs on like a scale-up system by fully utilizing the given hardware configuration. To this end, it uses query compilation to generate efficient execution plans for each query targeting the server's hardware. This targeting currently includes storage-aware state management and CPU-specific optimizations. As Darwin is still in early stages, we plan to add support for more opportunities discussed in Section 5.4, e.g., network-aware data transfer to achieve efficient multi-server processing or efficient checkpointing and recovery mechanisms.

5.5.1 Darwin Architecture

In this section, we present Darwin's high-level architecture, components, and execution flow, as shown in Figure 5.4.

Data Pipeline. Users create queries via a data pipeline object, which currently offers an SQL-like API inspired by Apache Flink's Table API [39]. Additionally, the

user configures runtime options such as the compiler to use, which storage medium to use for state, and which architecture to optimize for. To run on heterogeneous hardware without manually adapting for each server, this config also auto-detects system characteristics.

Query Plan. From the query, a query plan is created. The query plan represents the logical version of the query, similar to relational algebra for classical database queries. Compared to relational algebra, it requires a few additions, e.g., for windowing logic or external I/O. The query plan is the first step in the execution in which optimizations are performed, e.g., predicate push-down.

Operator Graph. Together with the config, the query plan is translated to an operator graph. We briefly describe the translation based on the query plan shown in Figure 5.4. Starting from the sink, each node recursively translates its input node. The resulting operator graph represents the physical operators, i.e., the specific implementation chosen for the given query, system, and config. As start and end nodes, source and sink operators require special treatment. They contain buffering logic, e.g., for external network-based I/O, and either have no input or output operators. After the source is translated, the selection node is translated to an equality-filter operator. The translation of window nodes requires reordering, as windowing logic impacts the operator order. The tumbling window assignment and trigger run before the aggregation, but count-based triggers run afterward. Thus, window translation requires splitting and distributing certain nodes across the operator graph. For the aggregation, the translator chooses a hash-based sum aggregation operator with state in PMem, as specified by the user.

Query Generator. The query generator takes the operator graph and generates a C++ string representation of the actual query. In Figure 5.4, we show a pseudo-code version of the produced code. When the FilterOperator is called, it receives a record with schema information. From this, it generates a conditional statement with the correct predicate (e.g., equality) based on the filtered attribute (e.g, `e.a`). Depending on the predicate and execution model, the filter operator can also produce vectorized or SIMD-based filters, allowing for more fine-grained hardware optimization. Afterward, it calls the downstream *tumbling assigner*. The assigner generates code to assign the record to a tumbling window by mapping the record's timestamp to a window key. This process is continued until all operators are called and the query is fully generated.

As the windowed aggregation buffers data, it represents a pipeline breaker [45, 118]. The sink operator is executed after the aggregation is complete, i.e., when the window is triggered. The query generator creates a new function for the sink, which represents a new pipeline that can be executed independently. Splitting pipelines allows us to independently scale sources, sinks, and other operators.

As the query generation contains runtime information such as data types or filter conditions, the generated query is optimized accordingly. The SumOperator knows the key and value types, so it instantiates a state object with them. This is an advantage over key-value store interfaces such as RocksDB in Flink, which operate on generic byte representations. Storing records with explicit type information removes serialization and deserialization overhead and allows the compiler to optimize data move instructions, e.g., by issuing SIMD loads/stores instead of regular 8 Byte `movs` [14].

Query Compiler. Once the query code is generated, the query compiler compiles it. It uses information in the config to target the underlying hardware, e.g., by enabling vectorization features of the CPU. As the compiler runs independently of Darwin, users can specify a different compiler than was used to compile Darwin. Darwin can be compiled once and distributed while still providing flexibility towards the system it executes queries on. The code is compiled into a shared library that is dynamically loaded by Darwin during execution. This allows Darwin to interact with the query, e.g., when passing allocated memory, data, or other resources.

Query Execution. In the last step, the query is loaded and executed. Depending on the generated pipeline and specified parallelism, multiple source, sink, and operator instances are started. During execution, Darwin monitors the performance of the query to allow for changes in parallelism and thread placement. If pipelines have low utilization, they are merged to free resources. If pipelines are creating backpressure, Darwin splits them to keep up with the data rate. This approach allows for some flexibility during runtime when data loads vary or are skewed. It also supports adaptive changes to the query if gathered performance metrics and data characteristics allow for more aggressive optimization [45].

5.5.2 Performance

We compare the performance of Darwin with the state-of-the-art scale-up SPE Grizzly [45] and the widely used scale-out SPE Apache Flink [21]. In this experiment, we run a 60-second tumbling window sum aggregation on 32 Byte records with 15000 unique keys. Our server contains an Intel Xeon Gold 6240L CPU with 18 cores, 96 GB DRAM, and 1.5 TB (6× 256 GB) Intel Optane DC Persistent Memory 100 Series. The experiments are run with 32 threads. For the in-memory version, Grizzly and Darwin use the TBB concurrent map. For the persistent version, Darwin uses the hybrid DRAM-PMem key-value store Viper(Chapter 4 and [14]. Flink uses RocksDB.

We show the results in Figure 5.5. On the left, we see that Darwin performs

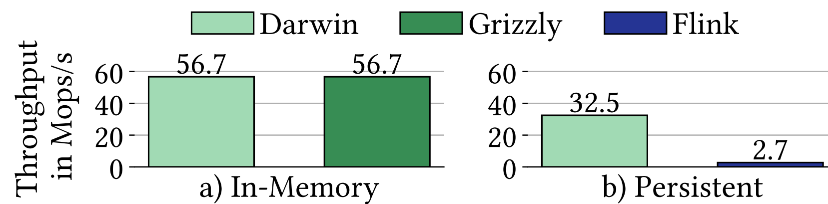

Figure 5.5: Throughput of Darwin, Grizzly, and Flink.

equally to Grizzly for in-memory processing, as both systems are limited by TBB to store the aggregations. We note that this is not the highest performance that Grizzly can achieve, but the additional optimizations proposed by the authors are orthogonal to the basic concept and could also be applied to Darwin. On the right, we see that Darwin outperforms Flink by over an order of magnitude (12×). Additionally, in-memory Darwin is only 1.7× better than the persistent version, which is in line with the PMem–DRAM gap presented in the Viper paper (Chapter 4 and [14]). This shows that modern storage significantly closes the gap between volatile and existing durable state management, enabling us to efficiently support larger-than-memory state. Overall, our results show that Darwin achieves both state-of-the-art scale-up performance and an order of magnitude improvement over existing larger-than-memory scale-out systems.

5.6 Conclusion

To bridge the gap between performance and core features, we propose *scale-in* stream processing and present our prototype system Darwin. By achieving high hardware utilization on a wide range of hardware setups, scale-in systems adapt to application-specific demands. Combining scale-out concepts with advancements in persistent memory and scale-up concepts that focus on the underlying hardware, scale-in processing achieves high system utilization without sacrificing key features for industry adoption, such as recoverable, larger-than-memory state. Our scale-in SPE prototype Darwin uses storage-aware state management and query compilation to match state-of-the-art scale-up performance while outperforming existing scale-out systems by an order magnitude. Scale-in processing enables application-proportional scaling of server requirements, making it economical for all levels of performance needs.

6 What We Can Learn from Persistent Memory for CXL

Parts of this chapter have been published in [17].

6.1 Introduction

With the arrival of Intel Optane Persistent Memory (PMem) in 2019, research on new data management techniques for byte-addressable persistent memory increased significantly. Among other questions, this research investigates how to handle varying memory access latency, how to place data based on available capacity, and how to design for memory access sizes larger than a single cache line but smaller than a page [14, 101, 129]. However, in 2022, Intel announced that their Optane product line will be discontinued in favor of recent trends toward Compute Express Link (CXL) [41]. We expect that while Optane was abandoned, research based on it still provides valuable insights.

In light of Intel citing CXL as one of the reasons for ending Optane, in this chapter, we raise the question: "What can we learn from PMem research for future CXL research?". Based on benchmark results from PerMA-Bench (Chapter 3 and [15]), we look at three insights from PMem that also apply to future research on CXL. While CXL is an interconnect and not a memory technology, we believe that certain characteristics apply to both.

First, we discuss how limited hardware access can lead to solutions that are too specialized for one hardware configuration or not specialized enough. We then discuss how existing CPU components interact with new memory types, based on the prefetching behavior with PMem. Next, we show that different memory types offer different price-performance trade-offs depending on the use case. Finally, we discuss when the common PMem concepts of flushing data and ensuring global visibility through memory fences are needed in CXL-attached memory. Even though CXL-attached memory is not yet generally available, these insights highlight some challenges that future research faces when integrating new memory types into a long-established memory hierarchy.

6.2 Compute Express Link

In this section, we briefly provide some background on the Compute Express Link (CXL) interconnect [30]. The key idea of CXL is to provide cache-coherent memory between host CPUs and e.g., accelerators, smart network, and memory-expansion devices. It is based on the physical layer of PCI Express (PCIe), allowing users to attach common peripheral devices, such as FPGAs or GPUs. However, it also allows users to attach memory-expansion devices, i.e., devices that only provide additional memory. This circumvents the limited number of memory channels directly attached to a CPU, which is the restriction in current systems.

CXL defines three protocols, *CXL.io*, *CXL.mem*, and *CXL.cache*. CXL.io covers the same functionality as PCIe, i.e., it provides a non-coherent load/store interface for I/O devices [28, p. 71]. It builds the basis for the other protocols. CXL.cache allows attached devices to cache host memory [28, p. 83]. CXL.mem offers a transactional interface between the host CPU and memory [28, p. 110].

Based on these protocols, there are three device types in CXL[15]. Type 1 devices implement CXL.io and CXL.cache, i.e., these are devices that may cache host memory but do not offer any memory for the host to manage. Type 2 devices implement all three protocols. They are devices that have a cache and offer additional memory, e.g., accelerator cards with on-device memory. Type 3 devices implement CXL.io and CXL.mem, i.e., they are devices that only offer additional memory to the host.

There are currently three major versions of the CXL specification. Major features for its adoption in data centers are introduced in versions 2.0 and 3.0, e.g., memory pooling and switch-attached memory. Pooling allows multiple CPUs to coherently and dynamically manage memory from a memory device on the same machine. Switch-attached memory allows for coherent network-attached memory pooling in PCIe/CXL networks. This feature allows for large-scale memory disaggregation as it is possible to have a memory-only server from which memory is accessed remotely by other servers.

6.3 Transferring Insights from PMem to CXL-Attached Memory

In this section, we discuss how insights derived from PMem transfer to future CXL research.

Benchmark Setup. We evaluate two servers with 256 GB PMem DIMMs of the

15 For more details on the specific device types, we refer to the CXL specification [28].

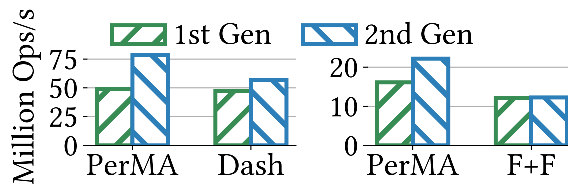

Figure 6.1: Lookup performance of PerMA compared to Dash and FAST+FAIR.

first and second-generation Optane. The first generation server contains a Cascade Lake CPU with 18 cores and six PMem DIMMs at 2933 MT/s. The second generation server contains an Ice Lake CPU with 32 cores and eight PMem DIMMs at 3200 MT/s. Both systems run Ubuntu 20.04 with a 5.4 kernel. We refer to Section 3.3.1 for more details on the server setup.

Application Tailoring

To understand how well applications utilize the hardware, we compare the `lookup()` performance of PMem index structures modeled in PerMA-Bench with the actual index implementations. The results obtained with PerMA-Bench show a performance upper-bound, as they include only memory access without computation or branching logic. The results for the hash index Dash [101] and the B-Tree index FAST+FAIR [58] are shown in Figure 6.1. The memory access for Dash is modeled as a 512-byte random read, as Dash reads two consecutive 256-byte hash buckets to find an entry. For FAST+FAIR, we issue three random 512-byte reads that represent B-Tree node lookups. The experiments are run with 16 threads. We observe that while the underlying bandwidth improves across generations, the indexes do not (fully) utilize this. Unlike on the first generation server, Dash spends ~20% of all cycles on non-memory access on the second generation server, which contains more PMem DIMMs, a newer CPU, and better DRAM. Due to the high price of Optane, researchers often do not have access to different setups, which leads to tailoring the application towards only a single setup and, in turn, does not always generalize. On the other hand, we see FAST+FAIR as an index designed for general PMem before Optane became available. The improved memory bandwidth does not translate to the index, as FAST+FAIR spends a lot of time on heavy-weight locking and inefficient PMem access patterns. As FAST+FAIR was designed pre-Optane, we see that the system is not tailored enough to the underlying hardware, and performance is lost.

Insight 1: Due to limited hardware availability, systems are tailored too much toward a single setup or not tailored enough toward the actual hardware. Through

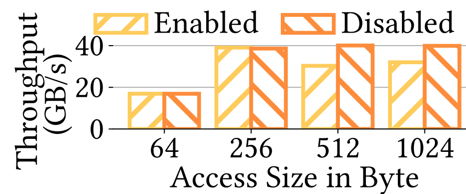

Figure 6.2: Impact of prefetcher on PMem random read bandwidth.

the CXL abstraction, future systems will cover a wider range of memory performance characteristics. Thus, it is important to design and research robust systems that generalize across different hardware and multiple memory tiers.

Prefetching

As a new layer in the long-established memory hierarchy, it is important to understand how well PMem interacts with existing CPU components, which are optimized for caches and DRAM. In Figure 6.2, we show the impact of the hardware prefetchers for random memory reads on the second-generation Optane server. We en-/disable all hardware prefetchers and run on 16 threads. We see that for small access sizes (< 256-byte), the prefetcher does not impact performance, i.e., the prefetcher does not prefetch. However, for 512 and 1024-byte access, the prefetcher speculatively loads unnecessary data not accessed by the user in the background, reducing the available bandwidth for requested reads. We observe this in hardware performance counters, where the underlying bandwidth utilization is identical in both runs, but the effective bandwidth in the application is not. Thus, disabling the prefetcher actually improves performance in this case. This effect is also observable for 2048-byte access but not for 4096-byte or more [31], as page-size access is a well-understood and optimized pattern in DRAM. As Optane's internal access occurs at 256-byte granularity, most applications design access in multiples of 256-byte. As a consequence, a 512-byte random access to Optane, e.g., a node lookup in FAST+FAIR, spans only two Optane "cache lines", which should not trigger prefetching. However, the prefetcher views these 512-byte as regular DRAM access, spanning eight consecutive cache lines, and starts prefetching for sequential access.

Insight 2: Prefetchers are highly optimized toward 64-byte DRAM cache line access, and CXL specifies 64-byte transfers in the transaction layer [28, p. 167]. However, CXL abstracts from the underlying device, i.e., it could support Optane PMem or other memory devices, and memory behind CXL may not be accessible in 64-byte granularity. As all CPU- and CXL-attached memory is available in the

Table 6.1: Price-performance of PMem and DRAM. Read/write values in €/GB/s. Calculation based on listing prices from dell.de in February 2022.

	€/GB capacity	seq. read	rnd. read	seq. write	rnd. write
PMem	**12.77**	**0.22**	**0.33**	**0.60**	2.12
DRAM	59.37	0.38	0.46	0.70	**0.91**

same unified virtual address space, prefetchers operate on both types of memory. Future research should investigate how existing components, like the prefetcher, interact with memory that is not attached directly to the CPU via memory channels and has different access characteristics. However, unlike Insight 1, this cannot be solved by applications alone and most likely requires hardware changes as well.

Price-Performance

In Table 6.1, we show the price-performance for basic sequential/random read/write access in PMem and DRAM on the same second-generation Optane server while disregarding persistence. The data access-related prices per GB of throughput are normalized to the device's price per GB to avoid including the higher price for larger capacity. We see that the price per GB capacity is significantly lower for the PMem DIMMs than for the high-end DRAM DIMMs. As PMem is not available in cloud vendors, we base our calculations on the list price on dell.de [135] in February 2022. Focusing on the relative scale between the listed prices rather than on the exact monetary values, for sequential access and random reads, we observe that PMem has a better price-performance ratio, as the bandwidth is often only 2–3× worse while the price per GB capacity is about 5× better. DRAM outperforms PMem only for 64-byte random writes, where PMem bandwidth is very low because of high write amplification.

Insight 3: For applications that do not require peak performance or persistence, PMem can be used as a cheaper DRAM alternative with significantly higher capacity. As increasing memory capacity is a selling point of CXL, future research should investigate the price-performance trade-off in multi-tier memory setups for slower and potentially cheaper CXL-attached memory.

Memory Fences and Data Visibility

Writing correct code for PMem requires the correct use of flushes and memory fences to ensure that data is persisted and globally visible (see Section 2.1.4). Similar to eADR, explicitly flushing is not necessary for CXL, as data in the caches is within

the cache-coherent domain of CXL. However, explicit memory fences are required for applications that share memory.

For example, given two servers s_1 and s_2 that access shared remote CXL-attached memory on server s_m. If s_1 modifies data, s_2 will observe the changes in a cache-coherent manner, as specified by CXL. If s_1 crashes, two scenarios can occur. Either the modified state was transferred to s_m or s_2, so the state is "persistent" from s_1's perspective, as it survived the crash. Alternatively, the data may still only reside on s_1, i.e., it is lost with the crash.

These scenarios are similar to a single node crash on PMem. Either the changes were globally visible (for eADR systems) or in PMem before the crash, i.e., they were flushed to a "persistent" medium,[16] or they are lost. CXL offers a Global Persistent Flush (GPF) since version 2.0 that ensures that data is flushed on power loss, similar to eADR with Optane [28, p. 581]. While this is originally added to support PMem in CXL, it also applies to regular memory. It is needed when the application semantics on s_1 assume remote memory to be persistent in that it survives a crash of s_1. Similar to the example in Section 2.1.4, it is important for applications to issue correct memory fences even for small modifications to shared data. If they do not, they may make invalid assumptions about state that did not survive the crash.

Memory fences are required with shared memory across processes on a single machine, regardless of CXL. However, PMem research has highlighted the difficulty of doing this correctly in numerous cases, even for threads within a single application [115, 116]. A key feature of CXL is memory pooling and cache-coherence across servers, so correctly identifying which modification must be guarded with memory fences becomes even more relevant in shared memory CXL setups.

Insight 4: With cache-coherent, remote memory in CXL, applications must consider that previously assumed volatile memory may now outlive a server crash. To handle this correctly, explicit memory fences are required for shared state, similar to existing approaches in PMem.

6.4 Conclusion

With PMem, various assumptions about the homogeneity of DRAM access have been disrupted, leading to new challenges and designs. In this chapter, we discussed how insights from these designs also apply to future CXL research. New CXL-based approaches need to focus on performance generalizability under initially limited hardware availability. They should consider how interaction with long-established components, such as prefetchers, impacts performance. In multi-tier memory

16 This does not need to be actual PMem. It must only appear to survive a crash.

setups, new designs should consider their economic viability as a key trade-off. And finally, remote shared memory setups require careful handling of memory visibility through fences to ensure correctness. While PMem is discontinued for now, we hope that future CXL work builds on these insights to establish a more general understanding of how systems interact with multi-tier memory.

7 Conclusion & Outlook

7.1 Conclusion

The performance characteristics of large-scale persistent memory challenged established assumptions about the storage hierarchy. With byte-addressable, random access similar to DRAM and persistence like secondary storage, PMem blurs the lines between two distinct layers of this hierarchy. Based on these characteristics, research found many ways to significantly improve the performance of database storage systems. In this thesis, we investigated the use of PMem for efficient state management. We posed the question:

> *With the emergence of a fundamentally different memory technology in the form of persistent memory, how do data management systems need to be designed to leverage its unique properties for efficient state management and how can we extend these insights to future disruptive memory technologies?*

Our answer to this question has multiple parts. In a first step, before designing systems for PMem, we must understand its performance. To this end, we performed an analysis of PMem performance across a wide range of server setups and configurations (Chapter 3). We identified eight general and implementation-specific aspects that influence PMem performance and highlight how they impact current and potential future designs.

Based on our understanding of PMem performance characteristics, we presented a hybrid PMem-DRAM key-value store called Viper in Chapter 4. Based on the performance of the individual memory technologies' access patterns, Viper splits storage and indexing into PMem and DRAM, respectively. With our PMem-optimized page layout, we showed that Viper achieves very high ingestion rates, outperforming other designs at the time, while offering full data persistence.

To leverage PMem's performance in new systems, we identified new design choices for stream processing engines in Chapter 5. We discussed limitations of current stream processing engines and proposed a new prototype engine, Darwin, that tailors its execution toward the underlying hardware. With our proposed design, we showed that Darwin can leverage the performance and persistence of PMem for state management in SPEs.

Even though PMem was discontinued, it is advantageous to apply knowledge gained from research on it to future memory technologies. In Chapter 6, we provided an outlook to multi-tier memory research based on insights gained throughput this thesis. We discussed how learnings from PMem research can be transferred to future multi-tier memory setups. We highlighted four challenges and showed how they apply to CXL-based systems in the future.

Concluding, in this thesis we have shown that PMem offers high performance for state management, bridging the gap between volatile but fast DRAM and persistent but slow secondary storage. Although Optane was discontinued, new multi-tier memory technology is emerging in various forms. By thoroughly investigating Optane's performance and building new systems based on it, we have shown how to incorporate such a new memory technology. By following our approach and insights gained throughout this thesis, future work can integrate future technology to improve state management for database systems.

7.2 Research Outlook

The introduction of PMem into the storage hierarchy led to a significant amount of research on how to integrate it into data management systems. Some work focused on the persistence, while other work investigated its use as high capacity DRAM with slightly lower access performance. Even with Optane discontinued and no current large-scale PMem alternatives, future research can still continue using Optane to investigate multi-tier memory setups.

As outlined in Chapter 6, various properties of Optane will likely apply to CXL-memory, e.g., unexpected interaction with existing CPU components, such as the prefetcher, or required memory fences for correctness. The exact characteristics of emerging CXL-attached memory are not yet known, but we have shown that even within the same product line of Optane, performance differs significantly between setups. As a solid understanding of performance in such complex, multi-tier memory setups is essential to design efficient systems, we are extending the PerMA-Bench framework in ongoing work to support such setups.

Beyond the high capacity of current PMem, novel *persistent* designs remain an interesting area of research. Optane was mainly discontinued for economical reasons [41], with various factors limiting its adoption. Optane PMem required high-end Intel CPUs because of the modified DDR-T protocol. By removing the necessity of attaching PMem directly to memory channels, opportunities arise for other manufacturers to produce and distribute PMem. Following an open standard, such as CXL, it becomes possible to combine Optane or other PMem with non-Intel

CPUs behind a standard PCIe interface. The economics of such setups may be significantly different to Optane–Intel setups, making it more viable to add PMem to the storage stack. In this case, future work should investigate the viability of existing designs on new hardware to come up with robust and general solution for PMem.

Overall, regardless of the exact outcome of PMem as a technology, we believe its introduction and disruption of the traditional storage hierarchy has led to a re-evaluation of the memory-storage divide for data management systems. Alternative memory technologies will continue to emerge, challenging how and where we efficiently store and process data.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. **The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing**. *Proceedings of the VLDB Endowment* 8:12 (Aug. 2015). 1792–1803. DOI: 10.14778/2824032.2824076 (cited on page 85).

[2] Paul Alcorn. *Intel Kills Optane Memory Business.* https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good. 2022 (cited on page 3).

[3] Paul Alcorn. *Intel Optane DIMM Pricing.* https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html. 2020 (cited on page 75).

[4] Alibaba. *Elastic Compute Service.* 2021. URL: www.alibabacloud.com/product/ecs (cited on page 83).

[5] Alibaba. *Four Billion Records per Second!* 2020. URL: www.alibabacloud.com/blog/four-billion-records-per-second-stream-batch-integration-implementation-of-alibaba-cloud-realtime-compute-for-apache-flink-during-double-11_596962 (cited on page 83).

[6] Joy Arulraj and Andrew Pavlo. **How to Build a Non-Volatile Memory Database Management System**. In: *Proceedings of the 2017 ACM International Conference on Management of Data.* 2017, 1753–1758. DOI: 10.1145/3035918.3054780 (cited on page 2).

[7] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. **Let's talk about storage & recovery methods for non-volatile memory database systems**. In: *SIGMOD '15.* 2015, 707–722. DOI: 10.1145/2723372.2749441 (cited on pages 23, 51, 54).

[8] Joy Arulraj, Matthew Perron, and Andrew Pavlo. **Write-behind logging**. *Proceedings of the VLDB Endowment* 10:4 (2016). 337–348. DOI: 10.14778/3025111.3025116 (cited on page 51).

[9] Jens Axboe. *fio: Flexible I/O Tester.* 2022. URL: https://github.com/axboe/fio (cited on page 51).

[10] I.G. Baek, M.S. Lee, S. Sco, M.J. Lee, D.H. Seo, D.-S. Suh, J.C. Park, S.O. Park, H.S. Kim, I.K. Yoo, U.-I. Chung, and J.T. Moon. **Highly scalable non-volatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses**. In: *IEDM Technical Digest. IEEE International Electron Devices Meeting*. 2004. DOI: 10.1109/iedm.2004.1419228 (cited on pages 3, 12).

[11] R. Bayer and E. M. McCreight. **Organization and maintenance of large ordered indexes**. *Acta Informatica* 1:3 (Sept. 1, 1972). 173–189. DOI: 10.1007/BF00288683 (cited on page 1).

[12] Lawrence Benson, Richard Ebeling, and Tilmann Rabl. **Evaluating SIMD Compiler-Intrinsics for Database Systems**. In: *Workshop on Accelerating Analytics and Data Management Systems (ADMS'23)*. 2023.

[13] Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, and Tilmann Rabl. **Disco: Efficient Distributed Window Aggregation**. In: *EDBT '20*. 2020 (cited on page 86).

[14] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. **Viper: An Efficient Hybrid PMem-DRAM Key-Value Store**. *Proceedings of the VLDB Endowment* 14:9 (2021). 1544–1556. DOI: 10.14778/3461535.3461543 (cited on pages 5, 11, 23, 28, 37, 42, 51, 53, 88, 90, 97–99).

[15] Lawrence Benson, Leon Papke, and Tilmann Rabl. **PerMA-bench: benchmarking persistent memory access**. *Proceedings of the VLDB Endowment* 15:11 (July 2022). 2463–2476. DOI: 10.14778/3551793.3551807 (cited on pages 2, 5, 11, 23, 99).

[16] Lawrence Benson and Tilmann Rabl. **Darwin: Scale-In Stream Processing**. In: *12th Conference on Innovative Data Systems Research, CIDR 2022*. 2022 (cited on pages 6, 83).

[17] Lawrence Benson, Marcel Weisgut, and Tilmann Rabl. **What We Can Learn from Persistent Memory for CXL**. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2023)*. Vol. P-331. 2023, 757–761. DOI: 10.18420/BTW2023-48 (cited on pages 6, 99).

[18] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. **Analyzing Vectorized Hash Tables across CPU Architectures**. *Proceedings of the VLDB Endowment* 16:11 (Aug. 24, 2023). 2755–2768. DOI: 10.14778/3611479.3611485.

[19] Maximilian Böther, Otto Kißig, Lawrence Benson, and Tilmann Rabl. **Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs**. In: *DaMoN '21*. 2021. DOI: 10.1145/3465998.3466010 (cited on pages 2, 4, 51).

[20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. **Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook**. In: *FAST '20*. 2020, 209–223 (cited on pages 67, 69, 72, 75).

[21] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. **Apache Flink(TM): Stream and Batch Processing in a Single Engine**. *IEEE Data Eng. Bull.* 38:4 (2015). 28–38 (cited on pages 4, 53, 87, 97).

[22] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. **Cutty: Aggregate Sharing for User-Defined Windows**. In: *CIKM*. 2016, 1201–1210. DOI: 10.1145/2983323.2983807 (cited on page 86).

[23] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. **Trill: a high-performance incremental query processor for diverse analytics**. *PVLDB* 8:4 (Dec. 2014). 401–412. DOI: 10.14778/2735496.2735503 (cited on page 87).

[24] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. **FASTER: A Concurrent Key-Value Store with In-Place Updates**. In: *SIGMOD '18*. 2018, 275–290. DOI: 10.1145/3183713.3196898 (cited on pages 55, 56, 69, 81, 91).

[25] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. **uTree: a persistent B+-tree with low tail latency**. *Proceedings of the VLDB Endowment* 13:12 (2020). 2634–2648. DOI: 10.14778/3407790.3407850 (cited on pages 23, 28, 61, 66, 70, 81).

[26] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. **Flat-Store: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory**. In: *ASPLOS '20*. 2020, 1077–1091. DOI: 10.1145/3373376.3378515 (cited on pages 2–4, 23, 51, 81).

[27] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. **Lock-free Concurrent Level Hashing for Persistent Memory**. In: *ATC '20*. 2020, 799–812 (cited on page 51).

[28] Inc. Compute Express Link Consortium. *Compute Express Link (CXL) Specification, Revision 3.0, Version 1.0*. https://www.computeexpresslink.org/download-the-specification. 2022 (cited on pages 100, 102, 104).

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. **Benchmarking cloud serving systems with YCSB**. In: *SoCC '10*. 2010, 143–154. DOI: 10.1145/1807128.1807152 (cited on page 79).

[30] CXL Consortium. *Compute Express Link: The Breakthrough CPU-to-Device Interconnect CXL*. 2022. URL: https://www.computeexpresslink.org/ (cited on pages 4, 18, 100).

[31] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. **Maximizing persistent memory bandwidth utilization for OLAP workloads**. In: *SIGMOD '21*. 2021. DOI: https://doi.org/10.1145/3448016.3457292 (cited on pages 16, 23, 30, 32, 33, 44, 51, 54, 84, 102).

[32] Benoit Dageville, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, and Martin Hentschel. **The Snowflake Elastic Data Warehouse**. In: *SIGMOD '16*. 2016, 215–226. DOI: 10.1145/2882903.2903741 (cited on page 53).

[33] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. **Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines**. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, 2471–2486. DOI: 10.1145/3318464.3389723 (cited on page 86).

[34] Wolfgang Effelsberg and Theo Haerder. **Principles of database buffer management**. *ACM Transactions on Database Systems* 9:4 (Dec. 5, 1984). 560–595. DOI: 10.1145/1994.2022 (cited on page 1).

[35] Ahmed Eldawy, Justin Levandoski, and Per Ake Larson. **Trekking through Siberia: Managing cold data in a memory-optimized database**. *Proceedings of the VLDB Endowment* 7:11 (2014). 931–942. DOI: 10.14778/2732967.2732968 (cited on page 1).

[36] Hewlett Packard Enterprise. *HPE NVDIMMs Memory – Overview*. 2021. URL: https://support.hpe.com/hpesc/public/docDisplay?docId=c05302373 (cited on page 11).

[37] Hewlett Packard Enterprise. *Server memory and persistent memory population rules for HPE Gen10 servers with Intel Xeon Scalable processors technical white paper*. 2021. URL: https://www.hpe.com/psnow/doc/a00017079enw (cited on page 46).

[38] Facebook. *RocksDB*. https://rocksdb.org. 2020 (cited on pages 1, 4, 53, 55, 56, 63, 67, 80).

[39] Apache Flink. *Table API & SQL*. 2021. URL: ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/table/overview (cited on page 95).

[40] Apache Flink. *Improvement in (de)serialization of keys and values for RocksDB state*. https://issues.apache.org/jira/browse/FLINK-9702. 2020 (cited on page 53).

[41] Pat Gelsinger and Dave Zinsner. *Earnings Call Comments from CEO Pat Gelsinger and CFO Dave Zinsner*. https://download.intel.com/newsroom/2022/corporate/Intel-CEO-CFO-2Q22-earnings-statements.pdf. 2022 (cited on pages 3, 4, 99, 108).

[42] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. **Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile**. *arXiv:2103.10169 [cs]* (Mar. 18, 2021) (cited on page 87).

[43] Google. *LevelDB, a fast key-value storage library*. https://code.google.com/p/leveldb. 2020 (cited on pages 53, 63, 67, 80).

[44] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. **Data structure primitives on persistent memory: an evaluation**. In: *DaMoN '20*. 2020, 14:1–14:3. DOI: 10.1145/3399666.3399900 (cited on pages 53, 54).

[45] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. **Grizzly: Efficient Stream Processing Through Adaptive Query Compilation**. In: *SIGMOD '20*. 2020, 2487–2503. DOI: 10.1145/3318464.3389739 (cited on pages 83, 87, 88, 90, 91, 93, 96, 97).

[46] Tim Gubner and Peter Boncz. **Charting the design space of query execution using VOILA**. *PVLDB* 14:6 (2021). 1067–1079. DOI: 10.14778/3447689.3447709 (cited on page 94).

[47] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. **Understanding the idiosyncrasies of real persistent memory**. *Proceedings of the VLDB Endowment* 14:4 (2020). 626–639. DOI: 10.14778/3436905.3436921 (cited on pages 3, 23, 30, 51).

[48] Xiaochen Guo, Engin Ipek, and Tolga Soyata. **Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing**. In: *ISCA '10*. 2010, 371–382. DOI: 10.1145/1815961.1816012 (cited on pages 3, 12).

[49] Gabriel Haas, Michael Haubenschild, and Viktor Leis. **Exploiting Directly-Attached NVMe Arrays in DBMS**. In: *CIDR '20*. 2020 (cited on page 2).

[50] Gabriel Haas and Viktor Leis. **What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines**. *Proceedings of the VLDB Endowment* 16:9 (July 10, 2023). 2090–2102. DOI: 10.14778/3598581.3598584 (cited on page 15).

[51] HdrHistogram. *HdrHistogram: A high dynamic range histogram*. http://hdrhistogram.org. 2020 (cited on page 79).

[52] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. **Evaluating Persistent Memory Range Indexes: Part Two**. *arXiv:2201.13047 [cs]* (Jan. 31, 2022) (cited on pages 39, 42).

[53] Uwe Heinz. *SAP HANA and Persistent Memory*. 2020. URL: https://blogs.sap.com/2020/01/30/sap-hana-and-persistent-memory (cited on page 12).

[54] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. **Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery**. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, 1049–1063. DOI: 10.1145/3514221.3517884 (cited on page 7).

[55] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. **Persistent Memory Hash Indexes: An Experimental Evaluation**. *Proceedings of the VLDB Endowment* 14:5 (2021). 785–798 (cited on page 72).

[56] Kaisong Huang, Tianzheng Wang, Darien Imai, and Dong Xie. **SSDs Striking Back:The Storage Jungle and Its Implications on Persistent Indexes**. In: *CIDR '22*. 2022, 1–8 (cited on pages 15, 51).

[57] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. **Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs**. In: *ATC '18*. 2018, 967–979 (cited on page 70).

[58] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. **Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree**. In: 16th USENIX Conference on File and Storage Technologies (FAST 18). 2018, 187–200 (cited on pages 2, 39, 41, 101).

[59] Intel. *Ice Lake Hardware Events*. 2023. URL: https://perfmon-events.intel.com/icelake.html (cited on page 41).

[60] Intel. *Intel Optane DC Persistent Memory Product Brief*. 2019. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf (cited on pages 11, 14, 15, 30–32).

[61] Intel. *Intel Optane Persistent Memory 200 Series Brief*. 2020. URL: https://www.intel.de/content/www/de/de/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html (cited on pages 14, 15, 20, 30).

[62] Intel. *Intel Xeon Processors*. 2021. URL: https://www.intel.com/content/www/us/en/products/details/processors/xeon.html (cited on page 17).

[63] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/253665-sdm-vol-1.pdf (cited on pages 19, 29).

[64] Intel. *Intel® Cascade Lake*. 2019. URL: https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html (cited on page 14).

[65] Intel. *Intel® Ice Lake SP*. 2019. URL: https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html (cited on page 14).

[66] Intel. *Intel® Optane×2122 Persistent Memory 300 Series Brief*. 2022. URL: https://www.colfax-intl.com/downloads/intel-optane-persistent-memory-300-series-brief.pdf (cited on pages 14, 15).

[67] Intel. *Intel® Sapphire Rapids*. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html (cited on page 14).

[68] Intel. *Optimizing Write Ahead Logging with Intel Optane Persistent Memory*. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/optimizing-write-ahead-logging-with-intel-optane-persistent-memory.html (cited on pages 27, 30).

[69] Intel. *Intel Optane Persistent Memory*. https://intel.com/optanedcpersistentmemory. 2020 (cited on pages 1, 54, 58, 75).

[70] Intel. *TBB concurrent hash map*. https://software.intel.com/en-us/node/506077. 2020 (cited on page 70).

[71] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. **Basic Performance Measurements of the Intel Optane DC Persistent Memory Module**. *arXiv:1903.05714 [cs]* (Aug. 9, 2019) (cited on pages 30, 51, 53, 56, 81).

[72] JEDEC. *Byte Addressable Energy Backed Interface*. 2020. URL: https://www.jedec.org/standards-documents/docs/jesd245a (cited on page 12).

[73] JEDEC. *DDR4 NVDIMM-P Bus Protocol*. 2021. URL: https://www.jedec.org/standards-documents/docs/jesd304-401 (cited on page 12).

[74] JEDEC. *JEDEC Publishes DDR4 NVDIMM-P Bus Protocol Standard*. 2021. URL: https://www.jedec.org/news/pressreleases/jedec-publishes-ddr4-nvdimm-p-bus-protocol-standard (cited on page 12).

[75] JEDEC. *Memory Configurations: JESD21-C*. 2023. URL: https://www.jedec.org/category/technology-focus-area/memory-configurations-jesd21-c (cited on page 18).

[76] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. **SplitFS: reducing software overhead in file systems for persistent memory**. In: *SOSP '19*. 2019, 494–508. DOI: 10.1145/3341301.3359631 (cited on page 51).

[77] Vasiliki Kalavri and John Liagouris. **In support of workload-aware streaming state management**. In: 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). 2020 (cited on page 88).

[78] Anuj Kalia, David Andersen, and Michael Kaminsky. **Challenges and solutions for fast remote persistent memory access**. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, 105–119. DOI: 10.1145/3419111.3421294 (cited on page 37).

[79] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. **Benchmarking Distributed Stream Processing Engines**. In: *ICDE*. 2018, 1507–1518 (cited on page 94).

[80] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. **SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures**. In: *SIGMOD '16*. 2016, 555–569. DOI: 10.1145/2882903.2882906 (cited on pages 83, 87, 88).

[81] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. **NVM: Is it Not Very Meaningful for Databases?** *Proceedings of the VLDB Endowment* 16:10 (Aug. 8, 2023). 2444–2457. DOI: 10.14778/3603581.3603586 (cited on page 4).

[82] Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, and Gustavo Alonso. **How to use Persistent Memory in your Database**. *arXiv:2112.00425 [cs]* (Dec. 1, 2021) (cited on page 46).

[83] Jay Kreps, Neha Narkhede, and Jun Rao. **Kafka: a Distributed Messaging System for Log Processing**. In: *NetDB*. 2011, 1–7 (cited on page 85).

[84] Avinash Lakshman and Prashant Malik. **Cassandra: a decentralized structured storage system**. *ACM SIGOPS Operating Systems Review* 44:2 (Apr. 14, 2010). 35–40. DOI: 10.1145/1773912.1773922 (cited on pages 67, 80).

[85] Per-Ake Larson. **Dynamic hash tables**. *Communications of the ACM* 31:4 (Apr. 1, 1988). 446–457. DOI: 10.1145/42404.42410 (cited on page 58).

[86] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. **Architecting phase change memory as a scalable dram alternative**. In: *ISCA '09*. 2009. DOI: 10.1145/1555754.1555758 (cited on pages 3, 12).

[87] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. **Recipe: converting concurrent DRAM indexes to persistent-memory indexes**. In: *SOSP '19*. 2019, 462–477. DOI: 10.1145/3341301.3359635 (cited on page 51).

[88] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. **Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age**. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 2014, 743–754. DOI: 10.1145/2588555.2610507 (cited on page 26).

[89] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. **LeanStore: In-Memory Data Management beyond Main Memory**. In: *ICDE '18*. 2018, 185–196. DOI: 10.1109/ICDE.2018.00026 (cited on pages 1, 88, 90).

[90] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. **Evaluating persistent memory range indexes**. *Proceedings of the VLDB Endowment* 13:4 (Dec. 9, 2019). 574–587. DOI: 10.14778/3372716.3372728 (cited on pages 43, 67, 76, 81).

[91] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. **Enabling low tail latency on multicore key-value stores**. *Proceedings of the VLDB Endowment* 13:7 (Mar. 1, 2020). 1091–1104. DOI: 10.14778/3384345.3384356 (cited on pages 2, 4, 23, 51, 53, 55, 81, 94).

[92] 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)*. 2013, 302–313. DOI: 10.1109/ICDE.2013.6544834 (cited on page 1).

[93] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. **No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams**. *ACM SIGMOD Record* 34:1 (2005). 39–44 (cited on page 86).

[94] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. **Algorithmic improvements for fast concurrent Cuckoo hashing**. In: *EuroSys '14*. 2014, 1–14. DOI: 10.1145/2592798.2592820 (cited on page 58).

[95] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. **MICA: a holistic approach to fast in-memory key-value storage**. In: *NSDI' 14*. 2014, 429–444 (cited on page 80).

[96] Hao Liu, Linpeng Huang, Yanmin Zhu, and Yanyan Shen. **LibreKV: A Persistent In-Memory Key-Value Store**. *IEEE Transactions on Emerging Topics in Computing* (2017). 1–1. DOI: 10.1109/TETC.2017.2787341 (cited on page 81).

[97] Jihang Liu, Shimin Chen, and Lujun Wang. **LB+Trees: optimizing persistent index performance on 3DXPoint memory**. *Proceedings of the VLDB Endowment* 13:7 (Mar. 1, 2020). 1078–1090. DOI: 10.14778/3384345.3384355 (cited on pages 2, 3, 23, 28, 35, 42, 46, 51, 81).

[98] Ruicheng Liu, Peiquan Jin, Xiaoliang Wang, Zhou Zhang, Shouhong Wan, and Bei Hua. **NVLevel: A High Performance Key-Value Store for Non-Volatile Memory**. In: *HPCC/SmartCity/DSS '19*. 2019, 1020–1027. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00146 (cited on pages 54, 81).

[99] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. **PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs**. In: *ASPLOS '19*. 2019, 411–425. DOI: 10.1145/3297858.3304015 (cited on page 19).

[100] Zhiye Liu. *Fujitsu Targets 2019 for NRAM Mass Production*. 2018. URL: https://www.tomshardware.com/news/fujitsu-nram-nantero-carbon-nanotube,37437.html (cited on pages 3, 12).

[101] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. **Dash: scalable hashing on persistent memory**. *Proceedings of the VLDB Endowment* 13:8 (Apr. 1, 2020). 1147–1161. DOI: 10.14778/3389133.3389134 (cited on pages 3, 23, 28, 35, 37, 39, 40, 51, 54, 58, 60, 61, 64, 66, 70, 81, 99, 101).

[102] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. **WiscKey: Separating Keys from Values in SSD-Conscious Storage**. *ACM Transactions on Storage* 13:1 (Mar. 2, 2017). 5:1–5:28. DOI: 10.1145/3033273 (cited on pages 67, 80).

[103] Kelly Lyon. *How Intel Optimized RocksDB Code for Persistent Memory with PMDK*. https://software.intel.com/content/www/us/en/develop/articles/how-intel-optimized-rocksdb-code-for-persistent-memory-with-pmdk.html. 2021 (cited on page 69).

[104] Kazuaki Maeda. **Performance evaluation of object serialization libraries in XML, JSON and binary formats**. In: *DICTAP '12*. 2012, 177–182. DOI: 10.1109/DICTAP.2012.6215346 (cited on page 53).

[105] Anton Malakhov. **Per-bucket concurrent rehashing algorithms**. *arXiv:1509.02235 [cs]* (Sept. 7, 2015) (cited on page 58).

[106] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. **Evaluating Multi-GPU Sorting with Modern Interconnects**. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, 1795–1809. DOI: 10.1145/3514221.3517842 (cited on page 51).

[107] Tobias Maltenberger, Till Lehmann, Lawrence Benson, and Tilmann Rabl. **Evaluating In-Memory Hash Joins on Persistent Memory**. In: *EDBT '22*. 2022, 2:368–2:372. DOI: 10.48786/edbt.2022.23.

[108] Mellanox. *200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC*. 2021. URL: www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic (cited on page 94).

[109] Memcached. *Memcached, high-performance, distributed memory object caching system*. https://memcached.org/. 2020 (cited on pages 55, 80).

[110] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. **CaSSanDra: An SSD boosted key-value store**. In: *ICDE '14*. 2014, 1162–1167. DOI: 10.1109/ICDE.2014.6816732 (cited on page 80).

[111] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn Mckinley, and Felix Xiaozhu Lin. **StreamBox: Modern Stream Processing on a Multicore Machine**. In: *ATC*. 2017, 617–629 (cited on pages 87, 90, 91).

[112] mmap. *mmap(2): map/unmap files/devices into memory - Linux man page*. https://linux.die.net/man/2/mmap. 2020 (cited on page 60).

[113] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. **Write-optimized dynamic hashing for persistent memory**. In: *FAST '19*. 2019, 31–44 (cited on pages 51, 58, 69).

[114] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. **Dalí: A Periodically Persistent Hash Map**. In: *DISC '17*. 2017, 37:1–37:16. DOI: 10.4230/LIPIcs.DISC.2017.37 (cited on pages 54, 58).

[115] Ian Neal, Andrew Quinn, and Baris Kasikci. **Hippocrates: healing persistent memory bugs without doing any harm**. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, 401–414. DOI: 10.1145/3445814.3446694 (cited on page 104).

[116] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. **AGAMOTTO: How Persistent is your Persistent Memory Application?** In: *OSDI '20*. 2020, 1047–1064 (cited on pages 20, 104).

[117] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. **Rethinking File Mapping for Persistent Memory**. In: 19th USENIX Conference on File and Storage Technologies (FAST 21). 2021, 97–111 (cited on page 51).

[118] Thomas Neumann. **Efficiently compiling efficient query plans for modern hardware**. *PVLDB* 4:9 (June 1, 2011). 539–550. DOI: 10.14778/2002938.2002940 (cited on pages 93, 96).

[119] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. **The LRU-K page replacement algorithm for database disk buffering**. *ACM SIGMOD Record* 22:2 (June 1, 1993). 297–306. DOI: 10.1145/170036.170081 (cited on page 1).

[120] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. **The log-structured merge-tree (LSM-tree)**. *Acta Informatica* 33:4 (June 1, 1996). 351–385. DOI: 10.1007/s002360050048 (cited on page 53).

[121] Matt Ogle, Trent Bates, Bruce Wagner, and Rene Franco. *How to Balance Memory on 2nd Generation Intel Xeon Scalable Processors*. 2021. URL: https://downloads.dell.com/manuals/common/balancing_memory_xeon_2nd_gen.pdf (cited on page 47).

[122] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. **SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery**. In: *DaMoN '14*. 2014, 1–7. DOI: 10.1145/2619228.2619236 (cited on page 51).

[123] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. **FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for Storage Class Memory**. In: *SIGMOD '16*. 2016, 371–386. DOI: 10.1145/2882903.2915251 (cited on pages 2, 3, 23, 28, 35, 37, 39, 42, 51, 54, 56, 60, 61, 64, 70, 81).

[124] PMDK. *Persistent memory programming*. https://pmem.io/pmdk. 2020 (cited on pages 20, 69, 70, 82).

[125] PmemKV. *pmemkv, key/value datastore for persistent memory*. https://pmem.io/pmemkv. 2020 (cited on page 70).

[126] PmemObj++. *libpmemobj++ concurrent hash map*. https://github.com/pmem/libpmemobj-cpp. 2020 (cited on page 70).

[127] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. **Scalable high performance main memory system using phase-change memory technology**. In: *ISCA '09*. 2009. DOI: 10.1145/1555754.1555760 (cited on pages 3, 12).

[128] Redis. *Redis, an in-memory data structure store*. https://redis.io. 2020 (cited on pages 4, 53, 55, 80).

[129] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. **Managing Non-Volatile Memory in Database Systems**. In: *SIGMOD '18*. 2018, 1541–1555. DOI: 10.1145/3183713.3196897 (cited on pages 2, 51, 54, 81, 99).

[130] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. **Building blocks for persistent memory**. *The VLDB Journal* (Sept. 23, 2020). DOI: 10.1007/s00778-020-00622-9 (cited on pages 23, 30, 33, 67, 76).

[131] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. **NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories**. In: *IMDM '15*. 2015, 4:1–4:8. DOI: 10.1145/2803140.2803144 (cited on page 58).

[132] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. **Building workload-independent storage with VT-trees**. In: *FAST '13*. 2013, 17–30 (cited on page 80).

[133] Julian Shun and Guy E. Blelloch. **Phase-concurrent hash tables for determinism**. In: *SPAA '14*. 2014, 96–107. DOI: 10.1145/2612669.2612687 (cited on page 58).

[134] SNIA. *NVM Programming Model (NPM)*. 2021. URL: https://www.snia.org/tech_activities/standards/curr_standards/npm (cited on pages 11, 15).

[135] Dell Technologies. *Dell Rack Servers*. 2022. URL: https://www.dell.com/de-de/work/shop/deals/enterprise-deals/poweredge-rack-server-deals (cited on pages 47, 103).

[136] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. **DFI: The Data Flow Interface for High-Speed Networks**. In: *SIGMOD '21*. 2021, 1825–1837. DOI: 10.1145/3448016.3452816 (cited on page 94).

[137] Jonas Traub, Philipp Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. **Efficient Window Aggregation with General Stream Slicing**. In: *EDBT*. 2019, 97–108 (cited on page 86).

[138] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. **Persistent memory I/O primitives**. In: *DaMoN '19*. 2019, 12:1–12:7. DOI: 10.1145/3329785.3329930 (cited on pages 3, 37, 51, 53, 81).

[139] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. **Drizzle: Fast and Adaptable Stream Processing at Scale**. In: *SOSP '17*. 2017, 374–389. DOI: 10.1145/3132747.3132750 (cited on page 87).

[140] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. **Plush: a write-optimized persistent log-structured hash-table**. *Proceedings of the VLDB Endowment* 15:11 (July 1, 2022). 2895–2907. DOI: 10.14778/3551793.3551839 (cited on page 7).

[141] Haris Volos, Andres Jaan Tack, and Michael M. Swift. **Mnemosyne: lightweight persistent memory**. *ACM SIGARCH Computer Architecture News* 39:1 (Mar. 5, 2011). 91–104. DOI: 10.1145/1961295.1950379 (cited on pages 23, 51).

[142] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. **Pacman: An Efficient Compaction Approach for {Log-Structured} {Key-Value} Store on Persistent Memory**. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22). 2022, 773–788 (cited on page 7).

[143] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. **Easy Lock-Free Indexing in Non-Volatile Memory**. In: *ICDE '18*. 2018, 461–472. DOI: 10.1109/ICDE.2018.00049 (cited on page 61).

[144] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. **Lessons learned from the early performance evaluation of Intel optane DC persistent memory in DBMS**. In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 2020, 1–3. DOI: 10.1145/3399666.3399898 (cited on page 46).

[145] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. **HiKV: a hybrid index key-value store for DRAM-NVM memory systems**. In: *ATC '17*. 2017, 349–362 (cited on pages 2, 54, 81).

[146] Jian Xu and Steven Swanson. **NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories**. In: *FAST '16*. 2016, 323–338 (cited on page 51).

[147] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. **An Empirical Guide to the Behavior and Use of Scalable Persistent Memory**. In: *FAST '20*. 2020, 169–182 (cited on pages 2, 3, 23, 30, 33, 51, 53, 54, 56, 58, 80, 81, 84).

[148] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. **NV-Tree: reducing consistency cost for NVM-based single level systems**. In: *FAST '15*. 2015, 167–181 (cited on pages 2, 3, 28, 35, 54, 56, 70, 81).

[149] Wang Yue, Lawrence Benson, and Tilmann Rabl. **Desis: Efficient Window Aggregation in Decentralized Networks**. In: *Proceedings 26th International Conference on Extending Database Technology (EDBT '23)*. 2023, 618–631. DOI: 10.48786/edbt.2023.52.

[150] Matei Zaharia, Scott Shenker, Haoyuan Li, Tathagata Das, Timothy Hunter, and Ion Stoica. **Discretized streams: fault-tolerant streaming computation at scale**. In: *SOSP '13*. 2013, 423–438. DOI: 10.1145/2517349.2522737 (cited on pages 4, 53, 87).

[151] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. **Analyzing Efficient Stream Processing on Modern Hardware**. *Proceedings of the VLDB Endowment* 12:5 (2019). 516–530. DOI: 10.14778/3303753.3303758 (cited on pages 83, 88, 94).

[152] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. **Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors**. In: *ICDE '17*. 2017, 659–670. DOI: 10.1109/ICDE.2017.119 (cited on pages 83, 88, 94).

[153] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. **BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures**. In: *SIGMOD '19*. 2019, 705–722. DOI: 10.1145/3299869.3300067 (cited on pages 83, 87, 88).

[154] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. **A durable and energy efficient main memory using phase change memory technology**. In: *ISCA '09*. 2009. DOI: 10.1145/1555754.1555759 (cited on pages 3, 12).

[155] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. **Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory**. In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, 2195–2207. DOI: 10.1145/3448016.3452819 (cited on page 2).

[156] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. **DPTree: differential indexing for persistent memory**. *Proceedings of the VLDB Endowment* 13:4 (Dec. 9, 2019). 421–434. DOI: 10.14778/3372716.3372717 (cited on pages 35, 51, 81).