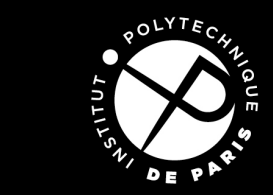
A Support for Persistent Memory in Java

Anatole Lefort



INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAS001

Thèse de doctorat

TELECOM SudParis

IP PARIS

A support for Persistent Memory in Java

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 24 mars 2023, par

M. ANATOLE LEFORT

Composition du Jury :

Sara Bouchenak Professeur, INSA Lyon (LIRIS)	Présidente
Paolo Romano Associate Professor, Instituto Superior Técnico, University of Lisbon	Rapporteur
Vivien Quema Professeur, Grenoble INP/ENSIMAG (LIG)	Rapporteur
Panagiota Fatourou Professor, FORTH ICS & University of Crete, CSD	Examinatrice
Pierre Sutra Maître de conférences, Télécom SudParis (SAMOVAR)	Encadrant de thèse
Gaël Thomas Professeur, Télécom SudParis (SAMOVAR)	Directeur de thèse

Anatole Lefort

A Support for Persistent Memory in Java

Directed & supervised by:
Gaël Thomas*
Pierre Sutra*

*Télécom SudParis - Institut Polytechnique de Paris
Palaiseau 91120, France
2022

To my loved ones.

Abstract

Recently released non-volatile main memory (NVMM), as fast and durable memory, dramatically increases storage performance over traditional media (SSD, hard disk). A substantial and unique property of NVMM is byte-addressability – complex memory data structures, maintained with regular load/store instructions, can now resist machine power-cycles, software faults or system crashes. However, correctly managing persistence with the fine grain of memory instructions is laborious, with increased risk of compromising data integrity and recovery at any misstep. Programming abstractions from software libraries and support from language runtime and compilers are necessary to avoid memory bugs that are exacerbated with persistence.

In this thesis, we address the challenges of supporting persistent memory in managed language environments by introducing J-NVM, a framework to efficiently access NVMM in Java. With J-NVM, we demonstrate how to design an efficient, simple and complete interface to weave NVMM-devised persistence into object-oriented programming, while remaining unobtrusive to the language runtime itself. In detail, J-NVM offers a fully-fledged interface to persist plain Java objects using failure-atomic sections. This interface relies internally on proxy objects that intermediate direct off-heap access to NVMM. The framework also provides a library of highly-optimized persistent data types that resist reboots and power failures. We evaluate J-NVM by implementing a persistent backend for Infinispan, an industrial-grade data store. Our experimental results, obtained with a TPC-B like benchmark and YCSB, show that J-NVM is consistently faster than other approaches at accessing NVMM in Java.

Acknowledgments

Oftentimes, a Ph.D. is referred to as a long and strenuous endeavor, requiring sustained self-discipline and loads of dedication; or for the very least, that is what I heard of it before embarking into one. My Past Self will hate me for admitting it, as I now feel I understand that being true for the most part. The tortuousness of the path, the hardships that run along; all seemingly increasing as one drags forth.

The predicament of a Ph.D. also entails that, this thesis – the final step towards completion – is of my own writing only and relates of work that had to be produced by a sole individual. Even so, that does certainly not mean no one else during those four years of tedious work was neither *critical*, nor *enabling* to me carrying out this work. I wish to hereby acknowledge all who have helped me reaching that milestone.

I would first like to thank the members of my thesis committee, for taking of their time to read and evaluate this document. Panagiota Fatourou, Sara Bouchenak, Paolo Romano and Vivien Quéma; thank you all for the interest you showed in this work. It was nice meeting you, answering your questions; I am glad you accepted to be part of this jury. I would like to acknowledge as well the members of my midterm evaluation committee: Julien Sopena and Alain Tchana, for the valuable feedback and insights they provided.

My advisors, Pierre Sutra and Gaël Thomas, deserve a profound thank-you for their constant support, benevolence and counsel during those years. I found our discussions truly enriching, in which having the both of you participating invariably led to an energetic clash of ideas. I strongly believe our diverse or, at times, colliding views elevated our production. Pierre, I sincerely appreciated delving in this specific topic that you proposed we work on. It was both timely and fascinating, challenged me, and demanded that I got acquainted with many theoretical and practical fundamental notions. I am so grateful for this fantastic experience. You also took great care and importance in us having appropriate and sufficient resources to conduct our research. For instance, investing in an Optane testbed as soon as one could, was a crucial decision. In all, I genuinely enjoyed working with you two, I learned a lot from it; and you deserve even more superlatives to describe the sensational advising you did.

I am beyond grateful to all who financially supported us, and basically enabled this research to happen. The people from the Future and Rupture program of Fondation Mines-Télécom who selected our project and generously granted me a Ph.D. scholarship. As well as the ANR Scalevisor project, the CloudButton EU Horizon 2020 program (No 825184), the IP Paris summer school grants, and NVMW student travel grants; who all partially supported my travel expenses to various international summer/winter schools,

conferences, or project meetings. I am very glad you helped us enabling the fruitful research discussions that happened there.

A number of persons graciously gave of their time and cared to provide us with precious feedback on the research presented in this thesis. To our anonymous OSDI'21 reviewers, your reviews were honestly out of this world. Both the level of technical detail and length of your reviews, which felt crushing at first, revealed a great underlying interest for this piece of work; and that felt truly exhilarating. Those such well articulated reviews were key for us to reshape the paper and bring more value forth. Similarly, I also thank our anonymous SOSP'21 reviewers and our shepherd, Sasha Fedorova, who made this submission stronger. I send applause to our SOSP'21 artifact reviewers who went through the troubles, in the middle of the summer while I was gone on a hiking trip, of re-running our experiments to assert the functionality and reproducibility of our results. Finally, I would like to thank as well Jonathan Halliday from Red Hat for his insightful comments and accepting to meet and share with us in London in 2019 at a CloudButton meeting.

For my time as a graduate student at Télécom SudParis, I felt sincerely honored and fortunate to be able to interact on a daily basis with the extraordinary people of the Computer Science department. Most of them I had formerly met as formidable undergrad teachers, but were in fact also superb colleagues of astonishing human value. The atmosphere, dynamics and kindheartedness of the people with whom I shared this workplace during those years made my experience there all the more enjoyable. I felt it truly nurturing for my research and broadening for my general knowledge; from the terrific technical discussions we had, to the lighthearted and jovial occasions we shared. I think we all realize now, with the aftermath of several lock downs, bi-localized offices, and customary work from home; how exceptional of an aura this place had, for thriving research and our own wellness alike. Forgive me for not spelling out any names, not that I do not remember any, but rather, I simply prefer to let you include yourself if you ever strolled with us all to the Ritz at 11:30, or if you ever set foot on third storey of Bat. B or D for a coffee, hot-water, (organic) dark chocolate, or Picard ice-cream break (carrot-breaks for the finest of you guys only).

Among those fantastic individuals were also fellow Ph.D. students that I surely could not omit. I genuinely appreciated the sense of companionship we all shared. Especially you guys and galls from office B312 in Évry - who lured me away from room B311 on my first week - and later settled in office 4A260 in Palaiseau. Monika, Nabilla, you were on the leave when I started but gave me a nice "starter pack" with valuable methodology tips to stay organized throughout. Alexis L., Tuanir, I enjoyed all the technical discussions we had, particularly, all of your Adum-seasoned senior tips and administrative shenanigans. Several other students started their Ph.D. at the same time as I did. Rémi, Alexis C., Damien, I am glad we journeyed together. Yohan, thank you for having my back and helping me out a great deal by conducting some experiments for an approaching deadline. Boubacar, we did not interact lots, but it was nice having you there as well. Last, I willingly choose not to name the remaining graduate students who joined during my last year of Ph.D., for their overly disrespectful attitude. Just kidding, you were fantastic dudes as well, who found surprisingly quickly how to get on my nerves too - I loved hating y'all during this short year overlap.

A Ph.D. may also be an emotional roller-coaster, and I wish to salute all my close "teamwork" friends who helped me stay on track. Our routinary reunions (a.k.a boozing Tuesdays) saved me lots in therapy sessions - even after deducing the drinks. In all seriousness, I am genuinely moved and forever indebted by your unconditional display of devotion.

Finally, I must also express my profound gratitude to my parents, for their unfailing and unwavering affection. You always gave me *carte blanche* over my decisions and supported my choices, how queer they might have looked. You two did not just plant a seed, but also watered it with education and experiences for almost two full decades; you did great. I also thank my two brothers for their continuous encouragement.

I would like as well to acknowledge all readers of this report. I am grateful for your time, attention, and valuable comments on this work.

Anatole Lefort

Preface

This thesis comes out as the fruition of the research I conducted from 2018 to 2022, under the supervision of Pierre Sutra and Gaël Thomas, within the *Parallel and Distributed Systems* (PDS) team at *Télécom SudParis*, and in the pursue of a Ph.D. in Computer Science from the doctoral school of the *Institut Polytechnique de Paris* (IP Paris).

Research presented in this thesis. The starting point of this research was the hope of harnessing looming persistent memory technology to provide faster durability in big data applications, through novel *Persistent Data Types* (PDTs) – specialized data structures and types that would manage data relative to their persistence property in non-volatile memory.

For a topic that lies at the crossroads of persistent data management, object-oriented programming, non-volatile memory and modern persistent memory programming; the body of work that I went across when surveying existing research was rather dense. Chapter 2 sums up the large majority of what I have learned over those years.

Our core contribution – J-NVM, presented throughout chapters 3 to 5, was published in an international conference:

- ***J-NVM: Off-heap Persistent Objects in Java*** [210]. Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, Gaël Thomas. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, Virtual Event, October 2021[1].

Later opportunities to present our work also appeared on many occasions. The most notable were two international workshops: *(i)* at NVMW'22 in San Diego (CA), USA[2], *(ii)* and for HMEM'22, held in virtual format.

Last, we were extremely lucky to receive two prizes and awards for this work. *(i)* The first prize for the *Best Student Publication in ICTs for the "Plateau de Saclay" in 2022*, issued by Labex Digicosme, l'Université Paris-Saclay, et l'Institut Polytechnique de Paris. *(ii)* I am also the winner of the 2022 *Engineers of the Future Award - Engineers for Research category*[3], issued by l'Usine Nouvelle.

Other research. For the first year of my Ph.D., I was also involved in a research project with Alexey Gotsman from IMDEA Software, Madrid, Spain; where I had interned before starting my Ph.D. The project itself was about a novel distributed protocol for fault-tolerant genuine atomic multicast, that weaved paxos-like logic with traditional multicast algorithm into a single cohesive protocol. Although not presented in this thesis because clearly out of topic, this work was also published in an international conference:

[1] Our SOSP'21 talk is available at: https://youtu.be/6RcV9PSsub8
[2] Our NVMW'22 talk is available at: https://youtu.be/SChlHo7ShiI
[3] The TIF 3min video pitch: https://content.jwplatform.com/previews/VtPS3YnL

- ***White-box Atomic Multicast*** [151]. Alexey Gotsman, Anatole Lefort, Gregory Chockler. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Portland (OR), USA, June 2019.

Contents

Chapter 1

Introduction

1.1 Research Context

General context. Big data stores form the backbone of modern computing infrastructures. They support large data sets and enable processing frameworks to mine information from this data. They are designed for quick response and parallel computing at unprecedented scale. Recent examples of such systems include in-memory databases, NoSQL databases and key-value stores. The fact that storage device speeds and latencies never caught up to that of main memory, now lagging behind by several orders of magnitude, have long constituted a major setback to the architecture of these systems.

In such systems, although processing occurs in main memory, the authoritative version of data is maintained on durable storage devices (SSD, disk) with significantly slower access times. Keeping those two versions of the data mutually consistent through synchronous operations to the persistent devices typically induce serious performance and I/O bottlenecks. Asynchronous persistence has grown very popular for this very reason, in spite of the degraded durability guarantees it provides and the complexity of the algorithms it requires. In addition, because of this dichotomy between memory and durable storage, data stores are bound to spend tons of time on reboot before being able to resume normal operation. In cause, the need to reconstruct previous states in main memory, or to repopulate a data cache. This boot-up phase then is so long because it involves copying from disk to DRAM a partition of the data, in order to avoid sluggish query processing right after recovering from a system failure.

The advent of NVMM. In 2015, Intel made a shaking announcement for their *Optane Persistent Memory* product lineup. The first commercially-available *Non-volatile Main Memory* (NVMM) technology to offer byte-grain direct access to data on the memory bus – identical to DRAM – but with up to 8x larger capacities and most importantly, non-volatility of the resident data – much like storage devices. Such a technology, that combines DRAM-like accesses and performance, with storage's durability, has potential to completely redefine the role and architecture of storage systems; by presenting novel and singular opportunities for fine-grained data persistence in applications.

Non-volatile Main Memory (NVMM), by their recent advent, also came to challenge the relevance of the traditional file abstraction and its data persistence interface. The very ones that system software had been exposing for decades and that nearly all applications were built from. In cause, the very simple fact, that present NVMM technology have raw access latencies only 2 to 3x higher than DRAM, meaning a range of operation

comprised in the 100ns to 300ns. A range that file systems are simply unable to deliver, suffering from decades-long buildups of cluttering optimizations, with in mind rotational, sector-based mechanical drives, that operate on the millisecond scale, far away from the sub-microsecond scale. One may still notice improved performance over SSDs when using NVMM as regular block-based storage media with file operations, however, NVMM true and full potential lies with direct-access mode.

In direct-access mode, applications can use memory instructions to access and manage NVMM-resident data at unrivaled speeds – completely by-passing the OS I/O subsystem. That is because NVMM-resident data are directly processor addressable, meaning Optane products are effectively the very first ones able to erase the memory/storage dichotomy in commodity appliances. Imaginably, NVMM introduction in storage systems could suppress the dual representation of data and lead to extremely reduced recovery times, along with a signification reduction and simplification of their code bases, and finally, better overall throughput or response times.

Conversely, direct-access also burdens applications with new responsibilities regarding data persistence. The consistency and integrity of persisted data, relative to potential system or software faults, now have to be fully assumed by user-level code. Were a crash to occur, system software stacks would no longer be of any help to ensure a consistent version of data is recovered. This is not easily achieved: as one might recall, current machine architectures feature volatile CPU caches and relaxed memory models. Which entails that, *(i)* nothing can prevent stores to memory locations from being evicted from caches and reaching NVMM prematurely (*implicit flushes*), or that *(ii)* hardware dictates the order in which memory stores are evicted from caches and reach NVMM. Overall, a situation in which any memory store could reach NVMM at almost any point of time and irrespective of program order.

In response, Intel extended their ISA for persistent memory and included revised instruction semantics or new instructions for persistence. Their newly formed (architecture-level) programming model for persistent memory presents instructions (*flush*) to request asynchronous write-back of a specific memory location and another (*fence*) to establish happens-before relationships between different flushes.

The point is that existing in-memory algorithms, such as data structures, are absolutely unable to operate over NVMM and recover a consistent state in the wake of a crash, without first being manually amended with flushes and fences. Even though some expert programmers might feasibly fulfill the task, it remains an ordeal for their patience and skills. Furthermore, this low-level programming model abidingly leads to brittle persistence in programs. Any misplaced flush or fence instruction can cause novel bugs, that silently put the whole data in jeopardy. Simply consider that persistence exacerbate memory bugs, since memory leaks or heap corruption are now permanent.

About this thesis. In this thesis, we surf on the Optane craze, and dully note that the bulk of the effort to alleviate complexity of NVMM programming concentrates towards native languages (C, C++, etc), and almost completely forgoes managed languages, as Java. A peculiar observation, considered that Java and its ecosystem stand as major players in the big data world – many modern data stores, data analytics or processing frameworks are indeed written in Java. All of them could immensely benefit from NVMM, yet, no efficient and readily-applicable way of addressing NVMM in Java exists presently.

Our perception, is that bringing the full potential of NVMM to Java could unlock a

new kind of fine-grained and game changing persistence to a plethora of applications. At the same time, high-level languages, and in particular object-oriented idioms, look like robust candidates to intuitively intertwine the notion of persistent or recoverable data in programs, effectively abstracting away the programming burden of NVMM.

1.2 Motivation and research problems

1.2.1 Language-level support for NVMM persistence

Until recently volatile media were order of magnitude faster than persistent ones. This fundamental difference much impacted the way systems are architectured.

Recent advances in persistent memory technology promise to re-shuffle the cards. In particular, non-volatile main memory (NVMM) is a byte-addressable memory that preserves its content after a power outage. It provides durability with memory performance similar to DRAM, offering the promise of a dramatic increase in storage performance.

To harvest the benefits of NVMM, it is key to integrate it with programming languages. This matters notably for languages used in the design of the distributed storage systems at heart of nowadays computer infrastructures. Such an integration is however challenging because managed object-oriented languages are complex software runtimes which inherit from decades of refinements and optimizations. This thesis tackles the problem of integrating NVMM with the Java language.

1.2.2 No viable solution for Java

To date, approaches that integrate NVMM with Java use it as a mass storage medium accessible through a file system interface [2, 184, 332], address it through the Java native interface (JNI) [16, 249], or transparently make part of the Java heap persistent [288, 329]. As detailed next, such approaches are generic and unsatisfactory for several reasons.

- The file system and JNI approaches maintain dual representations of data, one in-memory and another on NVMM. This requires to continuously marshal objects back and forth between the persistent and the volatile memory. In particular, complex software mechanisms are necessary to keep the two representations mutually consistent. We demonstrate that (§5.3), these software costs (marshaling + consistency), although insignificant with prior storage media, came to be serious bottlenecks with the advent of NVMM devices.

- The integrated design solves the dual representation problem: plain Java objects are directly and durably stored in NVMM. Through NVMM byte-addressability, persisted objects are directly accessible with memory read and write instructions. Durable data thus no longer need to be copied over by the application from the storage media in to memory to be manipulated. However, total integration requires significant and impractical modifications to the Java virtual machine (JVM), and comes with several performance limitations and reliability concerns whose are detailed next.

Garbage collection (GC). First, integrating persistent objects on the Java heap means they have to be garbage collected. We show (§3.2) that garbage collecting just 80 GB can degrade by 3 the completion time, yet NVMM is expected to host hundreds of GBs to TBs of data. Further, we report after studying several NVMM-ready data stores

that persistent objects however are often deleted in a very limited number of places. Altogether, the use of garbage collection for persistent objects seems unneeded.

Orthogonal Persistence. Second, the integrated design lacks static persistent types and relies on Java bytecodes instrumentation to transparently check whether an object is allocated on volatile or persistent memory at runtime. In detail, Shull et al. [288] register a 51% slow down when not even actually using NVMM. (9% with a subsequent compiler optimization [289]) Furthermore, when persistent states in the application are not made obvious by types, neither the developer nor the compiler can easily identify bugs since they occur at runtime [92, 220]. Mistaking a volatile object for a persistent one leads to data loss, the opposite to a non-volatile memory leak. Instead of silently loosing data or memory, the runtime should provide help to prevent these situations.

1.2.3 Research problem statement

Thus, at present, there is a real need for a proper Java-native solution to accessing NVMM. Until then, no heavy data processing workload in managed environment could feasibly harness the full potential of NVMM and benefit from it.

In particular, an appropriate solution would be one that: (i) *does not induce any software overhead when accessing persistent memory locations.* (ii) *knows no limitation tied to NVMM large capacities and heap management* (iii) *has minimal performance impact in crash-free executions.* (iv) *is safe and empowering for programmers, that leaves them in-control, while it abstracts the complexity of failure-atomicity with object-oriented idioms.*

In contrast, the integrated design offers direct NVMM access but it trades in code simplicity for performance penalty (GC + code instrumentation) and potential reliability issues.

1.3 Contributions

1.3.1 Key insights

In this thesis, we proposes to remedy these shortcomings by keeping NVMM outside the Java heap to avoid costly garbage collection while retaining direct NVMM access as in the integrated design. To this end, we introduce a *decoupling principle* between the data structure of a persistent object and its representation in the JVM. Specifically, persistent objects are separated into a data structure that is stored off-heap on NVMM and a proxy Java object that remains on-heap in volatile memory. The data structure holds the fields of the persistent object, while the volatile proxy acts as a gateway to the durable off-heap data structure and implements the methods of the persistent object. With this design, durable data remains outside the Java heap (using a dedicated memory layout), and thus cannot be collected by the Java runtime. The dual representation of data is also avoided thanks to a JVM interface that inlines the low-level instructions that access NVMM directly from Java methods.

1.3.2 J-NVM, J-PDT, P-PFA and JNVM-Transformer

These key ideas are implemented in the J-NVM framework [210], a lightweight pure-Java library that runs on the Hotspot 8 JVM with the minimal addition of three NVMM-specific instructions (`pwb`, `pfence` and `psync` [174]).

J-NVM is a low-level interface that focuses on efficient proxy and memory manipulation. Namely, the bare logic to instantiate and destroy persistent objects and efficiently access their fields. In order to ensure recoverability and crash-consistency of durable data through simple programming abstractions, we build up from J-NVM two higher level interfaces: J-PFA and J-PDT.

- J-PFA provides failure-atomic blocks of code, i.e., a generic way of making any code crash consistent.
- J-PDT is a collection of hand-crafted crash-consistent data structures for NVMM (e.g., arrays, maps, trees), which do not rely on J-PFA for performance.

Moreover, because J-NVM relies on explicit persistent types, we provide an automated way of making Java objects persistent with the addition of a single class annotation. Indeed, we include a java code transformer to automatically enhance and decouple legacy Java classes into a persistent data structure and a volatile proxy object. It is implemented as an off-line Java bytecode to bytecode post-compilation transformation plugin integrated in the application build system. It scans for annotated classes and applies the transformation for each one, while accounting and preserving user-defined functionality in any class hierarchy.

1.3.3 Experimental results and findings.

We evaluate J-NVM by implementing several persistent backends for Infinispan [229] - and industrial-grade data store - and test them on a TPC-B like workload [9] as well as the YCSB benchmark [99]. These implementations are available at [209]. We compare the performance (§5.2) on the YCSB workloads for backends based on J-PFA and J-PDT, the original file-system approach FS sitting atop DAX-ext4, as well as a backend based on PCJ that uses internally the Intel PMDK [19] through the Java Native Interface. J-NVM is significantly more efficient than prior approaches, at least one order of magnitude faster.
Throughout our evaluation campaign, we show that:

- Both the J-PDT and J-PFA systematically outperform the external design. In YCSB, J-PDT is at least 10.5x faster than FS or PCJ, except in a single case where it is only 3.6x faster.
- While the failure-atomic blocks of J-PFA offer an all-around solution, J-PDT, with its hand-crafted persistent data types, executes up to 65% faster. Compared to the Volatile implementation, J-PDT is only 45-50% slower.
- Integrating NVMM in the language runtime hurts performance due to the cost of garbage-collecting the persistent objects. For a Redis-like application written with go-pmem [143], increasing the persistent dataset from 0.3 GB to 151 GB multiplies the completion time of YCSB-F by 3.4

Other relevant insights from the performance analysis of J-NVM are as follows:

Marshalling. The low performance of FS comes from (un)marshalling operations to move the persistent objects back and forth between their file and Java representation. PCJ is highly impacted by the cost of JNI calls to escape the Java world. These operations were commonly used and had no significant impact with slower storage media, but can now be bottlenecks with NVMM and should be avoided where possible.

Caching. We observe in the YCSB benchmark that J-PDT does not benefit from caching. Indeed, because data is accessed directly and only proxies are kept in the cache, increasing the cache ratio has almost no impact on read or update latencies.

Recovery. The performance of the recovery procedure is evaluated with a TPC-B like (transactional) workload. J-PFA recovers about 4.7x faster than FS and up to 8.6x faster with a recovery optimization possible for purely transactional workloads. Conversely, FS has to repopulate the 10% in-memory cache eagerly on recovery when J-PFA can only recreate proxies lazily with much less NVMM bandwidth usage.

1.4 Thesis Outline

The remainder of this document organizes as follows:

Chapter 2 provides background material on persistence abstractions, programming challenges with NVMM, and the present state of NVMM support and techniques for failure-atomicity (first broadly, then in the specific case of Java). In particular, since NVMM media have only been available for a few years, we ought to examine them in lengths, and present the ample body of knowledge surrounding this novel technology.

Chapter 3 defines the programming model of our contribution, and in essence, how it pragmatically erases limitations of anterior proposals, to then provide base blocks for high-level NVMM programming with negligible overhead compared to native languages.

Chapter 4 describes the core of our contributions. In order, the system design and internals of the J-NVM library, our methodology to build efficient and recoverable data structures (J-PDT), our implementation of failure-atomic generic sections of code (J-PFA), and last, our bytecode enhancement tool that generates persistent objects from regular POJOs (JNVM-Transformer).

Chapter 5 presents a performance evaluation of our contributions: J-NVM, J-PDT and J-PFA. We compare them to the publicly available solutions for NVMM persistence in Java (file system and PMDK through JNI).

Chapter 6 concludes this thesis with a summary of our work and suggestions for future research directions.

Chapter 2

Persistence, Non-Volatile Memory and Java

Non-volatile main memory (NVMM) presents new opportunities for data persistence in applications. At core, it offers fine grain access to durable data at the level of memory instructions. However with NVMM, programs are also directly liable for data consistency in face of crashes. Integrating the notion of persistence into programming languages, such as Java, can make NVMM easier to use and applicable to a broader range.

NVMM is such a recent addition to commodity servers that we assuredly cannot delve into language-level integration challenges first hand. First, we have to review its subtle inner properties and tedious programming model. Furthermore, entangling persistence with programming languages demands interface design to be on-point with the user needs. For that, we must also delaminate past or present-day, failed or successful persistence facilities, so as to shape pragmatic and useful NVMM support software.

In this chapter, we propose to examine all of the above to then formulate requirements for satisfactory NVMM persistence facilities in Java. To this end, we organize the chapter as follows:

(§2.1) We present the notion of data persistence.

(§2.2) We go over historical solutions, and focus on traits that made them last long-term.

(§2.3) We introduce early schemes for software-based persistent memory. They were made for spinning drives, but laid persistence abstractions we might be tempted to relight.

(§2.4) We examine NVMM, how it compares to software persistent memory, and hardware properties that settles it further away from DRAM than we all anticipated.

(§2.5) We detail several active or prospective application domains of NVMM, while trying to get insight on user needs and expectations.

(§2.6) We stop on the low-level aspects of NVMM programming, to understand intrinsic challenges, correctness properties in faulty executions and why solutions have to differ from the ones established using traditional durable media.

(§2.7) We inspect existing and proposed programming abstractions for NVMM that simplify reasoning about recovery. We also stay alert about support they expect or assume from language execution environments.

(§2.8) We finally review industrial and academic work addressing NVMM persistence in Java. Doing so, we shed light on the fact that none of them properly address every essential attribute we establish from the preceding sections, for Java-level support of data persistence on NVMM.

(Table 2.4) We provide a comparison of NVMM support libraries and their features (failure-atomicity, model applicability, and NVMM efficiency).

2.1 Notion of data persistence

From the early days' punch cards to modern flash memories, forms of permanent storage have commonly been used by computer systems to operate on digital data. Regardless of the decade and technology, input and output data along with computer programs have all been stored on media where they could survive power cycles of the processing unit. This stemmed from the need in data processing for long term data storage, data portability and system interoperability.

Storage is certainly a central components of computer systems and present-day big data stores form the back bone of modern computing infrastructures. Yet, applications remained accountable for the proper handling of their transient state residing in-memory, in relationship to their authoritative state that stayed persistent on durable media. Interacting with durable storage have consequently been a major research concern with the recurring aim of providing painless ways of reading and writing bytes on non-volatile media in both an efficient and safe way. Fortunately enough, we now have abstractions to reliably manage, logically organize or share stored data. Yet, we have not completely strayed away from some archaic and unsafe ways. For instance, consider that programs may still have to ingest and process stored data from a stream of homogeneous and untyped bytes. Altering data on durable storage must also be done by applications in account with the potential system failures and program crashes that may occur while the durable media is being written to.

The notion of data persistence then emerged as a more reasonable way of envisioning the interplay with storage from a application design perspective.

Def. Persistence, for Atkinson et al. [46], is thought of as a property of the data, in that it characterizes their ability to outlive instances of a program. In turn, a piece of data remains persistent for the extent of time during which it may be recalled and used by a program.

In practice, this often results in the ability to denote or allocate data as persistent from within the programming language, where the data is also given structure. The clear benefit of which is supporting the system by abstracting away the storage from the programmer and instead providing them with an array of tools and APIs to manage durable data. With persistent data denoted, the programmer no longer has to do the heavy lifting, which is then deferred to a library, a language runtime and compiler or even the operating system. Therefore, the programmer needs only to be trusted with the marking of consistent states of the persistent data throughout the execution, to preserve the application logic, in respect to potential failures. The underlying persistence abstractions can in turn work out the recovery and eviction of the durable data accordingly.

Research has led to the inception of various abstractions following these principles, with resulting schemes for data persistence occurring at different levels of the system software stack. (§2.2) The next following sections detail them along with historical background and solutions.

2.2 Early Works & Historical Solutions

For decades now, computer systems have been relying on memory technology that was volatile, meaning their content did not carry over subsequent power cycles. As early as the 1970s, volatile semiconductor memory superseded the dominant technology and

could since then be found in computer memory as DRAM, of large capacity used for main memory, or as faster but more expensive SRAM, found in CPU caches. Their massive adoption, because of improved performance over former technologies, also deeply grounded the fact that data in computer memory were only transient. Typical durable data storage devices such as tape, hard disk, or flash, were also order of magnitude slower than memory. Therefore, efficient data processing demanded memory and storage to be tiered. The transient nature of memory effectively made it, in that hierarchy, a scratch space on which processing occurs using a temporary working copy of the persistent data found on storage devices.

Moreover, storage devices have not only just been slower than memory but feature much different access patterns as a whole, since they are sequential and block-based for the most part. As such, working with stored data is then typically done with large grain operations and using hardware-fitting algorithm in order to maximize device bandwidth and endurance. In contrast, in-memory data can be manipulated with very fine grain, in a highly concurrent manner, and kept organized in data structures carefully tuned to optimize program decisive operations.

Persistence in systems has thus historically been shaped around these two overarching facts:

- *In-memory data does not need to remain consistent at any point of time.* Because main memory is swept across system reboots, it can only act as a temporary cache and in-memory data structures will not be recovered. Consequently, systems have abused this property to perform optimizations. For instance by allowing reordering of memory instructions from the program order on non-conflicting data.

- *Durable data consistency can be managed with very complex algorithms.* Indeed, storage being orders of magnitude slower than memory, protocols for crash-consistency can maintain and update as much volatile metadata as they want before affecting storage response times. Transactional systems have leveraged this fact to provide a full spectrum of very useful guarantees, supported by extremely complex in-memory data structures. In addition, systems have also been used to perform tons of preprocessing on the data before issuing commands to input/output storage devices. As such, data is commonly transformed and laid out in a more compact format to reduce the impact on device bandwidth, and carefully arranged on the device to minimize latency of subsequent read or writes or prolong the device endurance. Data also often pass through multiple distinct buffers in memory to be coalesced into blocks that can be written to the device.

With all that considered, durable data management in systems have grown into a confusing stack of cluttering optimizations aiming at utilizing better the hardware for performance, but not directly serving towards data persistence. In the light of these architectural concerns, clearly, past research on data persistence might not be of much technical use to present-day persistent memory technologies, considering that they do not fit in the current memory hierarchy. Nonetheless, understanding older research on persistence will be useful to pinpoint where persistent memory may stand to benefit the most modern data processing systems. Hence, we revisit literature from the past with the underlying aim of designing successful abstractions for current persistent memory. Precisely, which features were crucial for some abstractions to pass the test of time, and among the ones that did not make it to this day, which ideas could be rekindled or definitively forsaken.

(§2.2.1) We begin with the concept of single-level store - a pioneering tentative of uniformly integrating main memory and persistent storage in a single coherent virtual address space. After that, we continue with the file systems (§2.2.2) and databases (§2.2.3), that have been the two ubiquitous interface for persistence in the span of the last decades. (§2.2.4) We close this section with persistent object systems, that bring persistence in object-oriented programming through database-like operations and transactions. We note that the outline for this historical tour on data persistence is taken from Seltzer et al. [283], who analyze how NVMM differs from traditional approaches to persist data.

2.2.1 Single-level store

The first instances of persistence as an abstraction goes back at least to the concept of single-level store (SLS). The single-level store builds on virtual memory to erase the distinction between main and secondary memory (e.g., on-disk storage); allowing processes to envision and handle data, whether volatile or persistent, in the same way. In practical terms, applications consistently access all of their data from main memory, by following pointers. For that, any combination of memory and storage is automatically tiered and uniformly presented in the processes' address space. When accessed data is in effect missing from main memory, the operating system transparently fetches it from disk; and conversely, data written to memory is transparently written back to disk eventually. From a high-level, it enables developers to write applications as if they never crash, and to omit code for persistence and recovery. From the system's perspective, the whole application state - which is for the most part its memory content - is transparently persisted. In that, SLS conceptually places data persistence as a service provided by the operating system.

The Atlas computer [136, 196] from the University of Manchester was the first system to implement the idea of integrating main memory with secondary storage seamlessly together. For economical reasons, Atlas employed jointly a small expensive core memory and a magnetic-drum store, but it eliminated the inconvenience of that arrangement by making transfers completely automatic. At the cost of an effective loss in machine speed, non-expert machine users were as such relieved from the time consuming task of programming transfers between the two types of stores. Unifying memory also made programs portable on machines with different memory divisions, and able to automatically take advantage of additional fast core memory. Kilburn et al. even recognized the broader applicability of the technique as they wrote: « [...] the paper describes an automatic system which in principle can be applied to any combination of two storage systems so that the combination can be regarded by the machine user as a single level » [196].

Atlas forms a virtual memory restricted to computational purposes, akin to modern paging and swap, and it consequently still requires a dedicated memory for long-term data storage. As the Multics designers pointed out, this results in the duplication of mechanisms for disk space organization, protection, and incurs a lot of copying between the virtual memory and the backing storage. This observation led them to the design of a new store that directly addresses any physical storage location and allows to control how information is shared among processes. Designed the mid 1960s, **Multics** [250] leverages a technique now commonly termed virtual memory to introduce persistence in SLS: « the Multics user no longer uses files; instead he references all information as

segments, which are directly accessible to his programs » [58].

The store can be seen as a single uniform persistent virtual memory [58, 107]. In detail, data is accessed in Multics through "segments", that are linear arrays of information referenced by symbolic names and directly processor addressable. These symbolic names are permanent and hierarchically combined to form unique paths to the data segments entries. Each entry is associated with a set of attributes to denote user capabilities over the segment and provide access control lists (ACLs). When users reference any segment, they are checked by hardware and software to provide controlled access to the data. By design, users need not to know the device or physical address of the data for it to be mapped and addressable through the virtual memory. Reading or writing to a particular address of a segment materializes the associated page and disk block into main memory synchronously. Conversely, alterations made to pages in main memory are written back to the disk asynchronously by the kernel at some later time.

In effect, Multics and its virtual memory design pioneered the integration of volatile and durable media behind a single-level cohesive address space with support for persistent data. With no prevailing standards at the time, its designers experimented and extended the ideas from Atlas with two fundamental concepts tied to persistence: naming and protection of the data. Even though the innovative idea of transparently managing persistence using the paging system consisted in a powerful abstraction for programmers, it remained insufficient in meeting the requirements of some applications. Precisely, when programmers would purposely carefully order disk writes to provide strong consistency over durable data, with the aim of preventing corruption or cleanly recovering corrupted data from a system interruption. It turns out that this is simply not achievable in this model, as noted in [152]: « Because the ordering of the actual disk writes bears no relation to the ordering of the modification of the pages, [...], it is impossible to place an ordering on the modifications that actually appear in the disk file. »

Eventually, the Multics designers provided a supervisor call to force all altered pages to disk. It was effective but expensive and breached the supposed transparency, contradicting the whole architecture of the Multics virtual memory. « That "file sync" was necessary admits of a major flaw in the design. » [152].

Flash forward to present times, and hierarchical name spaces or access control lists are found in virtually every file system on every modern operating system. Modern Unix-like and Windows operating systems also support a restricted form of Multics' virtual memory abstraction through memory-mapped files.

We will see in §2.4 that modern persistent memory still rely on this abstraction, with challenges reminiscently resemblant to those of Multics' memory-addressed storage.

The SLS idea was not abandoned with Multics, but carried over in the design of persistent object systems and later object-oriented databases that are discussed in §2.2.4. Before though, we must mention file systems and databases to call up their specificities towards data persistence, after which, we will resume this discussion.

2.2.2 File systems

File systems historically dominate, from the 80s onward, the management of persistent data. As underlined in §2.2, following the adoption of fast semiconductor memories, secondary storage media became largely block-based. With that arose the need to address the sequential nature and larger access grain of these devices.

This complexity is dissimulated with a layering of storage mechanisms in the operating system, all hidden behind a file abstraction and its rigid I/O interface. In detail, **(performance)** Caching mechanisms to coalesce individual data writes into write-back-ready device blocks or to retain in memory often-accessed data; **(sharing)** Mechanisms for proper scheduling of disk operations when multiple programs work with the same device; and **(resilience)** Mechanisms for the building of complex storage hierarchies and disk arrays engineered to aggregate spinning drives bandwidth while covering for potential hardware-failure-incurred data loss. In this thesis, we do not cover the internals of these mechanisms, but rather only say they have been extremely useful, awfully complex, and definitely system-specific problems - not applicative issues. Instead, we are rather interested with the implications of the file system and block-device interfaces on the user's perspective.

The file interface addresses the naming and protection of durable data similarly to the SLS. For the same purpose of referencing and multiplexing access to stored data, it relies on a permanent (on-disk) logical name space to reference files and per-entry permissions to provide ACLs.

Unlike in the SLS design though, data persistence with files is not transparently managed. In fact, it is quite exactly the opposite: users have to issue explicit interface calls to retrieve data listed in files, or propagate changes to the files. The inner structure of a file and the way data is laid out in it is as such fully defined by applications, which need not to be concerned with where the actual physical data chunks resides on durable media. The file system and block-device interfaces allow for a lot of flexibility in that, programs have full control over: *(i)* how data are structured within files, and *(ii)* the order in which data are accessed since device operations reflect the program order. Yet, the underlying device nature and physical location of the data are completely abstracted away from the users. This combination of flexibility and control may very well be the reason for which time can now attest the success of the file abstraction.

Further, despite how cumbersome the file interface might look in comparison to single-level stores, expert programmers are empowered with precise calls that allow them to perform application-specific persistence optimizations. Users can well-tune on-disk data layout or durable data manipulation algorithms to speed up the flow of specific application paths, without interfering with the handling of data in the rest of the program. In comparison the SLS took on the challenge of providing persistence orthogonally to the logic and execution of the application and stripped programs from these data design options. In turn, this leaves them with no choice but sticking with the in-memory representation of the data established by the programming language compilers, and adopting the one-size-fits-all write back and consistency patterns of the operating system. Remember that spinning drives are orders of magnitude slower than memory; optimizing data movement on a per-application basis was highly desirable to provide swifter systems with lower response times; not giving control over that was mostly a deal breaker for the SLS.

These are the facts we take away from the file interface, and for which it was highly favored and widely adopted over time. The file interface today knows limitations. The apparent flexibility turns into a double-edged sword when deployed over faster hardware storage devices, namely flash memories that are now capable of microsecond response times. In cause is the direct lineage of file systems, that have engendered over time an array of both system and user-level techniques carefully designed and layered on top of

each other with millisecond-scale hardware in mind, with an overall resulting software complexity simply too high for lower latency devices [55, 66].

Modern performance consideration aside, file systems have also faced more substantial semantic shortcomings. Traditional file I/O do not provide any guarantees to the data contained within files, in respect to system failures.

A file system needs only, to be deemed « correct », to ensure the resilience of its own on-disk data structures and metadata. Not that this property is unimportant, it is even critical to the integrity of the whole file system as well as of the user data. Corruption in the metadata can prevent the file system from mounting, cause loss of entries (i.e., files, directories) and ultimately, loss of data. For these reasons, proper management of metadata is paramount in file systems and various techniques, principally logging, are applied to ensure they are fault tolerant.

However, since they are restricted to the inners of the file system, following the general design philosophy, programs remained burdened with the implementation of their own data safety and recovery logic. In general, the file interface does not offer facilities to issue and execute multiple operations in an all-or-nothing manner over the data. That being said, advanced modern file systems typically offer resilient user-data writes through *copy-on-write* or *logging* [273] techniques.

All considered, the file abstraction consists more of a building block to access durable media, than an abstraction to program data persistence. Through files, data is manipulated unsafely as raw streams of untyped bytes, with no built-in safety, recovery or transactional facilities. Incidently programs directly leveraging files are fully in charge of the proper interpretation of durable bytes and must do the heavy lifting to reconstruct a consistent application state from them.

In the light of these weaknesses, the file abstraction usage should really remain limited to power users, and serve in the building of data storage services. Databases, that will be discussed next, typically provide an interface to persisted data with richer structure and semantics, and prevent inconsistent states originating from concurrent alterations or system failures.

2.2.3 Databases

Databases, and especially relational databases which have been dominant as early as the 80s, have been tremendously helpful in the building of software applications since then. They are not simply external services that multiple programs can interact with to fetch or store data, but fully data-centric systems. They were envisioned with the primary goal of relieving programs from the painstaking task of managing their durable or shared data.

Databases arrange data logically following a given model (e.g., relational). In particular, they offer a mean to model application data into database records, as user-defined combination of database builtin types. This allows databases to view and interpret fields of individual data records, unlike file systems that must treat whole file chunks as black-boxes.

In comparison with the file interface, the database's one is just as rigid but it offers a much richer semantics. Remember that, with files, the user requests/emits a sequence of raw untyped bytes from/to a given file at a given offset. That remains a fairly low-level

approach, in the sense that, the program must retain knowledge of the file map, meaning which data lives at which offset, and properly map the raw data bits to application in-memory types and structures. In contrast, databases traditionally allow for data retrieval or update through complex declarative query statements.

Data is no longer only accessed using uniquely defined symbolic names, as in a file system name space, but is enhanced by the capabilities of the query language. The use of external identifiers embedded in records may be leveraged to reference other pieces of data, mimicking in-memory pointers. Thanks to the indexing of the data fields within records, the database engine is able to support a query language that also allows for the use of filtering or combining predicates. For a given request, this permits to identify data by its characteristics and properties to selectively fetch/update part of it.

The white-box approach to the structure of data inside databases allows for internal optimizations performed by the database engine, along with semantically-rich operations performed by the user. The data space can even be dynamically reconfigured through queries to the database management service. This allows, for instance, to migrate application data from one model to another.

With a database, users need to specifically design and structure data records within the model supported by the database. In this task, they might feel restrained by the modeling power and expressiveness offered by the database. They must further bridge the representation of data records in the database to the in-memory structures defined and used in their programs. The two representations are in that mutually exclusive, much like in the case of the file system. Furthermore, maintaining the two definitions of the data records mutually compatible is prone to programming errors.

Despite the added complexity, databases have been historically a success and users are still willful to organize and arrange data according to the database model. The explanation lies with the service provided by a database in regard to the data and guaranteed properties that immensely simplify application programming.

In particular, the database interface is empowered by transactions [59]. Transactions allow a series of user requests to be bundled and processed as one single database operation. Users think and reason about a transaction as a single cohesive operation, from an outside perspective. As detailed next, this abstraction also provides a strong form of concurrency control over shared data. Overall, transactions are more reasonable to grasp and more convenient from the user's perspective than the file system interface. This explains why they are extensively employed in writing applications.

Data persistence has always been a core concern in the design of traditional database systems. Precisely, ensuring user data validity in presence of errors and power failures, for which database transactions have also been accountable for. Indeed, in 1983 the set of properties traditional database transactions offered was first coined with the acronym ACID (atomicity, consistency, isolation, durability) [155]. Where the 'D' in ACID stands for durable, requiring that the effect of a transaction becomes persisted once it is committed.

Persisting committed changes to the database is the task the recovery manager (RM) is in charge of [59]. It issues read/write operations to the cache and fetch/flush to stable storage. Multiple implementations exists based on different techniques, either based on logging or copy-on-write. Logging-based approaches, such as *undo* and *redo* allows in-place updates of the data by storing logs on the durable media. On one hand, redo

logging first adds modifications to the log, leaving the original data unchanged, then later applies them once the log is committed. In the event of a non-corrupting failure, the modifications are played back. On the other hand, techniques based on undo logs first save the original data in the log. Modifications are kept in a volatile variable until commit time, and only made persistent at the end of the transaction. They are reverted back in case of a failure.

For improved performance, logs may be written in an append-only fashion on disk, prior to updating the actual data. This pattern is called write-ahead logging (WAL). ARIES [233] is among the first popular algorithms of this family. Another advantage of WAL is that it enables to write in-place, meaning that updated data may be written safely at the same physical location, saving the hurdle of modifying the data indexes as well. WAL is also found in modern file systems where it is often used to preserve file system metadata. Other file systems use WAL for fast synchronous logging of disk writes, such as ZFS and its intent log (ZIL) [317].

Other non-logging based techniques to persist committed changes atomically include shadow paging, which derives from copy-on-write, used to purposely avoids in-place updates. Data updates are written out-of-place, in newly allocated disk pages, with no durability and consistency concerns since they are not yet referenced. Once data is ready, all references to the old page are swung to the new one, in an atomic way. Out-of-place updates may help reduce disk hotspots, when some programs have to write over and over the same information. However, typical storage media feature faster sequential access, making WAL a more popular option overall.

Databases and especially recovery managers have then a good lot of techniques and implementation to choose from in order to ensure atomicity and durability of the data. Modern systems may defer part of that decision to the end user, with the use of a recoverability or consistency criteria as a configuration parameter of the transaction manager.

A recoverability criteria defines the form of executions passed to the recovery manager by the transaction manager. In general, such executions are strict, that is there is no concurrent conflicting transactions passed to the RM. Some programs may however prefer to work with more relaxed executions constraints, and favor improved performance.

Database as a research area definitely bred a rich history of theoretical properties and practical techniques for data durability and atomicity. Techniques that later trickled-down to the lower layers of the system stack, as we hinted earlier they could now be found in modern file systems as well. They also served as a solid bedrock in the design of persistent objects systems and object-oriented databases, that we are just about to discuss.

2.2.4 Persistent objects

Let us just rewind to the early 80s, and consider the ideas work from the previous decade has fanned out. The single-level store blurred the line between main memory and storage, unifying the two in a cohesive virtual address space. File systems however were getting mainstream with their rougher yet explicit interface to persistent data, conveying more flexibility and control towards durability on a per-program basis, which in turn enabled the development of user-level services for data storage. Databases sys-

tems already have had a decade-long competition around data models, which resulted in systems providing high-level relational interfaces to data storage.

The end user was by then already as much as possible isolated from the underlying storage structure, and had access to facilities for data control, such as logging and recovery, or maintaining consistency in a shared-update environment. In this context, in-between relational databases and the remanence of integrated and transparent approaches to durability, persistent object systems and object-oriented databases arose in response to prior systems' limitations.

As we previously stated, relational databases had their own way of structuring data records. Conditioned by the relational data representation model, as originally conceived by Codd [95], and later realized in systems such as SystemR [45] or Ingres [160]. That model was further constrained by the set of builtin data types offered by the database engine, limiting their modeling power and ability to conceive complex object types. Since then, databases have had a problem dubbed "impedance mismatch" [46, 100]. This problem is the fundamental mismatch between the data representations in the database and in the program. It results from this problem a programming and modeling hurdle for application developers.

Despite this problem, time can now attest the success of the relational model [10], and how it trampled over alternative more integrated approaches. In particular, Postgres [302], presently an industry staple database system, was conceived in the mid 80s as the successor of Ingres [160]. Nonetheless, we are all still well-advised to refresh our memories with the rich history of persistence lying in object-oriented databases and persistent object systems. Especially with the advent of low-latency storage technologies that might want to rekindle similar ideas of persistence attached to in-memory objects. This is the purpose of what follows, where we detail the research which was conducted around persistent objects.

Back in the early 80s, the increasing popularity of object-oriented programming sparked the idea of eliminating the difference between the database management system (DBMS) and programming language models of data. In the line of the SLS that unified volatile and durable data behind the same virtual address space, these systems attempted to bridge volatile and persistent data representations behind the object abstraction. Ultimately, these systems strove at integration of persistence with programming language constructions and as a property of the data objects.

This time period has cultivated different sort of approaches to *persistent application systems* [48], to fill the needs of application composing long-lived data and programmatic computations. We will next set our focus on fully-fledged systems that allowed direct manipulation of persistent data through object-oriented idioms, while exhibiting database-like capabilities, essentially transactions and querying.

Atkinson et al. introduced with **PS-algol** [46] principles and an abstract machine to bring persistence as an abstraction to programming languages.

Their base hypothesis was the minimality criteria as they believed « it should be possible to add persistence to an existing programming language with minimal changes to the language. » [46] An hypothesis which stemmed from their observation that managing the mappings between long-term and short-term data was impairing the quality

of application code and distracting programmers. As such, they sought of offering « the facility for persistence for little or no extra effort. » [46]

A criteria they supported with the idea that persistence should be a property of data, orthogonal to object types or execution parameters. The same procedure should be able to work with persistent or transient objects, and that the way of identifying persistence in the language should be independent of data types.

The interface they specified to enable programmers to work on persistent data is eerily modern, in the minimalist and restricted set of procedures it provided. Programmers would call a procedure to open a database from a symbolic name, from which they would retrieve a pointer that « is the root from which preserved data is identified by transitive closure of reachability. » [46] A commit procedure was available to end the transaction and record all changes to the underlying database. A table abstraction for the means of associative lookups was also provided, with traditional key-value mapping and lookup/enter/scan procedures.

The article further describes the data movement between main store and backing store, which are hidden away as internals of the language runtime. The PS-algol abstract machine was devised to rely on a backing store, that could be a database shared by multiple programs. Persistent data is, in an example implementation, held inside the **CPOMS** [94], the persistent object management system written in C, an external persistent object store that acts as an external persistent heap for PS-algol.

Within the CPOMS, data objects reference each other using *persistent identifiers* (PID). Data is pulled from the external heap lazily, as PID are encountered by the programming language's abstract machine during execution. When that occurs, the persistent object manager is called to locate the object and place it on the local volatile heap if needed. The PID is then replaced by a *local object number* (LON), the equivalent of a traditional pointer, once the object was successfully copied on the local heap. Referential integrity is ensured by a single table used for tracking of objects copied onto the local volatile heap. LONs have to be replaced at commit time because of their transient nature, before data objects are effectively copied back to the external heap and made stable. This notable technique of replacing PIDs with LONs momentarily is known as *pointer-swizzling* [237].

As a whole, PS-algol managed to design an abstract and generic way of bringing persistence to programming languages. The minimal set of facilities to manage persistence makes it extremely easy to grasp. Persistence is treated orthogonally, enabling program procedures to be called on transient and persistent objects. The burden of managing persistence is then placed on the language runtime, leaving programmers with a single procedure to recover the graph of persistent objects living on the external persistent heap.

While PS-algol provides a language design method to orthogonal persistence, it does not tackle the data representation model issue. Data objects are simply put away in a backing store, with extremely simplified database-like semantics, and no support for useful relational queries.

In contrast, other systems from the same period that presents themselves as database systems underwent the way of incorporating secondary storage management inside programming language runtime.

OPAL [100] is a language derived from SmallTalk enhanced with an implementation of the GemStone data model. They conceived the GemStone model fully beforehand, as

one that would be suitable for data manipulation and general computation. It is derived from a pure set theoretic data model, formally specified, with support for declarative queries and in-object data history. The paper describes in detail this specific data model and how it was merged into the typing model of SmallTalk. They further explain how a simple path syntax could be used to navigate sets declaratively while being translated into database queries by the runtime. The system was deployed on special-purpose hardware for efficiency, and worked as a remote service. It received blocks of code from client software, in the form of compiled bytecodes, which it interpreted in separate user sessions.

Along the same lines, the **ObjectStore** [205] database system devised extensions to include data persistence in C++. They enhanced the C++ language into one that could provide a unified programmatic interface to both persistently and transiently allocated data.

Their take on persistence remained orthogonal, in that C++ object types did not reflect their allocation heap whatsoever. Objects could incidently be located and possibly later updated either by reference traversal or associative queries, regardless of their allocation nature. As we already pounded with other related work from the same period, no translation code was needed and the modeling power of persistent structures were limited only by the extent of that of the host language.

ObjectStore was still referred to as a database system, thanks to the facilities they provided through C++ user libraries to programmers. These included a library of collection types, bidirectional relationships, along with facilities for querying, versioning and database administration. Their extended C++ interface and syntax relied on a modified C++ compiler. **(Recovery)** was similar to PS-algol: the user would recover a database entry-point (*root*) pointer through a dedicated procedure call and interact with the database objects within transactional blocks of code. **(Allocation)** was performed with a decorator to the *new* operator, to request the object to be placed in the table identified by the *root* pointer. **(Collections)** resembled ordinary data containers, except for the *policy* criterion users could specify on a per collection-basis to dictate how external changes made to the database should be reflected in the local collection view of the data. **(Bidirectional relationship)** were available as a syntax extension to class definitions supported by the pC++ compiler of ObjectStore. They enabled to express one-to-many or many-to-many relationship between objects of distinct classes and automatically maintain and update reverse pointers. Reverse pointers are especially useful, when for instance, retrieving an object from a query to navigate back to its parent set just by following a reference. They are however a pain to manually maintain as programming errors result in loss of referential integrity and serious bugs, but pC++ made use of their compiler to statically enforce these constraints. **(Queries.)** were also supported by ObjectStore's pC++ compiler. It was necessary to statically verify their specific associative query syntax with usual selection predicates, which were essentially tailored C++ expressions applied to one or more of their collections. **(Data model)** wise, ObjectStore was then something akin to an object-relational model, as they empowered C++ object and classes with collections and relational DBMS capabilities.

The architecture and implementation of the ObjectStore system was realized on conventional hardware and operating system such as Unix, and worked in client-server mode. **The client machine** executed the C++ program within the same virtual address space as of the ObjectStore server holding the persistent data. The Unix operating system on the client machine reflected back the memory faults raised by missing memory pages to the ObjectStore service, which fetched them remotely. **On the server machine** , the

ObjectStore application had no knowledge of the contents of pages, simply passed them to and from the client, and stored them on disk. It relied on a dedicated file system to avoid unnecessary operating system intermediation, and was also responsible for concurrency control and recovery. It used to this end a log and WAL protocol, also found in traditional DBMS.

ObjectStore reminded us of the SLS systems that builds on virtual memory, where here the client holds a cache or subset of the total memory and the server stows colder pages on persistent media transparently. Where it parts from single-level stores is with the added facilities for database-style queries and relationships enhancing the C++ compiler and language with relational DBMS semantics.

Ultimately, in this object-store jungle, different groups combined their modeling efforts and formed the Object Data Management Group (ODMG) [81]. Their specification included descriptions of the core object model, along with languages for object specification or query, and specification of the bindings with C++, SmallTalk, or Java.

Speaking of Java, right about its first release in 1995, the *Persistent and Distribution Group* at the university of Glasgow, also behind PS-algol and other persistent object systems, jumped in with their hopes of demonstrating their hypothesis for *orthogonal persistence* on what was envisioned to become a language and platform for future enterprise applications. As such, Java benefited from works around persistence from its very beginning, on the trend of orthogonal persistent application systems.

Extending their previous definition for persistence: « This is the period of time for which the data exists and is useable. » [46] from PS-algol, the *Persistence and Distribution Group* at the university of Glasgow (Atkinson et al.) founded the **PJama** project [49] and proposed **PJava** (persistent Java) [47].

Building on their previous eighteen years or so of research on persistence [48], they integrated in PJava their three base principles: *orthogonal persistence*, *transitive persistence*, and *persistence independence*. **(Orthogonal Persistence)** First, PJava conceptually classified lifetimes of data on a scale from transient to infinite, and said of persistence to be orthogonal when data of any kind had the same rights towards lifetimes accessible to them. **(Transitive Persistence)** Second, remember that the systems we previously mentioned relied on explicit decorators to request persistent allocation and database-like facilities to locate and recover durable data. PJava however introduced the notion of persistence by reachability or transitive persistence, meaning from a set of "persistent root" designated objects, any transitively reachable object also inherited the property of persistence. **(Persistence independence)** Third, persistence independence implies that code and software may be operating on transient or long-lived data indistinguishably, enabling straightforward re-use of libraries and classes regardless of the execution context. By doing, they were adhering to the *Write once, run anywhere* (WORA) slogan touted by Sun Microsystems to illustrate the benefits of Java—also later mocked as *Write once, debug everywhere* due to the disparity of Java runtime implementations.

Next, onto the implementation side, PJava required no changes to the Java language, but used an additional API through the *PJavaStore* class to register new persistent roots, or orchestrate transactions dynamically. We have already alluded this generic store interface as a way of retrieving an entry point with PS-algol or ObjectStore's pC++. Similarly, *Transitive persistence* is the real addition of PJava, as we already discussed *orthogonal persistence* and *persistence independence* with previous work.

Persistence by reachability in PJava is implemented by piggy-backing on the garbage

collector running over the volatile heap. « The candidates for promotion are found by treating all the mutated pointers in objects in the cache as roots of the new reachability sub-graph » [47]. As in previously discussed designs, the objects are promoted to an external persistent heap managed by an object store.

The external object store also needed to run concurrent garbage collection over disk space, because of the requirements of Java towards automatic memory management. This could be achieved unobtrusively to the language runtime, as demonstrated by PMOS (Persistent Mature Object Space) [238]. PMOS required no changes to the language interpreter to perform garbage collection on a persistent store, with recoverability and stability constraints on the data. It was further designed to support a number of programming languages such as PJava or Napier88 [236], another orthogonally persistent programming language.

Although seducing, the idea of orthogonal persistence and its three principles behind the PJama project and PJava did not catch into mainstream Java. The quickly evolving Java ecosystem and specification were colliding on many aspects of the PJava principles, as they thoroughly reported [183]. As a whole, integrating these ideas in an efficient Java language runtime was perceived as far too ambitious from a research and engineering perspective.

Instead, Java standardized object-relational mapping (ORM) with the Java Data Object (JDO) [182] and Java Persistence API (JPA) [120]. It was largely preferred and eventually supplanted the idea of orthogonal persistence in Java. As the ODMG did before, JDO and JPA specified a common interface to persistence in Java, including operational semantics, transactional support and object to relational entity mapping. That common API was not limited to relational databases, but could enable many heterogeneous data sources to supply data as regular Java objects. Hibernate [11], developed by Red Hat and first released in 2001, still goes strong as one of most well-known example implementation of JPA, and can be used to provide transparent persistence to plain old java objects (POJOs).

Finally, to conclude on persistent object systems and object oriented databases, we seen through their evolution during the 80s and early 90s that they ventured on the unification of data models for both computations and long-lived storage. They took on the challenge of hiding the data property of persistence behind idioms of object-oriented programming. The trend was to support persistence with the least amount of visible changes on the programmer's end. In that, they devised persistence as an orthogonal property relative to data types and program executions. Denoting persistence would incidently be done for each data objects at allocation time, or be automatically computed following the transitive closure of special persistent "root" objects.

More interestingly, was their genuine emphasis on database-like facilities for locating and updating data, with functionality such as associative queries, transactions, versioning or replication. In the context of a direct competition with relational DBMS systems, they looked for ways of extending language syntax to support operations coming from older database systems.

Although the alluring idea of orthogonal persistence did not subsist, due to various limitations, data models were nonetheless consolidated under the object-relational model. The introduction of standardized ORM specifications and real implementations enabled conventional languages and DBMS to communicate, effectively enhancing object-oriented programs with database functionality.

2.2.5 Summary and limitations

Through this historical detour, as we came across the single-level store (§2.2.1), file systems (§2.2.2), database systems (§2.2.3), and finally object-oriented storage (§2.2.4); we refreshed our minds with historical techniques and interfaces to durable data. Each one of these incrementally building on the shortcomings of the previous, in the following manner:

First, the single-level store emerged as a result of experimentation around the newly-introduced virtual memory, persistence being abstracted behind a single cohesive address space. The transparent buffering and write-back of data allocated on durable segments however proved to be a major hindrance for programs serious about their data manipulation.

Later, file systems surged as an intermediate abstraction to allow programs to structure their own data logically, isolated one from another on the same device, with an explicit control over disk operations and their order. Fault tolerance was yet minimally enforced on file chunks, with no protocol for recovery of program data.

Database systems arose with facilities to make higher-level data semantics readily available to applications. These included transactions with durability and atomicity constraints in respect to program data, and powerful means of locating and retrieving pieces of data from their attributes.

Persistence in object-oriented languages and object stores then focused on easing the translation between program data structures and the data model of database systems. One lost approach, orthogonal persistence, conceived persistence as a property of data objects, managed transparently from within the language runtime, with database functionalities being rebuilt atop. Conversely, time-favored object-relational mappers (ORM) have bridged relational databases with object-oriented programming, leveraging tools to manage object/relational mappings through a standardized API.

Orthogonal persistence hypothesis. All things considered, unrolling these historical facts may sounds like a snake biting its own tail to some extent. The appealing simplicity of the single-level store take on persistence was forgone due to the transparent nature of durable data management. Yet, the return of orthogonal persistence with object-oriented programming demonstrates a lot of interest for these concepts. At this stage, the historical double loss of orthogonality still raises questions. Whether it were just balanced-out by a lack of applicability and efficiency with conventionally available hardware, or because of more substantial deficiencies in usability prospects.

On one end, the SLS seemed to have potential to provide a powerful and flexible software development environment for its users. However, efficient implementation had requirements for dedicated and purpose-built hardware until then. Furthermore, as secondary storage remained block-based, the persistent virtual memory scheme was perhaps never truly well-grounded. That is because, from a cost/capacity/performance standpoint: flash memories never buried the ever-cheaper hard disks. IBM reported the I series operating system [112] to be built for the future, featuring a SLS for seamless transition to upcoming storage and memory technology. Apparently, in anticipation of a future that never came.

In detail, storage devices remained block-based, lacked byte addressability, with latencies in the millisecond scale; leaving purely software-based approach to persistence plentiful, in respect to performance and compatibility with conventional hardware. From a pure performance standpoint, no hardware made it relevant for persistence to be ser-

viced by virtual memory and the operating system directly.

The seamless integration of persistence within programming languages was nonetheless appealing. Persistent object systems indeed provided the same facilities for data persistence on conventional hardware through programming language idioms. However, they faced the exact same challenges of moving data between volatile and persistent block-based storage. Jordan and Atkinson [49, 183] argued that persistent operating systems could provide automatic persistence to the majority of language runtime. However managing external state such as network communications may have required breaking transparency, among the other challenge of retrieving sufficient information about application data to perform disk space reclamation or evolution.

We do not hope to place our definitive answer on the matter with the current discussion. Rather simply to lay the foundations as we will come back to orthogonality on the review present-day work. For instance, **AutoPersist** [288] that devised an orthogonally persistent Java virtual machine for NVMM (§2.8.4). We will duly provide an answer though as we motivate the design decisions of our own contributions in the next chapter §3.3.

Persistence functionality. Another quaint takeaway lies with the emphasis of object stores on relational database functionality. One might be thinking that, having long-lived data readily accessible from within programming languages data model and structures would eliminate the need for such functionality. Indeed, aside from the transactional facility that undoubtedly is useful to concurrency control as well, database-like queries and relationships may seem antiquated considering usual programming language data structures. Even data versioning and history-keeping had already been tackled by in-memory data-structures [126].

The question we are on about to raise here, is whether the full set of database functionalities is indeed relevant in the context of in-memory persistence; atomicity and durability aside of course. Recent NewSQL-style DBMS may suggest that relational data view is still of interest. With that, the ability to perform computations on scan results from very large data sets, or create new data sets from complex joins. We already know these are simply orthogonal to data persistence, as big data processing systems can already be designed to completely run on volatile memory. The thing that comes out clear though, is that facility for persistence should come in, easy to grasp for the programmer, but as also easily composable with existing abstraction for data manipulation. Answers can be found around software-based durable-memory transactions, that we review in §2.3.2.4, with NVMM-specific implementations later in §2.7.3.

Finally, as we close this section on early and historical work on data persistence, one might have noticed that we omitted to mention persistent memory. Although persistent memory is in direct lineage with the single-level store, this omission was, of course, purposely made as persistent memory is the main subject of the next coming section.

2.3 Persistent Memory

Data persistence has had a long history strongly rooted in databases and file systems, as reminded by the previous section. A parallel line of research though attempted to design a totally new computational platform with persistence at its core. Persistent memory (PMEM) was the programming abstraction proposed by these systems.

Until recently, volatile media were orders of magnitude faster than persistent ones. This fundamental difference much impacted the way systems are architectured. Recent advances in non-volatile random access memory (NVRAM) technology promise to however re-shuffle the cards.

In this thesis, we consider how this new technology can be harnessed to breed modern PMEM and incidently modern data persistence facilities. To this end, we first revisit past schemes for software-based persistent memory. This section organizes as follows. (§2.3.1) We define persistent memory as an abstraction. (§2.3.2) We then survey the earliest efforts in that field. In particular, PMEM as found in persistent operating systems (§2.3.2.1), object-oriented storage (§2.3.2.2), languages for distributed computing (§2.3.2.3) and finally lightweight durable-memory transactions (§2.3.2.4).

2.3.1 Definition

Conceptually speaking, persistent memory is an abstraction that allows efficient manipulation of long-lived data, using only memory instructions or APIs. Data contained in regions of PMEM outlive the execution of programs that created or updated them.

The major separation from other interfaces to durable data, lies with the nature of accesses to PMEM resident data. Indeed, programs may solely rely on memory interfaces to manage arbitrary data structures, regardless of their lifetime (transient or persistent).

The key to understanding PMEM, is that it must behave, from an application's perspective, as regular memory; except for data that should remain available across executions or system power cycles.

We propose to principle this informal description into three essential properties fulfilled by PMEM: *granularity*, *stability* and *efficiency*.

- ***Granularity.*** A first defining factor of persistent memory is on the uniformity of accesses to bits of data, relative to regular main computer memory. This includes the interface to data, and incidently, the grain of those accesses. In general, PMEM relies on common memory load/store instructions that allow byte-grain access to the data.

- ***Stability.*** PMEM-resident data must persist across program executions or machine power-cycles. Including unprompted ones as well, such as software or system-wide faults; much like any durable storage facility so far. However, this extends beyond durability of stored bits: data integrity must be retained in presence of failures, which might corrupt them, or the metadata they are associated to.

- ***Efficiency.*** Uniformity relative to regular volatile memory does not limit to access interfaces and *granularity*. It is fundamental that the access delays of PMEM remain within the same range. Programs should not be startled by reaching persistent locations or, ideally, not be able to differentiate memory kinds from operation latencies.

Benefits Persistent memory alleviate the need for programs to retrieve long-lived data through external interface to durable storage (file, database). All data, regardless of their lifetime, are directly addressable in their in-memory/computational representation. The direct benefits of which are threefold.

(Unique data representation.) Firstly, programs only have a single format of data to handle. In so doing, they benefit from the full expressiveness of programming languages, erasing the semantic mismatch between internal and external formats.

Moreover, in-memory data structures have long proven themselves as extremely useful composable abstractions that outstandingly facilitate programming. Their use no longer have to be restricted to transient data with persistent memory.

(Code base decluttering.) Secondly, persistent memory helps reducing programming efforts, boosting productivity—a highly-valuable characteristic from a software engineering perspective.

For starters, because programs only deal with a unique view of data, complex and complicated algorithms are no longer required to keep these multiple copies of data mutually consistent, in respect to system faults.

In addition, programs are no longer directly in charge of communicating with storage, or managing the structure of data on-disk. Therefore, a sensible amount of codes are no longer required in programs: translating storage data format, organizing disk space, indexing stored data, or efficiently caching long-lived data in smaller main memory. Less code decreases overall complexity with its usual benefits (e.g., more predictable performance).

(Performance gains.) Thirdly, since persistent data are directly addressable, programs no longer have to manually move them back and forth from durable storage. The aforementioned code simplifications also reduce the complexity of operational logic, leading to better performance. Namely, conversion operations - or the lack thereof - to translate storage data format; or the mechanisms to manage disk space and file chunks.

2.3.2 Early occurrences

Earliest instances of persistent memory throw us back to Multic's virtual memory we already covered in §2.2.1. It did not strictly register as persistent memory though, as stores to persistent memory segments were buffered and written back asynchronously, irrespective of program instruction order. Unless all dirty pages were flushed to secondary storage, segments on secondary storage were not in a consistent state; therefore, not resilient to system faults. Although such transparent design lacked the facilities to implement *stability* protocols in presence of failures, it laid the premises for a body of research focused on providing persistence as an operating system's service.

We propose in what follows to introduce early forms of persistent memory as the virtual PMEM abstraction found and exposed by subsequent single-level stores and persistent operating systems. Persistent memory also has roots in object storage (introduced in §2.2.4) and distributed programming, as software-defined abstractions for object persistence. Finally, transactional memory systems in some instances also provided durability guarantees and, thus, an abstraction akin to persistent memory.

2.3.2.1 Persistent operating systems

The idea of persistent operating systems came from persistent application systems [48]. As a whole, they expected better support than the ephemeral virtual address space and file interface offered by conventional operating systems. A famous paper argued that they

did so poorly, for the specific case of DBMS [301]. The field was thoroughly summarized by Dearle et al. in [115, 116, 171], who worked on the Grasshopper persistent kernel [117].

They enunciated as a major requirement that any data object must live in a single space with a uniform addressing scheme were they were both stable and resilient to system faults. In other terms, they exposed a persistent virtual memory to processes.

Content of processes' virtual memory was usually backed by a large persistent store that allowed for concurrent processes to manipulate persistent data, while the store remained stable and resilient to system faults.

Resilience was achieved with checkpointing of the backing store, either explicitly through a custom APIs or periodically for transparency.

Examples of such systems include **Monads** [28, 188, 270, 271], **KeyKOS** [158, 206], **Eros** [284, 285], **Grasshoper** [113, 114, 272] and the **IBM i** -series operating systems [297]. We briefly go over them in order to demonstrate the impracticality of persistent operating systems and issues they raise.

The Monads-PC project began in the mid 70s at Monash University in Melbourne, Australia, with the aim of building a computing platform that improved security by encapsulating all data in capability-based modules. Their single-level addressing scheme was supported by purpose-built hardware, that allowed for access control and protection, directly serviced by their hardware-assisted virtual memory paging mechanism. Where Multics wrote back disk pages to the same location, Monads' storage management unit [271] updated dirty disk pages out-of-place, utilizing *shadow-paging* from Lorie [221]. Periodic checkpointing was used to stabilize new consistent versions of the store. This operation consisted in the consolidation of the shadow page table and write back of all dirty pages to disk, followed then by the atomic installation of the root page with a single disk write using Challis' *atomic update* [84].

KeyKOS also tackled memory persistence generically through system-wide periodic checkpoints. Instead of *shadow-paging* that constantly moves around pages and leads to a data locality mismatch, KeyKOS relied on two dedicated swap areas on disk to checkpoint memory content. Every 5 minutes, the system would stop all process activity, for around tens of seconds under normal load, in order to put all dirty pages in a copy-on-write state. After resuming processes, dirty pages would gradually be migrated to disk in the swap area that was not designated for recovery. The copy-on-write state of dirty pages prevented programs running concurrently from further updating the pages being migrated and ensured the whole state was consistent while being written back. Once all pages were evicted, the checkpoint header was atomically updated such that the roles of both swap area were reversed, effectively committing the new checkpoint. Finally, while the old checkpoint area is prepared and recycled to host the next checkpoint, pages in the current checkpoint area were migrated to their home location on disk.

KeyKOS additionally provided a custom API to flush to disk a specific page in the next checkpoint area, breaking transparency to explicitly enable applications to build custom recovery and implement full transaction processing, as they mention.

EROS proposed a similar persistence mechanism directly evolved from KeyKOS. In addition of taking a snapshot of the entire machine state, Eros ran a battery of consistency checks including unitary testing of essential kernel data structures. The

goal was to ensure no bad checkpoint would be committed, because in a setting were the whole system state is persisted, any inconsistent checkpoint lives forever. If tests were to fail, the system would be rebooted without committing the checkpoint being taken.

Grasshopper solely relied on a custom API to perform per-process snapshots, and completely separated from the *stop-the-world* periodic checkpoint mechanisms found in previously mentioned work. Now, because processes might share memory and communicate through IPC, similarly to distributed systems, individual process snapshots are insufficient to persist a globally consistent state. Grasshopper proposed to allow optimistic per-process snapshots, and to asynchronously rebuild a global consistent view from data causal dependencies. The kernel detects causal dependencies by tracking the dirty page list of each process at each snapshots, and maintains those causal dependencies between processes as vector clocks for recovery purposes.

IBM i along with its predecessors AS/400 [296] and System/38 [8], were a commercially successful series of operating systems that all integrated a patented single-level store. The operating system, runtime and compilers all worked in concert to provide persistence. The IBM compilers semi-automatically inserted API calls for persistence. This approach had the restriction of tying users to a monolithic system stack and a restricted set of supported languages.

Limitations To conclude on persistent operating systems, their uniform single-level memory adheres to our requirements for persistent memory. As such, they allow for straightforward persistent programming, while stability and resilience are guaranteed by the underlying store. However, essential design flaws make them impractical with regard to data persistence. We detail them below.

Programs are prevented from distinctively managing transient and persistent states. We do not need to stress the fact that applications want only a subset of their data persistent, and transient state to be reconstructed on restart. Otherwise, any unintended state would destructively become permanent, and no longer be fixable by rebooting; software evolution would become an engineering nightmare. EROS perfectly illustrates this with its online consistency checks enforced at each snapshot. It is unreasonable to expect any serious program to implement workarounds as costly as a battery of online tests before each checkpoint, to assert whether it stayed semantically sound as well.

Virtual memory as supported in commodity hardware is a wrong fit in single-level store. Contrary to persistent object systems we reviewed in §2.2.4, operating systems have a black box view of memory pages. Where persistent language runtimes could translate in-memory data structures to enable sharing (e.g., swizzle native pointers), persistent memory serviced by operating systems must adopt an addressing scheme that crosses the process boundaries and their lifetime. In particular, uniquely identifying any data object with native pointers in a global virtual address space with support for protection and access control. Monads-PC, KeyKOS or IBM's operating systems all depended on purpose built hardware to this end. EROS and Grasshopper built on previous work from single-address-space operating systems [85] (SASOS) to operate on commodity hardware. In addition to performance overheads, a global and permanent address space is not easily composed with essential security techniques, for instance

address space layout randomization (ASLR).

Fully transparent resilience protocols can not create semantically consistent application state snapshots. Periodic checkpoint techniques are not only limited by the capturing frequency the performance of storage hardware would allow for; but also as appearing consistent. Monads-PC is indeed the only of the above not to expose an explicit API for resilience, even though they were all built around a persistent kernel and made processes persistent as well. Reason being that always resuming a process execution, or even a complete system, from a previously known state does not make it consistent from an external perspective. Just as Grasshopper's optimistic process snapshots needed to be stitched back into a causally consistent system-wide snapshot; processes participating in distributed protocols might appear faulty externally when recovering at an arbitrary execution step, even when their local state appears consistent in isolation.

Dependence on custom APIs or whole system software stack. Explicit APIs are required for persistence but were not made compatible with ubiquitous operating system interfaces (e.g., POSIX). Persistent operating systems investigated interesting concepts to enable cooperation with persistent language runtimes; but as illustrated with IBM i, the resulting monolithic software stack for production-grade applications was a huge technological commitment companies have struggled to migrate away from.

To summarize on persistent operating systems and their approximation of a system that never stops; they promise programs they will no longer have to implement recovery and rebuild transient application states inferred from persisted data, which might sound handy. However, as we discussed previously, they do not reduce the software engineering burden in persistent programs, but rather move its inherent complexity around without eliminating it. The real world is made of buggy software and faulty hardware; these persistent operating systems would simply force users into a bunch of workarounds because of deficient design of their persistent memory. In particular, apparent simplifications for applications and increased developer productivity are largely undermined by the cost and commitment of re-engineering software and runtimes for their custom APIs.

In spite of all this, recent academic work proposed to revisit single-level store [306], motivated by the advent of new low-latency storage technology. The concept of persistent operating system is also actively pursued by PhantomOS [129, 344] since 2009. It somewhat parts from the idea of persistent memory and instead rely on a virtual machine abstraction with periodic snapshots. Their target is also limited to desktop and handled devices and incidently only client programs, where issues we pointed out above are of little concern.

2.3.2.2 Object-oriented storage

The area of persistent object systems and object-oriented databases, we already discussed in §2.2.4 for language/programming prospects, has also readily proposed persistent memory schemes to be used as a base abstraction for persistent stores above object storage management units. In comparison to persistent operating systems that consisted in *system-managed persistence*, the work described here could register rather as *application-directed persistence*; meaning programs are accessing persistent memory along with specific tools to, for instance, coordinate resilience protocols and ensure their objects remain semantically consistent.

Earliest PMEMs related to object persistence were designed for special-purpose object-oriented hardware architectures. Language runtimes were tightly coupled to the system and hardware; PMEM, among other services such as garbage collection, were supported by the whole system software stack. Even though persistence was system-supported, they smartly did not fall into the same pitfalls and simplification trap the persistent operating systems did. We can see the context of object-oriented programming obviously made them consider issues that works from our previous section did not even raise. Example systems include Intel's iMAX 432 [186, 261] Ada machine and its filing system, or Thatte's recoverable virtual memory [304] intended for TI's Explorer Lisp machine.

The iMAX 432 filing system outright rejected uniform single-level address space, and favored a two-space approach. « We rejected the one-space model because of the difficulty in maintaining object consistency in the presence of system crashes » [261]. Persistent objects were referenced by symbolic names from the filing system, and lived in the *passive* space. They could be activated by opening access descriptors, a copy of the data would then be instantiated in the *active* (memory or swap) space. The two diverging versions were brought back together in stable storage with either: (1) an explicit commit of the object, (2) an automatic policy implemented in the Type Manager, (3) garbage collection of the active version. Multiple objects could also be committed at once, and resilience was achieved through a redo-logging-based technique. Garbage collection was used to manage the active space for its superior productivity in dynamic environment; however an ownership scheme for the passive space was employed, as they anticipated the impracticality of garbage collection in large persistent spaces.

Thatte's recoverable virtual memory design on the other hand held onto the single-level uniform memory model, but without strictly advocating for transparency, which alleviated most of the limitations. The scheme relied on a specific virtual address fixed at a disk location, to host the *persistent root*, and enforced *transitive persistence* by preventing garbage collection of any non-transient object. Hence transient objects were simply identified as being referenced in the transitive closure of the transient root and not the persistent one. The recoverable memory was periodically checkpointed incrementally, at the whole system scale, for fault tolerance in respect to system or disk crashes. However, they realized system-wide snapshots were insufficient to construct resilient and consistent objects. For that reason, they implemented on top of it a persistent object manager to organize persistent name spaces as well as to provide a transactional facility with undo and redo logging. From their experience they admitted this abstraction was superior in terms of flexibility and representational power, for the construction of database systems on top of it.

Later, around the mid 80s and early 90s, persistent object stores backing PMEM for conventional hardware and operating systems started to show up. First, as fully supported by software and simple files, like in Brown's stable store [72, 73, 94]. Afterwards, using modern operating systems memory-mapped files (e.g., *mmap* in Unix, SunOS) and hardware-assisted virtual memory. Examples include the Texas Persistent Store [292, 293] and QuickStore [320].

The CPOMS (Persistent Object Management System in C), we previously referred to as Brown's stable store, supported PS-algol, a programming language for orthog-

onal persistence we discussed earlier. The CPOMS cooperated with the PS-algol abstract machine in order to perform address translation or checkpointing in the store. For instance, it swizzled pointers lazily (see §2.2.4), that is, upon first access. Virtual addresses were retrieved from a global in-memory repository that kept a mapping between virtual addresses and persistent identifiers of all instantiated objects. When no mapping existed for the persistent identifier being accessed, it meant the object had to be retrieved from the persistent store. Checkpointing of the persistent store was directed from PS-Algol transactional API; precisely, from the explicit database-style commit operation. The store itself was managing disk space through simple Unix files, and implemented atomic checkpoints with a page *shadowing* technique. The disk space was split into two regions: one containing the current versions of the pages, and the second *shadow* region held undo copies of them. Writes were instrumented to create an undo copy of pages in the *shadow* region for every first modification following the previous checkpoint. An on-disk bitmap ensured it was only created once. Recovery involved reading the bitmap and writing back the modified pages from the *shadow* region onto the main one. Storing undo copies of pages had the convenient property of allowing in-place updates since the working copy was never relocated. Not relocating pages has the additional benefits of preserving the write order as well as the spatial locality of accesses, and last, to remove the need for instrumented reads.

Texas is a persistent store that brought persistence to C++ in a lightweight and portable way. They conceded sharing common design ideas with ObjectStore's pC++ [205], without the OODBMS functionality though, as it was not a commercial product. It was implemented as a concise C++ library, and worked with conventional compiler. Persistent memory was materialized through an alternative heap manager that was backed by a memory-mapped raw Unix partition (*mmaped*) to avoid the file system overheads. Persistence was then identified at allocation time, for any type, with no class code modification; objects could be linked to a persistent root object for recovery. Explicit heap checkpointing was ensuring object resilience.

The core idea was to use conventional virtual memory abstractions to implement address translation and write logging. The technique was coined « pointer swizzling at page fault time » by Wilson as it first appeared in [321]; and had the main benefit of eliminating substantial software overheads of lazily unswizzling every single pointer when dereferenced. To this end, memory pages were initially protected with *mprotect* and pointer swizzling was performed at page fault time, with page-wise granularity. Locating pointers within pages was achieved with a custom C++ preprocessor that was used to generate C++ type descriptors. No caching needed to be implemented, as it could conveniently be offered as part of conventional virtual memory. In all, the store made sure unprotected pages contained at all time only hardware-supported addresses.

Likewise, checkpointing relied on a simple pagewise *write-ahead logging* approach in a first iteration. Write protection was used to detect alteration on pages in-between checkpoints. Later on, they moved to a *log-structured* technique for storage, in order to reduce bandwidth usage. Modified pages could be written out in a sequential way, with no consistency concerns; and the indexing structure atomically updated afterwards to designate new page versions. The disk storage component was also

working directly with a raw Unix partition to further avoid irrelevant caching in a file system's buffer.

Down the road, in [187], they experimented mixing pointer swizzling at page-fault time with finer grain swizzling through C++ smart pointers. As they reckon the coarser grain of pages for swizzling might not be appropriate for data structures that exhibit poor locality of reference, such as sparse indexing structures.

In all, Texas experimented with portable ways of implementing persistent memory; with either no or little code modification, depending on whether client code needed to differentiate transient from persistent data. They showed working at page granularity could be made sufficiently efficient, compared to object-wise more intrusive techniques, in regard to both swizzling and checkpointing. As a research-oriented prototype, Texas' PMEM sure lacked functionality and remained single-user with no support for client-server mode for instance.

QuickStore had an almost identical approach to PMEM in C++ than Texas. It also leveraged hardware-supported virtual memory though it did not swizzle at page fault time. They also differed in that QuickStore relied on an external DBMS client (EXODUS [80]) for stable storage, instead of a custom file-based storage layer. This made support for client-server mode rather straightforward and opened the possibility of extending transactions to both concurrency control and recovery. Precisely, their PMEM was overlapping with the DBMS client buffer pool (with `mmap`) in order to avoid an extra data copy. They still relied on page fault to read-in whole pages to the client buffer pool; but they did not swizzle pointers before writing out pages to storage. Instead, they persisted metadata relative to the mapping of disk pages, and attempted to remap them at the same location. Swizzling was then occurring as pages were pulled in from storage and the mapping had already been reused. In that case, pointer values were rebased from their previous location onto the new one.

Regarding recovery, a page-diffing technique was implemented to minimize the amount of logged data during transactions. Pages would be write-protected at the start of a transaction, and their original values copied when first modified. At commit time, the two versions were diffed and log entries generated for every mismatching byte sequences.

Further down, they admitted that relying on page fault was not enough to support object identity and that some silent errors could not be avoided. For instance, upon dereferencing dangling pointers to some deleted objects, no error would be flagged. The page could still be active, even in subsequent program runs, and the page could be pulled in when accessing other objects. If a new object occupies the free-ed space, then the dangling references would reference this new object. The additional cost of checking references would be prohibitive. As nothing is done to prevent dangling pointers, they may still happen as any other programming bug; failing silently is, especially in this instance, a recipe for data corruption.

To summarize, persistent memory as proposed by object-oriented stores corrects almost the 4 issues we raised with persistent operating system:

- **They always allow for distinct allocation of transient or persistent data.**

- **Virtual memory is properly made use of**, even smartly leveraged as a mean of intermediating store operations without changing binary code.
- **Object resilience and semantic consistency is possible**, as the application is directing atomic crash-recovery of the store.
- **They stay portable and easily integrate with existing software stack**, relying on generic compilers and all using the same `open`, `close`, `persistent_allocation`, `persistent_root` interface.

In turn, refraining from persisting whole process states and virtual address spaces incurs challenges specifically towards persistent reference management.

- **Pointer translation between stable storage and in-memory virtual addresses.** Pointers can be (un)swizzled eagerly, lazily, with fine or coarser granularity. Systems above found a good trade-off in using pagewise swizzling for general purpose, and fine-grained only in indexing data structures.
- **Dangling references.** Not all systems resort to garbage collection on persistent heap, dangling references might happen there as the result of programming bugs. In comparison to traditional allocators though, the *mmap*-approach would never raise explicit errors and always induce silent bugs as detailed above. Although the problem is pointed out, no applicable solution is provided.
- **References from persistent to transient data.** This one was not addressed by any of the above, yet could lead to silent bugs as well. They could not even nullify transient references during swizzling, as object types are entirely orthogonal to persistence.

Finally, preferred interfaces for atomic and consistent updates of persistent object data were transactions in multi-user environments, and simple explicit store checkpoints in single-user application-embedded stores. Techniques themselves to stabilize and provide resilience in the stores were all comprised in the usual *shadow-paging*, *log-based* or *log-structured* families, introduced by database systems. Following the fact that all used disks as storage media, all favored techniques that minimized the amount of data written and which avoided seeking while checkpointing to maximize write bandwidth.

2.3.2.3 Distributed computing

Although bringing up distributed computing in a persistent memory discussion might feel a bit of a stretch on first thought, data persistence is a fundamental concern of the field, as it had been for database systems before. Building distributed programs is inherently complex. Memory abstractions were sought out to facilitate the process, for instance, by hiding away message-passing paradigms or complicated communication protocols. As we have already discussed, stabilizing semantically consistent data to storage in multi-user environment with shared data requires coordinating concurrency control mechanisms with data resilience protocols. That is to say, memory abstractions tackling distribution need to account for data persistence as well. The surging ideas in distributed systems from the mid 80s and early 90s then put all that together and attempted to tackle these new challenges with either operating systems, programming languages, or object storage. We illustrate that with a few examples:

Clouds [109, 110] was a distributed operating system developed at the Georgia Institute of Technology in the mid 80s. The two base abstractions were *objects* and *threads*. All data were encapsulated in objects, and threads executed within objects. Objects had their own persistent virtual address space and their own procedural interface to provide operations for threads to execute. Although each object implemented a single-level store, user-level utilities enabled management of a transient heap and persistent heap separately. Running threads could cross machine boundaries by invoking operations from remote objects, but data were only moved between objects as argument or return value of operations. Multiple threads might have executed concurrently within the same object with the help of consistency policies specified when invoking operations.

Argus [218, 246] was a system and programming language for distributed programs, created at MIT around the same time. It provided built-in atomic data types that guaranteed both indivisibility and recoverability. Indivisibility meant that « the execution of one action never appears to overlap the execution of any other action » and in that is similar to linearizability [163]. Recoverability required that the overall effect of an action was all-or-nothing. To this end, all data were guarded by an encapsulating unit that internally used read-write locks and accessed stable storage. The model for persistence was similar to Thatte's: a collection of fixed root objects and transitive persistence enforced by garbage collection.

Thor [217] was a object-oriented database system for use in distributed environments, also developed at MIT. It relied on strict object encapsulation to provide safety from strong typing. Automatic memory management was employed in the store to both implement transitive persistence and prevent dangling references. Object methods were also encapsulated and stored on media, described in an abstract and statically typed language, to be then safely instantiated and manipulated from multiple language environments. Persisting methods is a way of safely dealing with code evolution without the risk of linking against the wrong implementation version. However, a lot more work is required: Thor was dynamically generating stub-objects from stored implementations upon data retrieval. They rather considered this approach to save work, as implementations needed to be written only once for many languages.

2.3.2.4 Durable memory transactions

Transactions are admittedly everywhere, persistent memory is no exception. The idea of transactional memory (TM) started out as a hardware solution (HTM) for optimistic concurrency control [162]; and software transactional memory (STM) was the fruition of research work attempting to pass by restrictions of HTM [286]. In that spectrum, recoverable (or persistent) software transactional memory were conceived to provide durability and atomicity of operations. The results were lightweight and portable abstractions that only brought transactional facilities in programs, that could be applied on any block of code, but without the database functionalities and specific data models or types.

RVM [276], as in lightweight recoverable virtual memory, is an important piece of work that was widely referenced and steered the first modern persistent memory designs. RVM was a minimalist and portable software library, with transactions that provided only atomicity and persistence over a virtual memory address range. Its transactional facility was meant to be easily composable, and gave user independent control over atomicity, persistence and serializability properties of transactions. Atomicity was always guaranteed, but other properties could be layered on-demand with explicit flags. For instance, about durability, flushing the transaction log on commit could be omit-

ted, or delayed to a later point in time with an explicit `log_flush` operation. Each transaction spanned over a finite number of memory ranges, which were declared with an explicit API call. Ranges were copied on `tx_begin` to different memory locations for potential aborts (undo), `tx_commit` then used WAL (redo) to safely write them out. This algorithm was simple and effective, and needed no recovery aside from the log, because only committed data were stabilized on storage. Unclean working data were kept in-memory and thus swept with crash faults. They declared RVM as a building block for anywhere persistent data structures were needed to survive local client failures (think about file system metadata management for instance, or runtime systems for persistent languages). In particular, in [251], RVM was used to implement concurrent garbage collection of a persistent heap.

Rio Vista [222] is another recoverable transactional memory that improves performance over RVM. In essence, RVM copied database segments (transaction ranges) inside application main memory for processing, and performed two more data copies (undo & redo) for aborts and safe write back. In contrast, persistent stores used to directly mmap portion of the database inside applications and relied on virtual memory to trigger safe data movements. Rio Vista instead assumed a reliable file cache: Rio detailed in [86], as its memory-mapped stable storage. The Rio file cache coped with power faults using an uninterruptible power supply, and software memory corruption (e.g., wild memory stores) with virtual memory protection. As such, Rio Vista considered data reliably stabilized as soon as it entered the Rio file cache. In particular, individual store instructions to a mmap-ed region were considered persistent immediately, with no `msync`. RVM performance was increased by 20x by running on Rio, but Rio Vista further improved performance from tailoring its crash-recovery techniques for Rio. In detail, Rio Vista did not need to use any system call for transaction processing, and the redo WAL could be completely omitted, saving one data copy. Omitting WAL was possible because data were altered in-place in the file cache during transactions, on reboot, transaction abort undo logs were used to revert any uncommitted transactions. Put together, Rio Vista avoided any system call and did copied data only once, all resulting in a 5µsec overhead per transaction.

One controversial medial thought before moving to the last system. The use of a battery-backed system might feel like cheating, but beyond that, it demonstrates that even in this case, recovery is still needed for crash-consistency. Plus the fun of toying with the idea of a stable directly-addressable main memory, which resembles at lot prototyping for NVMM.

Stasis [280] was a transactional storage framework from UC Berkeley that aimed at filling the gap between file systems and {R, OO, OR}DBMS. Its authors argued that even lightweight embedded databases, such as the much well-known BerkeleyDB, were not suited for system programming. They claimed that easing the implementation of transactional systems required more powerful primitives that could degrade their guarantees on-demand, or be extended for fine, hand-tuned data movements. A view that shaped Stasis into a persistent object system that combined traditional ACID database semantics with a configurable and extensible interface, meant for implementation of sophisticated performance optimizations. At its core, Stasis provided page-based transactions. To this end, it used a customized variation of a well-known database technique, ARIES [233], that generalizes *write-ahead logging* to any block of code. Additionally, the framework supported tweaks to turn down ARIES guarantees on a per-operation basis, mixing physical (page) or logical (operation) logs and flexible locking options. Over the course of their experiments, they demonstrated the flexibility of the framework by denot-

ing how concise implementations of a number of systems would be above Stasis, or how some ad-hoc storage approaches could be supported. For instance, they showed with Stasis that the RVM algorithm with concurrency support could be implemented simply from Stasis flexible interface and user-defined operations—for the record, RVM and Rio Vita had relegated concurrency or isolation to future work and upper layers. Instead of a rigid API that forces programmers into work-arounds or re-implementing a whole new storage architecture from scratch, Stasis addressed their needs with a transactional storage library that could be bent in many ways to accommodate various usage.

2.3.3 Summary

Before turning into works of the new century, we had to check out for now-antiquated forms of persistent memory. We brought up designs abiding to our definition with appropriate *granularity*, *stability* and *efficiency* (§2.3.1) from the territories of persistent operating systems, persistent object systems, distributed computed or transactional systems.

Persistent operating systems (§2.3.2.1) we reviewed took it upon themselves to stabilize whole-system states, including processes with their execution context and virtual address space. The resulting kind of PMEM promised hands-off data durability and recoverability to applications. Appealing claims that were shortly muted on second thought: having programs forfeit transient state is a recipe for disaster in a world of buggy software. Design deficiencies of persistent operating systems could be coped with APIs allowing explicit checkpointing, but admittedly with no prevalence over the abstraction of user-level persistent stores.

Reason being that persistent object stores (§2.3.2.2) demonstrated the same degree of support for transparent checkpointing of in-memory data structures, whilst additionally allowing for more flexibility with explicit APIs. Specifically, they considered mapping in virtual memory database segments, relied on virtual memory protection to coordinate data movements, and orchestrated store checkpoints with database transactions. The database interface allowed for distinguishing transient and persistent objects at allocation time, but without language support and only a limiting coarse grain (page-wise) control over data. Here, it is worth noting that this kind of PMEM could not easily maintain objects' referential integrity or prevent persistence bugs factored by dangling pointers.

We then found language-level support for persistent memory in object oriented programming and distributed computing (§2.3.2.3), above persistent stores. Data strictly safeguarded behind language objects with guarantees towards consistency or recoverability as part of type specifications. Blending with garbage collection support for transitive persistence and displacement of dangling references.

Finally, portable transactional libraries (§2.3.2.4) proposed themselves as lightweight persistent memory to ease storage interaction in system programming. Their stripped-down transactions ensured only atomicity regarding crash recovery, and their explicit API left no hint of transparency for maximum control. They were built with the underlying goal of offering a low-overhead transactional interface to on-disk data structures — a valuable base building block for persistent stores or language runtimes.

Perhaps the most obvious overarching theme between all of the above is resilience of persisted data. Which all sample work provided through atomic commit protocols inspired by well-known techniques from database transactions. Whether it is *shadow-paging*, *log-based* or *log-structured*; reviewing those systems got us a feel for the com-

plicated trade-off space of transaction implementation. Navigating this space properly requires understanding the reality of the under layer (storage), but also client requirements and operation archetypes. That is to say, factoring in that modern persistent memory will not rely on spinning disks, the change of too many parameters may throw off the deductions of prior studies.

Another common topic, is allocation routines for the persistent heap. Crash consistent allocation is crucial to avoid leaks, but was rarely brought up. Especially as most of the scheme we reviewed were designed for « unsafe » languages, where it is common to assume that proper memory management is the task of programmers. Too little effort were put into describing how heap recovery stays consistent with user maintained references. For instance, no technique preventing object allocation with no pointer installation inside of a transaction were described, or heap reconstruction procedure to recover leaked spots. Fortunately, modern persistent heap design cover for both, as we will see in §2.7.5.

We must duly note that all systems presented above made use of conventional disks for stable storage (perhaps with the exception of Rio's battery backed file cache). As such, they were presented with the challenge of moving data back and forth between different media. Moreover, their protocols for resilience, were all optimized for sequential access and block-based devices. The work conducted in this thesis instead focuses on emerging storage technology that lifts the two preceding statements. Before tackling persistent memory opportunities and recent work inspired by that media, we are obliged to properly introduce it, NVMM, in the next section.

2.4 Non-Volatile Main Memory

Non-volatile main memory (NVMM) and persistent memory are now often reciprocally confused. We however decided to define persistent memory as an abstracted programming construct, and further refer to NVMM as hardware technology or media matching the following attributes:

Def. Computer main memory that retains data without power, have latencies of the same order of magnitude as traditional random-access memory (DRAM) and storage-class capacity.

In that, NVMM descends from both memory and storage, mixing non-volatility with granular direct access. Those properties of NVMM are commonly denoted *direct-byte addressability*, *low-latency*, *durability* and *density*.

- ***Direct-byte addressability.*** This is the game-changing factor of NVMM. Modules have the form factor of regular DRAM modules, physically connect on the same slots, and sit on the processor's memory bus. Processors have then direct access over NVMM physical addresses, enabling data to be manipulated exactly in the same way, through common load/store and other byte-wise instructions (atomics for instance).

- ***Low-latency.*** DRAM access, for instance with DDR4, features around ten nanoseconds of read or write latency. NVMM technologies expect several tens of nanoseconds, and no to exceed the hundred. This alone accounts for claims that NVMM may close the memory-storage gap.

- ***Durability.*** Non-volatility was a characterizing trait of storage technology. That data remained stable without power applied. Beside storage use cases, this factually makes NVMM an appropriate low-power memory alternative.
- ***Density.*** NVMM modules are several times more dense than DRAM at the same price point, but remain quite far off from flash memories or spinning disks. Density is perhaps the main determinant factor in future wide adoption of NVMM, as it directly correlates to cost of the system. Spinning disks, for instance, are vastly inferior and do not make computer run any faster or smarter; yet we have carried on engineering complex storage architectures that offset their weaknesses in order to reap big profits from their unrivaled cost-efficiency.

PMEM support The striking consideration at this point is how close NVMM as a hardware device is to the essential components of persistent memory. It strictly adheres to *granularity* and *performance* thanks to its byte-addressability and low-latency.

The main concern however is that non-volatility alone is not sufficient to reliably provide data *stability*. In current computing systems, CPUs use write-back caches that are volatile and hardware that might not reflect program order when evicting data to memory modules. Meaning programs can carry on their execution past a store instruction, with data that have not yet reached the persistency domain of NVMM modules, or that have but in a different order. Possibly leading to loss of data integrity or inconsistencies when considering recovering from potential system failures. We elaborate on that in §2.6 along with other challenges faced by NVMM as a support for PMEM.

NVMM is nonetheless game-changing when it comes to persistent memory. We previously listed the three direct benefits of PMEM as *unique data representation*, *code base simplification*, *performance gains*. Additionally, direct byte-addressability of NVMM allows for a single copy of the data to serve both computations and storage; when anterior PMEM had to transparently manage the two versions of the data living on memory and durable media. That single copy brings *unique data representation* and *performance gains* to a whole new level:

- Erasing the need for data movement across media, making recovery rapid and warm-up-free.
- No more (un)marshaling or (de)serializing to translate between storage and memory data format, saving precious CPU cycles.
- Managing those two data copies was a continuous complex trade-off between efficient caching and preserving consistency; but NVMM altogether has potential to eliminate the need for caching - withal making PMEM simpler and practical.

2.4.1 Usage & system interface

Reference documents specifying system support for NVMM are: the Non-volatile memory Programming Model (NPM) [295] edited by the SNIA (Storage Networking Industry Association) and Intel's book on « Programming Persistent Memory » [278]. The SNIA is an industrial conglomerate with a working group on NVMM, responsible for development of reference libraries and programming models for upcoming devices. Intel, as pioneer NVMM vendor, also played a defining role in system support for NVM. Three distinct use scenarios are covered: main memory (volatile), block-device (storage), and persistent memory direct access.

(Memory mode.) Memory configuration is handled directly by machine's firmware and requires no change in system software. DRAM is used as a last-level cache for NVMM, and the firmware then blends DRAM and NVMM physical ranges as volatile memory when exposing them to the operating system.

(App-direct mode.) When memory configuration is disabled, NVMM modules are exposed by the machine's firmware as persistent memory ranges in ACPI tables. The operating system can then detect the special ranges and configure them freely, as volatile memory, block-devices, or direct-access devices.

Modern operating systems (e.g., Linux, Windows) have user-space utilities to allow these configurations. Users first isolate or interleave NVMM modules with namespaces; then each namespace can be configured as: *(i)* volatile memory, in a CPU-free virtual NUMA node, or *(ii)* block devices, with optional support for DAX (direct access) [2].

DAX-enabled block devices can be used like any other, starting with the creation of a file system on them, and followed with regular block I/O, with support for memory-mapped files (`mmap`).

(DAX-mmap.) NVMM direct access (DAX) is then requested with a specific `mmap` flag on DAX block-devices. When DAX is enabled, the operating system's I/O subsystem is bypassed and the processor's MMU is programmed to map NVMM physical ranges directly in the process virtual address space. With DAX `mmap`, programs may access uniformly, without any intermediate software, an *hybrid virtual address space*, made of both volatile (DRAM) and durable (NVMM) memory regions. We describe later in §2.6.2 the subtle programming model that enables crash-consistent updates to NVMM regions. Intel has released its "Persistent Memory Development Kit" (PMDK) [19], a suite of basic low-level tools (memory allocators, logging facilities) to aid DAX NVMM programming in C or C++.

(NVMM-native namespace.) One may have noticed at this stage, that current direct-access method for NVMM modules leverage file system namespace and metadata. Although this approach has the benefit of offering a file system's basic form of naming and protection, it also restricts NVMM use in applications. For instance, dynamically extending the range of persistent memory mapped in a process. More NVMM-native solution to direct access have been presented in research work, such as **HEAPO** [172] which implemented in Linux a directory to service NVMM objects in applications, with their own dedicated sharing metadata for ACLs and protection. Keep in mind though, this topic do not seem to be very active anymore. The DAX mmap method remains the only one employed ever since technical documentation for Optane modules have been available.

In all, NVMM can currently be exposed as either: a slower tier of volatile memory for large memory applications, a block device for legacy storage applications, or as explicit non-volatile memory. That last one is obviously the novel support for persistent memory.

Real NVMM do exist! We have been fortunate enough to have access to modules for almost the entire duration of the PhD, as we got our hands on some by the end of the first year. We detail next the commercial product that motivated our project.

2.4.2 Intel's Optane DC Persistent Memory modules

Optane DC Persistent Memory is the commercial name under which Intel and Micron announced in 2015 their upcoming jointly-developed NVMM modules. The technology commercially labeled 3D-XPoint was first released as traditional NVMe SSDs in 2017

before being available as memory modules in April 2019. These, as the first ever commercially available byte-addressable non-volatile memory, were enough to rejuvenate interest in persistent memory. Their announcement alone spurred various research communities to consider the implications of NVMM, for instance in system design, storage systems, data management, or entire systems.

Compatibility Intel Optane DC PM modules are compatible with Intel's x86_64 server platforms, starting with Cascade Lake microarchitecture (2nd generation Intel Xeon Scalable platform, debuted in April 2019) and onward. With this generation, Intel has introduced ISA extensions specifically for NVMM: instructions for cache line persist ordering, essential to reliably provide data *stability*. More on that later in §2.6 as we review NVMM programming models.

Endurance Intel's NVMM has a limited wear reliability, as opposed to DRAM for which endurance is approximately infinite. Optane specification sheets announce a ballpark of 10^8 full write cycles, when DRAM is considered above 10^{16} and flash SSDs sit around 10^5. Intel's marketing claims this to equate with 5 years of continuous abuse, at maximum write bandwidth 24/7, 365 day a year. This shortcoming forbids Optane to fully substitute for DRAM, which leads in practice to hybrid/heterogeneous memory architectures using both NVMM and DRAM.

Performance characterization On the release of Optane modules, multiple research studies [175, 257, 331, 338] have disclosed raw performance figures at both micro and macro levels. This proved remarkably helpful in understanding the specificities of Intel's technology to later design efficient schemes for real NVMM. In particular, these studies found that Optane modules are **not truly random-access, but block-based, with limited bandwidth compared to DRAM.** In detail,

- The nominal read latency of Optane DC memory is about 3x slower than DRAM for random patterns, but only 2x slower for sequential access. Write latency is similar to DRAM, that is because, according to Intel's specification, writes are considered persistent as soon as they reach the processor's integrated memory controller (iMC) write pending queue. We elaborate on persistency domains later on in §2.6.3.

- Optane memory internally uses a 256B block size, as shown by bandwidth-per-access-size performance experiments. Byte-wise operations are internally cached by one read and one write buffer those sizes approximate 16KB. Which means that smaller random reads or writes can be tolerated for very small working sets; however, as soon as those internal buffers are reaching saturation, any subsequent access will force a whole block operation, leading to apparent read or write amplification. Small accesses are then preferably avoided, to maximize Optane saturation bandwidth.

- Maximum bandwidth for a single module is around 6GBps for reads and 2GBps for writes (with optimal 256B ordered writes). When interleaving several Optane DIMMs, the maximum bandwidth compared to DRAM was between 2x-6x lower for reads, 6x-18x lower for writes, depending on access pattern and ratio of reads vs writes. Bandwidth is also higher (up to 4x) when accessed sequentially, showing that the device contains merging logic for adjacent requests. Last, parallelism is

also limited, read throughput declines above 8 threads with a single module and 16 threads for interleaved modules; write throughput starts declining with only 4 threads.

Reaching optimal performance with Optane PMEM is not straightforward. These modules can work as drop-in replacements for DRAM, but performance will likely suffer. A recent study [331] from Xiang et al. in-depth analyzed the internal on-DIMM buffering and showed asymmetry in the read vs write buffer management. Moreover, they established that the mismatch between cache line granularity and 3D-XPoint 256B blocks negatively impacted CPU prefetching and was responsible for wasted DIMM bandwidth.

We shall also add that, as the memory model of Intel platforms supporting Optane NVMM does not enforce strict ordering of writes, **to actually issue sequential writes one needs to explicitly order them.** Meaning that a sequence of adjacent memory writes made by a program may result in random write requests to the module when the program does not use write ordering instructions. In cause the cache eviction policy that enlists cache lines for write back as they move to the last-level cache (LLC), irrespective of program instruction order. In practice, we were measuring similar bandwidth with random write patterns or sequential writes with no or infrequent memory fence instructions. **According to general belief, memory fence instructions are expensive, but in the case of sequential writes, they are essential to achieve optimal bandwidth.** Even more so to avoid write amplification from torn blocks, that may appear when not explicitly ordering sequential byte-wise writes, as a result of the mismatch between CPU cache lines and Optane blocks.

Takeaways of these studies are highlighted by Yang et al. in [338] under best practices for NVMM. The first and most disappointing one is « Avoid random accesses smaller than 256B », then recommend to « use non-temporal stores for large transfers » and finally « limit the number of concurrent threads ». In the light of these facts, they suggest that prior art produced in anticipation of NVMM should be reevaluated and re-optimized for real Optane NVMM; as they significantly differ from random-access memory with emulated latencies that were used back then.

End of life As of 2022, Intel has shut down Optane device production and removed equipment from their fab. Micron had discontinued Optane as early as March 2021, Intel made their announcement one year after in July 2022. They still claimed third generation of Optane to hit the market, with several years worth of supplies in existing inventory. Intel took the harsh decision of winding down Optane business due to sales hardly picking up, in the middle of a bad year. It was largely relayed in press articles [39, 105, 226, 263, 294], and we all had troubles accepting it: “It was the biggest step forward since the minicomputer. But we blew it” [263].

Optane memory was definitively a product with a bright future, but also one that have struggled to find a market. Coming at around $5-6 per GB, Optane is 50% cheaper than server-class DDR4 ($11-12 per GB) but almost 10 times more expensive than server class NVMe SSDs (<$1 per GB). When Optane SSDs and NVMM are put head to head in a persistent index comparison [170], the cost per performance of SSDs is highly competitive although NVMM has better overall performance. Hence, the craze about extra IOPS and tenfold lower latencies for storage was cut short by the much higher price point compared to SSD, and caused a disengagement in both consumer and data center markets. In memory applications, Optane significant edge on cost-per-GB versus

	ReRAM	PCM	STT-MRAM	SRAM	DRAM	Flash (NAND)	HDD
Cell size (F^2)	<4	4-16	20-60	140	6-12	1-4	2/3
Energy per bit (pJ)	0.1-3	2-25	0.1-2.5	5.10^{-4}	0.05	2.10^{-5}	1-10.10^9
Read time (ns)	<10	10-50	10-35	0.1-0.3	10	10^5	5-8.10^6
Write time (ns)	≈ 10	50-500	10-90	0.1-0.3	10	10^5	5-8.10^6
Retention	years	years	years	voltage	<< 1s	years	years
Endurance (cycles)	10^{12}	10^9	10^{15}	10^{16}	10^{16}	10^4	10^4

Table 2.1: Memory technology trade-offs. [23]

DRAM was dulled by its unique set of performance properties, requiring Optane-specific tuning.

Intel's NVMM altogether was so much more than just some sort of really fast disk drive, or denser main memory. It was an enabling technology when it came to persistent memory and a huge step forward that challenged at least four decades of computer system design. PMEM support obviously could not build itself overnight, still it could have gradually matured as more killer app and use-cases were demonstrated. Marketing however blew Optane, by not immediately making it worthwhile in legacy systems. Whether a lower price tag would have secured Optane's future, or incurred even more loss to Intel, we will likely never know. NV-DIMMs are poised to reappear in the near future in the market, from different manufacturers and/or with alternative technologies. As noted in this ACM SIGARCH blog entry [298], PMEM devices are fully part of the CXL 2.0[1](*Compute-Express Link*) specifications backed by Micron; which heralds the (near) return of NV-DIMMs.

2.4.3 Alternative technologies

Truthfully, byte-addressable non-volatile memory did not start and will hopefully not stop with Intel's modules. A variety of emerging non-volatile memory technologies are being actively explored, as summarized in [335], as denser and lower-power alternative to DRAM. The main ones are *phase-change memory* (PCM) [324], *resistive* RAM (ReRAM) [33] and *spin-torque transfer magnetic* RAM (STT-MRAM) [43]. Table 2.1 illustrates properties of each, and shows that all are fit for byte-addressable NVM use. We can mention that Optane is a representative of PCM, according to data gathered from performance studies, even though Intel never disclosed 3D-XPoint underlying technology.

For reference purposes, we now briefly present NVM's past, and alternative future approach to support persistent memory that do not rely on byte-addressable NVMM devices.

Magnetic core memory We could trace back as far as the 50s and 60s to find magnetic ferrite-core memory that were also non-volatile. Later in the 70s, Bubble memory, another kind of core memory, was developed and pushed for storage applications, even incorporated in commercial products. However, by the early 80s, faster semiconductor memory (DRAM) and mechanical hard drives had completely buried them with their faster response time, higher densities and overall lower cost. Bubble memory subsisted for a while in a niche of systems operating in harsh environments (e.g with high vibra-

[1]CXL is an industry standard interconnect with low-latency (sub-Infiniband scale). It enables cache-coherent device sharing in the whole rack.

tions), where they were used to avoid the high failure rate of mechanical drives. An anecdotal but telltale example was Konami's Bubble arcade game system that used bubble memory game cartridges for storage, to cope with ruthless players. Eventually, these systems adopted either ROM chips (no need to save states in arcades) or flash memory as they became cheaper.

Simulation, emulation and Vikings NVMM has been long anticipated, and researchers did not wait idly for Optane NV-DIMMs to be readily available; by the time they were released, a slew of papers had already been written. Starting from 2010, they began proposing new hardware designs, programming models, file systems, libraries and even applications built to hardness to full potential of byte-addressable NVMM. While work introducing new hardware components resorted to simulation, system research drew conclusions using DRAM hardware in lieu of upcoming NV-DIMMs. At the time, expectations for NVM were DRAM-like behavior with higher latencies and narrower bandwidth. Performance studies were made in anticipation using PCIe-attached FPGA-based NVM-emulating devices [82]. Techniques for NVM emulation induced those lower performance through software emulation, new CPU microcode [350], exploiting NUMA remote-node access, and simply pretended DRAM to be persistent. A few work were also evaluated using NV-DIMM contraptions. For instance, using the battery-backed DDR4 modules from Viking Technology, as did Espresso [329] (§2.8.4.1), Anecdotally, [197] evaluated a new WAL technique for NVMM, by running SQLite benchmarks on a Nexus 5 smartphone, attached to an NVM emulation board.

Software persistent memory Momentum gathered from NVMM and Optane announcement, combined with a scarce availability, eventually led to rediscovery of persistent memory designs for conventional hardware.

SoftPM [153], for instance, proposed in the midst of the NVM hype, a software persistent memory abstraction for C. The abstraction is conceptually close to RVM we discussed, but with the aim of reducing programmer involvement. To this end, they mix in other ideas from older object persistent stores, such as write detection at page-fault time, or transitive persistence. The storage layer implements an atomic-commit protocol for resilience. The programming interface revolves around asynchronous persistence checkpoints, with calls to synchronize on outstanding checkpoint I/Os.

Kelly has re-demonstrated with an in-depth tutorial for C++ in [189] the use of `mmap` in modern operating systems to implement on conventional hardware (spinning or flash block devices) a persistent heap with the foundations of a persistent memory abstraction. One of which is the introduction of a failure-atomic msync (FAMS [254]) call to facilitate programming.

PM-gawk (persistent memory GNU AWK) [191, 303] and PMA (persistent memory allocation) [190], by Kelly et al., recently brought persistent memory to scripting languages without relying on NVMM. Scripts are simple programs; used for clear, concise and quick writing of automated tasks or recurring jobs. The proposed persistent scripting paradigm offers to "remember" script whole-state across executions. The PM-gawk script interpreter attaches to a file-backed persistent heap managed by PMA, that it uses to re-populate previous variable states, and save state at the end of a run. One good usage example are scripts that analyze continuous data streams (e.g., append-only files, log files). In this instance, recovering previous state allows for dead simple incremental computations, where only new data can be processed at each script invocation.

Flash memory SSDs and flash have long been relegated to the traditional block interface. They were commonly thought of as fast drop-in replacement for spinning mechanical drives and have thus far formidably emulated the behavior of sector-based drives. However, the NVMe ecosystem and specification nowadays grows larger with support for modern interfaces and I/O frameworks. Featuring new capabilities like user-space I/Os (with the SPDK [341]), zoned namespaces [67] and NVMe directives. New capabilities also keep landing, such as upcoming programmable interfaces for on-storage computations. New NVMe extensions are constantly being proposed, for instance ccNVMe [216] (*Crash-Consistent NVMe*), that brings transaction-aware memory-mapped I/O to efficiently provide atomicity. Additionally, the NVMe specification starting with version 1.4 [21] (2019) introduced support for an on-device PMR (*Persistent Memory Region*), directly byte-addressable as it bypasses NVMe command queues. All for an enhanced versatility that marks the dawn of a new era of system-storage co-designs. For reference, that body of knowledge is summarized in [211], a tutorial paper evocatively entitled « Not your Grandpa's SSD ». Alternatively, designs for byte-addressable flash were also proposed:

eNVy [328], in the middle of the 90s, described a non-volatile main memory storage system that presented a linear storage space to the application. eNVy sat on the memory bus, with a combination of flash memory for data persistence, and battery backed SRAM for internal buffering. Internally, the system used a copy-on-write approach to work around flash inability to perform in-place updates; and remapped the memory pages in the battery backed SRAM to make in-place updates transparent.

FlatFlash [29], in the last years, proposed to leverage the byte addressability of the PCIe interconnect to turn flash SSDs into cheap main memory. Even though the goal was to extend volatile memory more efficiently than with usual OS paging, they effectively demonstrated a byte-addressable SSD design. Through existing interfaces and using memory-mapped I/O, CPUs are capable of issuing byte-grain operations to SSDs, but the NAND flash chip on SSDs are not byte-addressable. The core idea was to leverage the internal DRAM in SSDs, previously used for the flash translation layer (FTL), to allow the CPU to address the SSD memory with cache line granularity.

Our closing note on byte-addressable NVM goes to Optane DC PMEM. Intel's memory is gone, and we hardly got to know her. The future however is not looking so bleak. Hopefully, persistent memory is here to stay, whether supported by another emerging technology, or a smart arrangement of software-storage co-design. Optane has put persistent memory back under the spotlight. We must now play our part and focus on attesting its applicability as an abstraction (§2.6 onward). Before though, we next go over potential use cases of NVMM in §2.5, to fully comprehend the enormous reach of the technology.

Please note that we will henceforth, unless explicitly specified otherwise, exclusively consider Intel's platform and Optane NV-DIMMs when addressing NVMM or persistent memory. Recent development on Intel and Micron winding down of Optane business were fun to document, as were referencing alternatives; yet we had assumed Intel's model and hardware throughout the entire duration of this PhD.

2.5 Applications & Use cases for NVMM

We previously concluded that, without non-volatile main memory, the abstraction of a persistent memory was never truly grounded (§2.2). Past developments were closer from research test vehicles than industrial products (§2.3). The promise of increased productivity from the erased distinction between in-memory and storage data representations did not win the argument. Overall, it is fairly understandable that such a paradigm switch was compromised after decades of asset investment in the now all successful relational databases and file abstraction. Programmers already knew well enough their way around the traditional interfaces for persistence to firmly believe in or advocate for their superiority. They could genuinely not feel the urge to adopt newer, shady-looking, ways to go about their valuable stored data, even more so for the lack of general support for persistent memory from trustworthy organizations. The apparent productivity gains must have resonated like a farce to anyone realizing the whole code base needed to be re-designed and re-written.

The availability of non-volatile main memory hopefully echoes a more engaging message (§2.4). This time, the increased productivity is supplemented by real-world performance gains. Above all, persistent memory feels like the natural and proper way to harness NVMM and harvest all of its benefits. NVMM-backed persistent memory is not another logical layer to abstract storage, but the most direct path for data to be persisted. Someday, it may even overthrow the antiquated and unsafe file systems that can only stream untyped bytes. However, this day is still far ahead: a colossal amount of work must be carried out before persistent memory could reliably and easily support systems. Until then, applications of persistent memory will span across support for legacy systems and data persistence interfaces, to whole new system designs opened by the technology.

We propose in this section to introduce use cases of persistent memory within this spectrum from obvious to novel, that is, building giant memory systems (§2.5.1), speeding-up file systems (§2.5.2), recovering large databases in seconds (§2.5.3) or assembling totally novel persistent components in systems (§2.5.4). Additionally, we attempt from this overview to get a better grasp of potential user expectations towards persistent memory programming abstractions.

2.5.1 Large memory systems

The easiest way to use NVMM is to ignore its persistence, to build giant main memory systems. Real world big data processing or HPC applications often require or perform better with extremely large amounts of main memory. Employing persistent memory *in lieu* of DRAM could potentially grow by 8x systems' maximum memory capacity while keeping the cost down and making them more energy-efficient. Even so NVMM can be twice as cheap, recall that it noticeably increases latencies by 2-3x and narrow memory bandwidth up to 8x. In [106], the authors went extreme and used only NVMM (no DRAM) to survey the performance impact of the new memory kind on OLAP workloads. They found that DRAM-like read accesses were harmless, but unrestricted write patterns to be damaging, and thus proposed guidelines to keep writes from overly hurting bandwidth.

Mixing NVMM and DRAM technologies is key to constitute an hybrid or heterogeneous main memory of higher capacity and reap the cost, performance, or energy benefits. A simple off-the-self, but limited solution is to use the "memory mode" of Intel's NV-

DIMMs. The firmware-builtin DRAM-NVMM tiering mechanism is fully transparent to OS and software. Results are extremely mixed depending on applications, and such hardware-level mechanism leaves little opportunity for smarter refinements. A body of work is dedicated to exploring sensible approaches for heterogeneous main memory tiering, with the goal of essentially hiding slower memory characteristics. These approaches either fall into transparent tiering *(i)* at the OS level, *(ii)* at the language runtime level, or explicit tiering through *(iii)* specialized memory allocators.

OS-level tiering. OS-level support for heterogeneous memory in Linux was introduced with a series of patches to expose NV-DIMMs as virtual NUMA nodes [101, 287]. The two memory tiers are simply managed with NUMA balancing. **Thermostat** [31] further implemented in Linux a page-sampling technique to identify hot and cold memory pages, and demote cold data to slower memory. **HeMem** [268] presented a low-overhead memory access sampling method based on CPU events, and implemented their memory management policies in a user-level library for flexibility.

Runtime-level tiering. Managed workloads constitute quite a hefty portion of big data processing, but garbage collection makes ineffective transparent physical memory page sampling techniques employed at the OS-level. In cause, mainstream garbage collectors that are copy-based, meaning they keep changing the data layout by relocating objects to different physical pages for memory compaction. Additionally, the mixing of small random read-write operations found in memory compaction algorithms was identified as outstandingly disrupting for NVMM bandwidth in [340]. Naturally, this led to specific solutions and language runtime enhancement for hybrid memory, as found in [34–36, 215, 311, 340]; which we save details for later discussion on Java garbage collection in §2.8.1.

Allocation-time placement. Hybrid memory allocation APIs forgo transparency and allow hinting the purpose of memory objects. **Memkind** [14], supplied by Intel, is a general-purpose memory allocator that allows choosing the destination kind of memory between DRAM, NVMM or HBM. **Unimem** [327] is a runtime system for HPC that leverage allocation-time hints and online profiling of memory access patterns to choose the best data placement of memory objects. **NVCache** [132] employs such hybrid allocation APIs to devise a volatile NVMM cache for MongoDB. At its core, NVCache throttles writes to balance with the rate of reads, so as to avoid NVMM bandwidth to crumble under unrestricted write traffic.

As a whole, the take-home message of volatile usage of NVMM is - besides guidelines for access patterns and limitations of managed runtime mechanisms atop NVMM - that exogenous policies provide a good starting point with broad support, but do not eliminate the need for explicit application-level NVMM management: necessary to treat peculiar policies close to application semantics. Then, NVMM programming libraries could also come as a solution to explicitly manage transient NVMM objects, by allowing crash-consistency to be disabled.

2.5.2 Block storage

The second easiest way to use NVMM ignores its memory characteristics, and exposes a block device on which OSes allow conventional file systems to run. File systems bring

NVMM persistence to legacy applications "for free" through conventional file I/O. Although forfeiting unique opportunities for single data representation and copy along the way. In a nutshell, legacy I/O intensive applications can outright perceive a performance boost without any code modification.

Unfortunately, conventional file system have known decades of careful optimization for sector-based disk drives that can add microseconds to common operations. Moreover, file system operations are typically handled in kernel space, which incurs system call overheads at every I/O and additional memory copies of the data. This extra latency may be irrelevant to spinning drives with millisecond-scale response times, or even SSDs with hundred-of-microsecond latencies; but definitely noticeable with low-latency media such as tens-of-microsecond SSDs or our microsecond-scale NVMM.

A line of work then focuses on designing NVMM-specific file systems that thrive at reducing system overheads of file systems, and that adopt NVMM-friendly access patterns to further enhance common operations latencies.

Optimized file systems. **NOVA** [332] is one of the most notable efforts in this area. It adapts conventional log-structured file system techniques [273] and data structures to be a better for fit NVMM. Additionally, it provides atomicity to metadata, data, and mmap updates, while remaining between 10% and 10x faster than file systems with equally strong data consistency guarantees. **NOVA-Fortis** [333] further address reliability issues in the face of media errors and software bugs. Achieving resilience versus data corruption through checksums and raid-style parity. The authors measured a limited impact on performance versus plain NOVA: it still outperforms reliable file systems running on NVMM by 3x on average. **FLEX** [334] (*FiLe Emulation with DAX*) is not a file system, but a transform that rewrites file I/Os with user-space memory operations. It was used to gauge the impact of file system software stacks on NVMM performance. For short, it turns `open()` into a DAX `mmap()`, replaces `write()` with non-temporal stores, `read()` with `memcpy()` and `fsync()` with `SFENCE`. Note that FLEX is not atomic, thus breaks applications that assume atomicity of the write system call. **SplitFS** [184] distributes the responsibilities of handling metadata and data updates between the kernel and a user-level library. It intercepts POSIX calls to service data operations with direct-access load & stores by memory-mapping the underlying file. In doing so, it bypasses costly system calls and in-kernel operations. Overall reducing the software overhead by up-to 4x compared to NOVA, while providing the same consistency guarantees. **WineFS** [185] notices that fragmentation severely degrades performance of NVMM file system. It then proposes to use log-based consistency techniques to avoid disruption of hugepage usage, at the cost of an increased write traffic. Prior log-structured file systems were unable to allocate 2MB hugepages after being highly utilized, having to resort to 4KB pages instead and causing extra page faults and misses. WineFS matches NOVA performance in the basic settings, and performs twice as fast as an "aged" NOVA file system.

Caching file I/Os. NVMM has also long been seen as a suitable media to build fast persistent client-local I/O caches for distributed file systems. As early as 1992, Baker et al. [53] showed through trace-driven simulations, that small amounts (1MB) of NVMM on disk-less clients could halve remote writes to server disks. Similarly, they demonstrated reduced disk traffic by using small NVMM server-side write buffers. **ZFS** [317], an industrial grade file system, have long had the capability of employing low-latency medium to log writes and reduce latency of the file systems. **Strata** [203] is a file system

built for a 3-layer storage hierarchy, made of emulated NVMM, fast flash SSDs, and high-density HDDs. Data updates are logged in user-land on NVMM for fast response time, then asynchronously propagated to the kernel portion of Strata. This arrangement achieves good performance with a synchronous I/O model without being restricted to smaller capacity of NVMM modules. **Ziggurat** [351] expands on this idea to build a tiered file system with smarter I/O predictors and data migration policies to optimize for every tiers' characteristics. It profiles application use of `fsync` to predict I/O synchronicity, such that small synchronous writes are allocated on NVMM, while larger asynchronous ones are directly sent to slower tiers. Plus, only cold data are migrated out of NVMM, and hot data can be brought back to NVMM or DRAM under frequent accesses. Migration policies also coalesce data into variable size chunks, to optimize for sequential access at each device preferred granularity. **NVCache** [128] proposes a pluggable user-land NVMM file I/O cache that rival NVMM-specific file systems performance, with conventional file systems and tiered storage hierarchy. **Assise** [41] is a distributed file system that leverage client-local NVMM to build a linearizable and crash-consistent replicated coherent cache between compute and storage nodes. Features include near-instantaneous application fail-over to a warm replica that mirrored the file system cache, and faster remote reads by accessing NVMM caches of warm replica instead of network disks. **Hydra** [352] and **Orion** [337] utilize RDMA networks to further accelerate such decentralized distributed NVMM-aware file system designs.

Direct-access file systems. This last category of NVMM-specific file systems is the most compelling for us. They provide direct access (DAX) through `mmap`, which allows applications to access the content of a file directly in user space using load and store instructions. **PMFS** [127] from Intel, first proposed that DAX `mmap` by-passes the block layer and OS page cache to improve performance. These changes were then all integrated in conventional file systems with DAX support, such as the Linux file system **Ext4** [2]. They all maintain POSIX-like guarantees, that is metadata integrity and atomic metadata operations through common journaling techniques. However, they perform data updates in-place, and with Ext4-DAX, the data-journal of regular (non-DAX) Ext4 is disabled; in all data updates or appends are not atomic. As a consequence, it behoves applications developers to take care of data integrity. These file systems also do not provide the necessary features to detect and correct media errors and protect data against corruption.

To summarize, NVMM-specific file systems aim at supporting legacy programs and quicken their file I/Os with no code modifications. They are however limiting the potential of NVMM with the antiquated file interface that incurs additional data copy in main memory. Direct access from the CPU to NVMM modules is possible by memory-mapping a file on a DAX-enabled file systems. With that, a whole new support for in-memory durable data types, reminiscent of a persistent memory abstraction, that were beyond the realm of possibility with conventional file systems interfaces. However, as DAX by-passes completely the file system control plane operations, it falls onto applications to reliably write data onto the media and preserve their integrity.

2.5.3 Database systems

Database systems are among the most popular and anticipated field of application for NVMM. Its formidable characteristics are especially relevant for big data analytics, in-

memory databases, or high availability systems. In-memory persistent data types bring back the promise of productivity benefits from past persistent memory, together with unprecedented data access latencies. In that, they are extremely well indicated base building blocks for database systems; as attested by the recent burst of research activity in the area. Countless research key-value stores for NVMM spurted over just the span of the last 5 years or so; all in line for the immense expected benefits. Whether to lower database response times, increase operation throughput, attain near instantaneous system recovery; the list is long.

Benefits. NVMM enables groundbreaking evolution in DBMS architecture, calling for substantial code base simplifications. The fact that the technology might be suitable for data indexing, storage and caching all at once, may basically let DBMS maintain a single copy of any data object. Incidently deprecating the protocols that were responsible for maintaining consistency of data across those multiple representations and copies. Moreover, low media latencies might cut the need for asynchronous persistence, by making less significant the cost of strong consistency in the critical path of user operations.

Legacy support. With the release of Optane modules, performance studies empirically analyzed opportunities for NVMM in databases. In [202], the authors surveyed off-the-shelf (memory & block mode) Optane configurations for traditional relational DBMS (PostgreSQL, MySQL, SQLServer). Despite fine tinkering with many of the engines' knobs, they reported little improvement with NVMM versus SSDs for storage media. Quite expectedly, their findings corroborate that storage performance in DBMS is held back by software inefficiencies rather than storage hardware characteristics. Especially for write-heavy workloads, where an unoptimized I/O path dip overall performance on NVMM. Precisely, DBMS storage engines spend significant CPU resources to reduce I/O traffic, which turns out to be baseless and detrimental with NVMM media. In all, DBMS re-design is not only advisable but necessary to make NVMM hardware worthwhile.

DBMS re-design. Strategies for NVMM integration can be broken down into 3 distinct stages, each of higher complexity but better payoff.

- *Stage 1* optimizes the block storage layer for NVMM devices. This is the least intrusive strategy and is typically enough to witness a throughput increase. Latency critical I/O operations, such as logging (e.g., WAL), are placed on NVMM.
- *Stage 2* takes a more radical approach. It turns volatile memory tables into NVMM-resident persistent tables. By making memory tables persistent, the memory footprint is substantially reduced (no more volatile DRAM caches). User data need no longer to be marshalled and hosted on block devices, making the DBMS code for block storage redundant and dispensable. However, the complexity of keeping memory tables crash-consistent may hinder throughput versus *Stage 1*, as it becomes harder to maximize sequential NVMM bandwidth.
- *Stage 3* directly implements persistent index structures. That is, this stage leverages NVMM-native persistent data structures for data indexing. Further cutting down on the database startup time, as in-memory indexes need not to be re-populated. We will tackle this point in §2.7.1 together with other data structures for NVMM.

Several industrial-grade modern databases have benefited from NVMM re-design efforts. Experimental code base forks are available, and leverage Intel's PMDK to im-

plement NVMM support strategies. Example includes NVMM ports of MongoDB [18], RocksDB [7], SAP Hana [42], Redis [6], Memcached [125] or Apache Cassandra [17].

Network stack overheads Fedorova relates on the **MongoDB** developer's blog [131] their experience with Optane PMEM and Optane SSDs on WiredTiger, MongoDB's default block storage engine. Their experimental findings are on a par with the studies performed on other DBMS: WiredTiger exhibits no superior throughput with Optane PMEM in block mode versus Optane SSD. They concluded that DBMS storage engines were effectively hiding storage media latencies with DRAM caching.

Interestingly though, Izraelevitz et al. in [175] also benchmarked Intel's Persistent Memory Storage Engine (PMSE) for MongoDB [18]. **PMSE** is a drop-in replacement for WiredTiger, that rely on Intel's PMDK to implement the engine's key-value interface with a crash-consistent hash map. They found no throughput difference as well between PMSE and WiredTiger on Optane PMEM modules; which they associated to the cost of client-server communications and MongoDB query processing engine in the benchmarking application. Note that client and server were co-localized, meaning the dominating cost lies in the networking software stack and not the actual network data transfers.

The conclusion we can draw here with MongoDB and the *stage-1-ish* re-design strategy of PMSE, is that in a nutshell, Optane PMEM failed to demonstrate in a client-server setting a throughput increase versus SSDs. As Fedorova noted, Optane PMEM may better serve operations where access latency to storage is the bottleneck, such as write logging. We shall add - when not undermined by the network stack latencies - considering that Optane SSDs and their hundreds-of-microsecond delays are otherwise hardly restricting on conventional ethernet stacks.

Embedded databases. As seen previously, the purpose of NVMM is unclear in client-server databases: whether performance gains may still be attainable with faster networking stacks (e.g., Infiniband), or whether exclusively recovery times may be improved with Optane PMEM. Fortunately, embeddable databases can help us complete the picture. As living in the application's process, they negate the cost of networking. **RocksDB** is such a database, and benefits from an NVMM-optimized WAL scheme [7] (*stage 1*). In [175], the authors report an appreciable throughput increase versus Optane SSDs: almost 8x when going from an Ext4 file system with Optane SSDs to a NOVA file system with Optane NVMM. Additionally, the authors performed a *stage-2* re-design by making RocksDB's *memtable* (skiplist) persistent using PMDK transactions, and removing the block layer (WAL & file storage) that had become unnecessary. In doing so, they registered an additional 73% throughput increase over the previous best configuration. At last, this depicts a compelling illustration of the potential performance benefits that NVMM may bring to database systems. Inconveniently, these may not benefit other kinds of databases until networking stacks catch up.

In-memory databases. Replicated and distributed object caches are high-performance databases built for quick response times in highly scalable applications. Their distributed and highly-available nature made them resilient to node failures without resorting to persistence. However, being entirely volatile made recovering local node crashes a rather costly operation: receiving lost data over the network from replicas is both time and energy inefficient. As such, in-memory databases usually implement a rudimentary data

persistence scheme (e.g., an append-only-file), that asynchronously persists data to alleviate the node recovery cost.

RAMCloud [248], a research decentralized in-memory storage system, made the case for fast crash recovery by recollecting data in parallel from multiple nodes. It could recover 22GB of data per second in a 60-node cluster, reaching the upper limit of the Infiniband network device bandwidth on the recovering server; equivalent to 8-10 aggregated Optane SSDs. In contrast, it would take only 3 interleaved Optane NVMM modules to rival that rate of recovery while relying only on local storage.

Going further with fine NVMM integration (*stage 2*-ish strategies), this class of database could become crash-consistent and experience near-instantaneous node recovery. Assuming that data and the primary store reside on NVMM, no data movement would be required on recovery, only the re-generation of secondary indexes. The point for NVMM impact on in-memory databases is made and summarized with more depth in an IDC white paper [247], wherein SAP Hana is taken as example of joint efforts between SAP and Intel to showcase NVMM direct access potential.

SAP Hana was the first commercial product to benefit from Optane PMEM support, as described in [42]. Hana is a modern in-memory database built from the ground up with large scale data analytics in mind. In all its complexity, the authors in [42] emphasize on « an early adoption, where HANA consumes NVRAM without heart surgery on the core relational engine. » As such, NVMM in Hana supports the primary columnar store and hosts only the backing arrays of column dictionaries or data vectors. Being published prior to Optane PMEM release, the performance figures in the paper are tainted by the lack of real NVMM hardware. Nonetheless, marketing claims assure up to 12x reduced startup times (from VMWare's blog [269]) - loading terrabytes in minutes rather than hours - thereby incurring an overhead contained under 7% on most workloads, and up to 30% for the worst one (data insertion), according to Fujitsu's white paper [24].

Memcached or **Redis** are popular distributed in-memory object caching systems. While Memcached exposes a simple key-value store interface, Redis, also referred to as "data structure server", offers an API with richer commands. They both make use of NVMM, as found in [125] and [6]. However, under the complexity of programming persistent memory, both approach NVMM as volatile and, as a mean of reaching higher memory capacities per CPU.

Overall, non-volatile main memory is truly a re-defining media for data store systems. All anticipated benefits could be validated from preliminary studies and real Optane PMEM; but none of them may be harvested without surgical evolution in DBMS architecture. Sadly, performance improvements will go unnoticed except with faster networking stacks, or unless the client-server paradigm is avoided - as in the case of embeddable databases. To top it all, these game changing evolution are held back by the complexity of correctly programming with persistent memory in DAX mode. These major setbacks are hardly balanced out by shorter recovery times alone: given how huge amount of time, asset and material expenses they require to implement. To this day, no major player has conducted an official and thorough NVMM integration campaign - Hana is perhaps as close as it gets - when others like Redis or Memcached avoid persistence altogether. Hopefully, reticence will cease as the NVMM ecosystem matures, with simpler programming libraries, paired with tools to assert and validate integrity of persisted data. In order to evaluate our contributions (§5.1), we augmented the Infinispan NoSQL industrial store [229] for NVMM using our system. Thanks to our design decisions, we

were able to avoid dual representation of data and marshaling when porting Infinispan to NVMM, in less than 200 lines of code.

2.5.4 Exotic applications

Non-volatile main memory truly is an earthshaking piece of hardware with never-before-seen properties and characteristics. As such, applications extend beyond the enhancement of traditional storage interfaces, including file systems or databases. NVMM together with the versatility of a persistent memory programming abstraction pave the way towards novel and original means of going about durable data. To the point where it redefines modern computing system architectures and proposes new execution models. So as to illustrate this, we are about to take a peek at topics that emerged very recently around *(i)* persistence of user data in network devices, *(ii)* using GPU as storage workload accelerators, or even *(iii)* rethinking data centers around unreliable power sources.

In-network persistence We previously made the point for NVMM being held back by traditional networking stacks lagging behind in respect to both throughput and latency. On one hand, a reasonable line of work attempts at soothing the issue by mixing-in fast disaggregated memory abstractions [314], or by combining PMEM with RDMA [224, 313], all to build low-latency in-memory distributed databases. On the other hand, a more disruptive take on the matter proposes that emerging programmable network devices directly address and manage NVMM.

PMNet [281] for instance builds on the idea of caching read requests on programmable network data plane (e.g., NIC, switch), as in **NetCache** [180], and extends it to also log update requests *in-network* by attaching NVMM on those devices. In doing so, clients no longer have to stall, waiting for servers to commit their updates and acknowledge them. Instead, the network critical path is shortened as PMNet allows to reliably stash data by logging update requests on programmable network devices, potentially closer to the client. In all, PMNet turns emerging programmable network devices into read/write request proxies for applications, extending the data-persistence domain from servers to network devices; with the goal of alleviating impact of network transfers.

PASTE [164] complementarily proposes that user-level I/O stacks, meant for lower latencies as kernel by-passing, be built jointly for storage and network purposes at the NVMM era. The result is a network programming interface that supports standard transport protocols (e.g., TCP, UDP) and DMA data straight from NIC to host NVMM; such that they "never need to be copied again - even for persistence". Effectively negating the software overhead of traditional networking stacks.

GPU-accelerated storage Functionality and utility of GPU as a "general-purpose" computing platform is an ever expanding province. In a nutshell, any program with data-intensive computations, that is either - *(i)* very demanding on memory bandwidth, and/or that *(ii)* executes undemanding tasks, fit for slim computing units, but with massive data parallelism - might be sped-up by running on GPU. Two properties that in-memory databases verify, according to **Mega-KV** [348] that made a strong point for high-throughput GPU-based volatile key-value stores.

In addition, data path to persistent media from GPUs - or the lack thereof - are shackles to the diversification of computing kernels and the generalization of GPU as a computing platform. **GPUfs** [291] proposed file system primitives for GPU kernels.

Although it helps eliminate CPU support code required to feed data to GPUs, data ingestion is still assisted by the CPU under the hood - they transit through the main memory hierarchy before reaching the GPU. More recently, in 2019, NVIDIA introduced **GPUDirect Storage** [305] in CUDA. The technology exploits DMA capabilities of PCIe interconnects to bypass the CPU and access NVMe SSD data directly; however relies on a proprietary IO subsystem and programming interface.

GPM [253], very recently in 2022, envisioned making NVMM directly accessible to GPU in order to achieve fine-grain persistence in compute kernels. With conventional hardware and by leveraging NVIDIA's Uniform Virtual Address (UVA) technology, NVMM memory ranges can be mapped to kernels' virtual address space. From there, GPM proposes a fully-fledged persistent memory programming abstraction for GPUs, including logging and checkpointing facilities optimized for GPU parallelism. Creating in this way a system where GPU kernels may directly manipulate NVMM-resident data structures crash-consistently, without help from the CPU or the OS. The reported results present impressive speedups across-the-board for a host of popular GPU applications. In particular, for in-memory key-value stores, the authors report throughput increases by: • 8x in Mega-KV when using GPM instead of GPUfs to persist updates on NVMM, • 4x in Mega-KV when persisting updates with GPM instead of delegating persistence and NVMM data flushes to the CPU, and finally, • 3x for Mega-KV+GPM compared to RocksDB-PMEM running on CPU.

Intermittent computing Long before byte-addressable NVM were sensibly available to mainstream computing platforms with the recent introduction of Intel's Optane lineup, NVRAM chips have been deployed on small devices and embedded computing platforms such as IoT devices, wearables, sensors and environmental monitors.

Energy harvesting systems [262] are such small devices that run off unreliable power sources. NVRAM chips enable them to go battery-free, reducing their environmental footprint, while achieving ultra-long operation times without maintenance. As they collect energy from different ambient sources (e.g., solar, thermal, radiations) and face frequent unpredictable power outages; these platforms require programmers to reason about energy to compose data consistency with forward progress in their programs.

Intermittent computing is the field that strives at solving these challenges: avoid data loss or inconsistencies, while balancing performance, energy and result quality. In particular, through lightweight checkpointing that saves hard-earned energy by reducing overheads of saving/restoring states. Techniques vary, for instance, **QuickRecall** [177] performs *just-in-time* (JIT) checkpoints including whole register states when voltage is about to drop. **Ratchet** [325] does not rely on voltage monitoring and energy buffers, but rather adds at compile-time lightweight checkpoints between idempotent regions of code. **Chinchilla** [225] also instruments code at compile-time with checkpoints, but disables them dynamically based on an adaptive timer. **ReplayCache** [345] enables the use of volatile write-back caches without the associated data crash-consistency issues through software-only techniques. Instead of resorting to costly logging of programs' memory stores, they combine JIT store register checkpoints, as in QuickRecall, with compile-time persist code region detection. By doing so, they allow in-place NVM stores to be asynchronous throughout persist regions, with no write amplification; since stores that were not persisted before a power failure may be replayed first thing on restart from the register checkpoint.

The advent of NVMM on server platforms creates interesting openings for an in-

termittent computing model on traditional computer architectures. Whether to power energy-aware computations, shut down under-loaded servers, or even have whole data centers run off self-produced energy - as scarcer unreliable power sources. This might just as well be one of the first step towards energy-driven large-scale computing; where services and jobs could be scheduled not only relative to customer traffic, but also according to energy rations or shares.

2.5.5 Summary

In summary, the application range of NVMM is truly wide. From enabling giant memory systems (§2.5.1), enhancing performance of traditional storage - file systems (§2.5.2) or databases (§2.5.3); to novel scenarios where low-latency fine-grained data persistence might empower programs with data durability or fault-tolerance almost for free (§2.5.4). Of course, these boons may not easily be unlocked without proper programming support for NVMM. In all instances we presented, the full potential of NVMM could only be noticed with in-depth system re-engineering. A task that might be largely eased by integrating NVMM and persistence with programming idioms. Unsurprisingly that does not come for a cheap, as we are about to understand in the next section while we discuss hardships to overcome when ensuring reliable persistence with NVMM.

2.6 Challenges for persistence with NVMM

Considering what we have covered so far regarding both persistent memory and byte-addressable non-volatile memory, the only remaining grey area in the design of a successful PMEM abstraction over NVMM, is the case of answering requirements for data consistency in respect to potential faults.

The specificity of NVMM when compared to traditional storage devices lies with the fact that NVMM resident data is already persistent. No explicit data copy are needed to/from the storage device for persistence: durable bits are mutated in-place through direct byte-addressability. Problem arise with the physical arrangement of NVMM devices on the memory bus, requiring byte-wise operations to go through CPU caches. This means that the actual data updates are still buffered on volatile caches before navigating back upward to the NV-DIMMs. Now remember that CPU cache protocols were designed at a time were volatile DRAM were the only main memory option available. Obviously, with no account for persistence or data crash resilience. The challenges we are about to present regarding implementation of persistent memory on Intel platforms directly stem from there: the memory model enforced by cache protocols.

Please note that we intentionally restrict the discussion to Intel platforms. Not that the broader body of work on memory persistency models is unimportant, but is simply beyond the scope of our problem instance, which is again, persistent memory abstraction for Optane NV-DIMMs.

(Memory store re-ordering.) A first challenge is to have the data remain consistent upon reaching the *persistence domain*. That is, the region of a computer system where content will be preserved in the event of a failure, which may not be restricted to NVMM modules. Of course, dirty CPU cache lines can be evicted explicitly with a flush instruction, allowing data to reach the persistence domain. However, in this access hierarchy, cache lines may also be written back implicitly to main memory as part of the CPU cache management protocols built in hardware. These so called "implicit flushes"

also propagate the effect of memory store instructions to the persistence domain, but following an arbitrary order dictated by hardware - typically *not* the program instruction order. In other words, unless something specific is done, any memory store may reach the persistence domain in any order [2]. Although data *stability* can be trivially reached with explicit flushes, in presence of implicit flushes, guaranteeing data consistency in the wake of a crash is complicated and requires extra care.

(The read of not-yet-persistent-writes.) The second challenge comes as extension of the first one in concurrent settings. When multiple execution threads share a piece of data, consistency is provided by the cache coherence protocol. Precisely, besides locks, concurrent programs may synchronize on the notion of "globally visible write". Simply put, this notion denotes writes that were made visible by the cache coherence protocol to concurrent observers. The challenge comes from the mismatch between this concurrent visibility and the persist order. Not only that a write can be made globally visible and be observed by a concurrent thread without yet being persisted, i.e., not observable post-crash, but also that the cache protocols may further persist changes in another order, leading to potential inconsistencies given that a crash may occur at any time (in-between any two atomic steps).

To picture that, let's say that a write was made globally visible and allowed a concurrent observer from taking actions based on value he read, with for instance, writing at another memory location. In that scenario, the cache coherence protocol granted causal consistency: the original write (the cause) was globally visible before the subsequent write (the action). However, the cache eviction order could (and for the sake of correctness, we must assume that it always would) have the second write (the action) reach the persistence domain before the first one (the cause). Should a system crash happen in-between (and for the sake of correctness, we must assume that it always will), an inconsistent state would be recovered.

(Failure atomicity.) The third and last challenge is a familiar one, as it is also found in early persistent memory we previously discussed. Namely, in program and applications, typical data structure operations generally transition between multiple intermediate states in order to evolve the structure from one consistent state to another. Traditional persistence interface (files, database) allowed for explicitly stabilizing consistent states only. Any semantically inconsistent state was kept in volatile memory, and swept away with failures or reboots. Recall that we have however seen with persistent memory (especially with fully transparent persistence), that intermediate states might also be captured on non-volatile storage. Breaking transparency was then necessary, at least with an explicit checkpointing call, to identify semantically consistent states in the application. This is even more true for modern persistent memory, where nothing can prevent memory stores from reaching NVMM and becoming persistent. Bulk of the work required to properly answer this challenge in the NVMM era, is in the adaptation of prior techniques found in file systems or databases. Early failure-atomicity schemes for PMEM were optimized for mechanical drives; modern PMEM must re-explore this body of work under the new performance profile of NVMM.

Persistent memory from the past faced diverse difficulty: pointer representation, heap management, atomic checkpoints... The point is not that present persistent memory

[2] This statement is a bit of a stretch, especially since we consider Intel architectures which implement TSO (total-store order) memory model, where any two writes to a single memory location can not be re-ordered.

will not face them as well, but first and foremost, a gap must be filled in the platform's classical memory model to construct persistence with load/store instructions. Until then, the three aforementioned challenges can not be addressed, and no data can be reliably stabilized on NVMM.

This section then organizes as follows. (§2.6.1) We detail memory model extensions for persistency. (§2.6.2) We present the low-level (architecture-level) programming model extensions for NVMM, and proposed correctness properties to characterize faulty program executions. (§2.6.3) We examine attempts at reducing the gap with hardware modifications, and ultimately why software will always have to be involved for consistency. (§2.6.4) We discuss the implication of the preceding points towards protocols for failure-atomic updates, and why they must be re-adapted to NVMM.

2.6.1 Memory models

The previous informal description served as an introduction to the challenges presented by programming persistent memory on Intel machines. Hopefully, the reader could sense that the mismatch between the memory consistency model and the memory persistency model was at the core of the issue. Solving the issue requires defining correctness properties of persistent executions. Besides, the notion of a persistent execution implies defining persistency extensions to the memory model of processor architectures. Put simply, defining « the semantics of instructions controlling the ordering and timing under which cached values are pushed to persistent memory. » [174]

2.6.1.1 Persistency models

Pelley et al. [256] in 2014 importantly identified that a concept analogous to *memory consistency* was necessary to minimally describe write constraints with respect to failures. They dubbed the concept *memory persistency*, of which they defined two main classes: *strict* and *relaxed*.

- *Strict persistency* couples the persist order to the memory consistency model, e.g., SC (sequential consistency) or TSO (total-store order). Meaning the persist order matches the order in which stores become visible. The consistency write barrier thus also enforce persist order, simplifying the reasoning task of the programmer. *Strict persistency* is yet impractical: any volatile write with an ordering constraint on a persistent store will be stalled until the slower NVMM write completes.

- *Buffered strict persistency* is then introduced as an optimization where the volatile and persist order are still coupled, but the persistent state is allowed to "lag" behind. An implementing hardware typically achieves this through the use of a persist write queue. Persist writes are enqueued synchronously on persist barriers to record the order, while the queue can be processed asynchronously. In case of recovery, the persist state will match some prior point of the observable memory order.

- *Relaxed persistency* allows for further performance optimizations by fully decoupling the two models. Which comes at the cost of extra programming complexity, with a separate set of write barrier (`pfence`) used only to define persist order constraints. The advised reader might have noticed by now, from difficulty we mentioned previously, that Intel has chosen a *relaxed persistency* implementation for its platform.

- *Epoch persistency* is a notable model found in contemporary work (BPFS [98]) that Pelley instantiates within their framework. *Epoch persistency* is a relaxed persistency model that exhibits an interesting intuitive reasoning mechanism. Under this model, persist barriers (`pfence`) delimit "persist epochs", meaning, on recovery any write observed after a barrier implies all the writes before the barrier. Writes within the same epoch (in-between the same two barrier) are free to reorder or occur in parallel, improving persist concurrency. The *buffered epoch persistency* variant similarly allows the next epoch to start without waiting on previous persists to reach the non-volatile device. When necessary, a separate instruction (`psync`) can be used to block until the persist buffer has fully drained, emulating the behavior of the *un-buffered* variant.

2.6.1.2 Explicit buffered-epoch persistency

Epoch persistency may sound like a viable model that allows for intuitive reasoning and reasonable amount of persist concurrency. It is in fact part of the persistency model assumed by Intel platforms, give or take two subtle changes: Intel ISA extensions make for an *explicit* persistency model and forbid *persist-epoch races*.

Explicit models. Izraelevitz et al. in [174] duly notes that Pelley's persist models are *implicit*. Meaning persist barriers order persistent stores, without persistent stores appearing in the instruction stream. However this is not practical to implement in real hardware. A more grounded approach is to use *explicit* persist models, where persistent stores are differentiated from volatile stores. This is what manufacturers did, in both Intel and ARM, by introducing a *persistent write-back* instructions (`pwb`). Under these models, `pwb` is used to queue up persist stores relative to a given memory location; while `pfence` provides ordering constraints across memory locations.

Persist-epoch races. Pelley additionally noted that « reasoning about persist order across threads can be challenging ». In cause, the decoupling between consistency and persistency makes persist epochs local to each thread, and do not restrict cross-thread *persist-epoch races*. Racing epochs are any two persist epochs from different threads that share data. While allowing for persist epoch races may improve concurrent persists by lessening the ordering constraints, they are difficult to reason about: persists can not be ordered across racing epochs other than with individual atomic-ordering instructions. A strategy to avoid them, is to use thread synchronization primitives on shared resources to coordinate new persist epoch start across threads. If racing persist epochs are to be avoided altogether, at the cost of less concurrent persists, a `pfence` persist barrier might as well just imply a consistency barrier upon starting a new persist epoch. That reasoning gets us closer to the practical limits of the decoupling in Pelley's *relaxed persistency* model. Real solutions rather coordinate consistency and persistency models, for instance, to avoid persist epoch races.

Persistent TSO. Raad et al. in [266] followed that approach and intertwined Pelley's *buffered epoch persistency* with *Intel-x86* consistency model (TSO[3]). Their effort of formalizing persistency semantics under realistic consistency model resulted in *P-x86*, a

[3] A preliminary work called *P-TSO* [264] did exactly the same, but ended up mismatching actual NVMM support due to some inaccurate forethoughts.

combined consistency/persistency model. *P-x86* is effectively a formalization of actual Intel's memory model persistency semantics. This rigorous work showed for instance, that `psync` was unnecessary for correctness - and we will get to defining correct persistent executions - when the buffering persist write queue was part of the persistency domain. Since Intel deprecated their `psync` equivalent instruction, they have claimed their persist domain to extend to the WPQs (*write-pending queues*) in the iMC (*integrated memory controller*). Both of which are on-die CPU components (so not durable), but Intel supports this claim by stating that in power-fail events, latent energy is sufficient to drain the WPQs. They coined this technology ADR, for *Asynchronous DRAM Refresh*.

We clearly now understand the value of rigorous formal approach. It makes for a deeper understanding of the construction steps of persistency extensions and their *raison d'etre* in the memory model. Additionally, a formal definition provides a solid framework for verification of either programs correctness or model implementations in hardware. In all, essential to programmers, for them to reason about correctness when programming on persistent memory and to detect persistency bugs - whether manually or automated with tools. Last, an abstracted persistency model allows for portability across multiple computer architectures, which is important to understand as NVMM and software may not be restricted to Intel servers in the future.

Which finally brings us to detailing the full picture of the programming model for persistent memory, comprised of: Intel's ISA extensions for persistent memory, the persistency model it abides to, and the correctness properties programs must verify.

2.6.2 Programming model

Intel devised ISA extensions for persistence on x86, to permit NV-DIMMs to be used as persistent memory and circumvent the memory write-back problem (challenge 1). These extensions allow threads to control the ordering and timing of persistence enforced by the cache coherency protocols in hardware. The complete view of Optane PMEM programming model is available in Intel's book on persistent programming by Scargall and Rudoff [274, 277].

2.6.2.1 ISA extensions

The persistency domain on Intel microarchitectures with PMEM support, as we discussed, extends past the NVMM devices. On platforms benefiting from the ADR mechanism (Asynchronous DRAM Refresh), a write operation is considered to be persistent as soon as it reaches the WPQs in the iMC. That is because on power-fail, ADR ensures the WPQs to be drained with remaining latent energy. ADR protects the on-die integrated memory controller (iMC) of the CPU, but not the content of the CPU caches. Special instructions are then necessary to control the timing and ordering with which persistent writes to reach the WPQs. Notice that thanks to ADR, most of these special instructions can be asynchronous and at most require an acknowledgment from the iMC.

New instructions. The memory model of Intel machines supporting Optane PMEM is an implementation of the *explicit buffered-epoch persistency* model we just detailed. We remind that under this model:

- `pwb` is a persist write-back instruction that initiates write-back of a memory location without blocking,

- `pfence` enforces an happens-before ordering relationship between any previous and subsequent `pwb` instruction in the current thread,
- `psync` blocks until all previous persistency epochs delimited by prior `pfence` instructions have reached the persistency domain.

The correspondence between that abstract memory-consistency-persistency model and Intel's implementation is summarized in Table 2.2.

Explicit buffered-epoch persistency	NVM on x86	
`pwb`	`CLFLUSH`	`CLFLUSHOPT` or `CLWB`
`pfence`	`NOP`	`SFENCE`
`psync`	`NOP`	`SFENCE`*

Table 2.2: Intel equivalent instructions for *explicit buffered-epoch persistency*.
*asynchronous instruction because WPQs are assumed part of the persistency domain.

These new instructions leave programs with a *relaxed* and *low-level* approach to persistence. In detail, Intel propose 3 distinct persistent flush instructions: `CLFLUSH`, `CLFLUSHOPT`, and the brand new `CLWB`. The persist barrier is implemented by re-purposing, or rather extending the semantics of, `SFENCE` and other store-fencing instructions. The `PCOMMIT` instruction, originally planned to implement `psync`, was deprecated from earlier Intel specifications even before release. As it turned out with ADR, a blocking instruction waiting for the WPQs to be drained was unneeded. We now detail the effect of these instructions:

- `CLFLUSH`: *Flush Cache Line* invalidates the cache line containing the memory location pointed by the source operand and waits for persistence. Were the cache line dirty, the data modifications would be written back to memory. This instruction orders with respect to each others, but also with respect to writes, locked read-modify-write instructions, and fence instructions. From this description, we develop that a program may use `CLFLUSH` only to persist writes, and have a behavior analogous to Pelley's *strict persistency* with *sequential consistency*. This instruction implicitly orders with basically everything, making for an expensive primitive that further reduce concurrent persist opportunities. In that, it is typically avoided and not even considered in persistent programming.
- `CLFLUSHOPT` *Flush Cache Line Optimized* invalidates the cache line containing the memory location pointed by the source operand, but does not wait for persistence. This instruction orders with respect to older writes to the same cache line, and locked read-modify-write instructions or fence instructions only.
- `CLWB` *Cache Line Write Back* writes back the cache line (if modified) containing the memory location pointed by the source operand, it does not wait for persistence, and does not invalidate the cache line from any level of the cache hierarchy. It follows the same ordering constraints than `CLFLUSHOPT`, and in that consists only in a performance optimization where the cache line may be retained in the cache hierarchy to avoid subsequent cache misses.
- `SFENCE` *Store Fence* orders all memory stores (and persist write back) prior to the instruction, such that they are globally visible before any store (and persist write

back) occurring after the SFENCE. This typically means waiting for a handshake from the WPQs until all previous writes-back issued by this thread have reached their WPQ.

A funny observation made by performance studies on Optane, was that in all current Intel microarchitectures, CLWB is not properly implemented. In particular, the opcode is available, but the operation executes without the optimization, thus remains identical to CLFLUSHOPT.

For the sake of clarity, we will adopt from now on a simplified view of these hardware instructions. We already ruled out CLFLUSH due to over restrictive semantics. CLWB is in all aspect an optimized version of CLFLUSHOPT. This leaves us with CLWB and SFENCE only, or equivalently pwb and pfence to remain architecture independent.

Practical flush ordering. We stated that CLWB orders with respect to any store-fencing instruction, or anterior stores and CLWB to the same memory location. This leads to three fundamental remarks:

- "Any store-fencing instruction" includes SFENCE obviously, but also more lightweight write barriers and most importantly, any lock-prefixed read-modify-write instruction. Meaning all instructions commonly referred to as « atomic primitives », e.g., *fetch-and-add*, *compare-and-swap* (`lock xadd`, `lock xcmpxchg` in x86 assembly) and so on. For short, store barriers and atomic primitives effectively delimit persistency epochs. A first aftereffect is that it makes it impossible to encounter Pelley's *persist-epoch races*; a second is that it makes the pairing of SFENCE redundant with atomic primitives as they already produce an equivalent persist barrier and start a new persist epoch on their own.

- Persist writes-back and older stores to the same cache line will not be re-ordered with CLWB. Meaning a thread-local sequence of byte-wise stores to the same cache line will not be re-ordered, and a single CLWB can be used to persist write-back the whole cache line. SFENCE in this case can be omitted for persist ordering, so long the stores affect the same cache line.

- Memory stores to PMEM addresses are always ordered by SFENCEs but they will not be made part of any persistent epoch so long no CLWB is issued to these memory location. Meaning these memory stores will become persistent implicitly when the cache management protocols evict them. Not issuing CLWB to an updated PMEM location will then not force that location in the current persistent epoch. Leveraging this behavior for increased parallelism can sound tempting. Remember however that these implicit stores will also disturb the internal write buffering of Optane physical blocks, and most likely restrict available bandwidth. That is, because the cache protocols will implicitly evict 64B cache lines in a random order, when the Optane internal write granularity is 256B, leading to fourfold amplified writes.

Last, CLWB and SFENCE are intrinsically expensive. Of course, SFENCE reduce parallelism by constraining ordering in respect to both persistency and consistency. Above that, each one of them approximately cost 95ns, due to the on-chip synchronous handshakes with the iMC on the control path. Synchronizing at this level is necessary to allow the program to proceed only after a persist write was pushed to its WPQ. Because of these costs, reducing the number of flush and fences in programs and algorithms is

key for good performance. In particular, multiple consecutive flushes will pipeline, but flush-fence pairs will not.

2.6.2.2 Ordering persistent stores for crash-consistency

In summary, we thankfully now completed the series of explanations required to grasp the resolution of the write-back problem (challenge 1). We had stipulated that implicit flushes were the cause of consistency loss upon system faults. Assuming x86 memory model without persistency extensions, nothing could have prevented memory stores to PMEM from becoming persistent. In the wake of a crash, no assumption and logical reasoning on persisted data could have guided recovery in inferring the pre-crash execution state. We now know that memory stores to PMEM locations can be paired with a `CLWB` instruction to register them for asynchronous write-back, which conceptually places them in the current persist epoch; and that persist epochs can be delimited with `SFENCE` instructions (or equivalent) for ordering. These simple mechanisms are the base building blocks we will need to work with to implement recovery routines, and other mechanisms we surveyed from past persistent memory or persistent object systems.

As to understand how to properly wield these new instructions in programs, assessing the correctness and safety of persistent programs is the next logical development. After a thorough overview of the model, intuitively, we would deem a PMEM abstraction correct if no inconsistent state might be recovered in the wake of a crash. In single thread environment, it is made trivial by the following pattern: *(i)* issuing a `pwb` for every dirtied cache line on PMEM, *(ii)* issuing a `pfence` before updating the metadata associated with the progress of the executing thread.

However, when considering concurrent thread executions, genuine head scratching begins. The two following executions highlight the issue we dubbed « the read of not-yet-persistent-writes » (challenge 2):

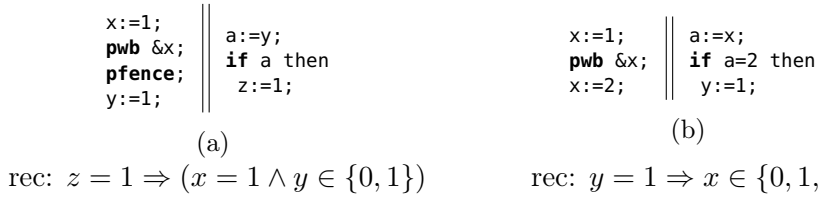


Figure 2.1: Misleading executions of concurrent persistent programs from [266].

Figure 2.1a illustrates that global visibility of memory stores do not imply those stores were made persistent. In this sample execution, having thread τ_b (right) recover $z = 1$ implies thread τ_a (left) had entered its second persist epoch before the fault. That is because in thread τ_b, $z = 1$ has a causal dependency on $y = 1$ from thread τ_a, hence the $y = 1$ store was made globally visible before the fault. Which in turns implies thread τ_a had executed its `pfence`, allowing $x = 1$ to be persisted. However, y is undefined after recovery, because nothing prevented it from being globally visible *before* persisting. Additionally, even though $z = 1$ was recovered, no persist order was enforced between $z = 1$ and $y = 1$, thus the two writes might have been re-ordered and z persisted before.

Figure 2.1b exhibits that an unreachable state during normal execution might be recovered after a crash. In a normal execution, thread τ_a or τ_b may never observe $y = 1 \wedge x \neq 2$. The alert reader may think that, after recovery, $y = 1 \Rightarrow x \in$

$\{1, 2\}$, that is, $x = 2$ was or was not persisted. The alert reader further assumes that, because we recovered $y = 1$, we know that `pwb` had executed in thread τ_a, and $x = 1$ is the oldest possible recoverable state. This assumption is based on the fact that `pwb` may not be re-ordered relative to previous writes to the same location. However, nothing prevents `pwb` to be re-ordered relative to later writes to the same location. That is to say, $x := 2$ could have been executed *before* `pwb` in thread τ_a and be made globally visible. After which $y = 1$ could have been persisted from thread τ_b, with odds of a crash occurring before `pwb` was ever executed in thread τ_a. Leading to a possible $x = 0$ on recovery.

As the head scratching intensifies, one might be lost under what seems to be an overwhelming programming model with room for persistency bugs at almost every single lines of code. When `pwb` should be used? On what data? Where to place fences? What about failure-atomicity? And recovery? How to tell it's bug-free? Is it optimal?

2.6.2.3 Correctness criteria

Traditionally, programming complexity is detangled by frameworks for reasoning about system correctness, with properties that come to the rescue of expert programmers. In the field of concurrent programming, the standard safety criterion of transient data structures is *Linearizability* [163], proposed by Herlihy and Wing in 1987. Admittedly, we realize that linearizability proofs are mind-boggling on their own; yet they are a, slight, improvement over the initial problem complexity. For starters, proofs can be aided by standard methodologies, but more importantly, thanks to composability, abstractions can be stacked and proved individually, providing welcomed opportunities for problem decomposition.

Linearizability assess that a concurrent execution can be made equivalent to some single threaded execution. In other words, there exists a legal single threaded execution that respects: *(i)* program order within each thread, and *(ii)* "real-time" order across threads - non-overlapping operation invocations can not be re-ordered. A program verifies *linearizability* if and only if all of its executions do.

Durable Linearizability [174] was proposed in 2016 by Izraelevitz as an extension of linearizability for persistence. **(Model)** The idea is to introduce the notion of full-system crashes in program executions, and to reduce a faulty execution into a classical fault-free concurrent execution by eliding operations that are concurrent to crashes. The rational being that, for any operation concurrent to a crash, since a recovery procedure must run after the crash, then the sequencing of the operation-crash-recovery is equivalent to either no operation or the fault-free operation. **(Def. 1)** An execution is then *durable linearizable* if it is linearizable after cleaning the crashes. Similarly, a program is *durable linearizable* if and only if all of its executions are. Since full system crashes can basically happen at any time, this implies that no system crash will induce inconsistencies in respect to both persistence and concurrent accesses. **(Def. 2)** Friedman et al. in [138] proposed an equivalent (was proven to be) definition of *durable linearizability*. On their terms, an execution is *durable linearizable* if and only if: *(i)* every operation persists before returning, and *(ii)* the persist order matches the linearization order. **(Locality)** Conveniently, *durable linearizability* was also shown to be *compositional*, just like regular *linearizability*. **(Buffered d-lin)** A buffered variant, *Buffered-Durable Linearizability* was also defined as a weaker property that allows for operation to return before persisting data, for performance at the cost of data loss. That is, because operations do not synchronously persist data, some data may be lost after a crash, but the reminder is

consistent with the persist and concurrent access order established before the crash. **(Persist points)** With linearizability, the concept of linearization points is often used as a proof writing strategy. An analogous concept also exists with *durable linearizability*. For short, all stores must persist and be fenced *after* the operation linearization point and *before* the operation return value or the linearization point of concurrent operations.

Izraelevitz, still in [174] after defining the durable linearizability correctness property, proposes a mechanical program transformation that guarantees correct persistence. Their formalized memory model extensions for persistence apply to *release consistency* model, but as a subsume of x86, they apply on Intel as well. His mechanical recipe consistently decorates *store*, *load*, *store-release*, *load-acquire* operations with *pwb*, *pfence* instructions. This transformation takes any *linearizable* base program into a *durably linearizable* program, which has a *null recovery procedure* if the base code was lock-free.

Although this mechanized transform is no optimal solution to our (challenge 2), it demonstrates that any linearizable program can be adapted to be durable linearizable. In other words, that under Intel's persistent memory programming model, correct concurrent programs can always be re-written to be consistently correct in regard to both concurrence and persistence. Which implies that (challenge 2) always knows a solution, even if reaching it is another challenge of its own.

2.6.2.4 Crash-consistent concurrent objects

As a side note and for reference purposes, we must mention that *durable linearizability* is not the only safety and correctness criterion for persistent executions. While Izraelevitz's work has most certainly became the *de-facto* property used to gauge integrity and consistency of persistent data throughout program executions; other researchers have taken the opportunity to tackle the fundamental problem of formalizing concurrent objects behavior in crash-fault environment.

Linearizability extensions Anterior proposals to extend *linearizability* and include crashes in history include: *strict linearizability* [32] by Aguilera and Frølund, *persistent linearizability* [154] by Guerraoui and Levy, and *recoverable linearizability* [62] by Berryhill and Golab. Each one of them sacrificed either *locality*, program order after a crash, or precluded implementations of some wait-free objects to retain *locality* and program order. In cause their failure model that allowed for individual process crash and recovery. In contrast, Izraelevitz with *durable linearizability* assumed a full-system crash closer to real-world system faults. In doing so, they noticed that the former definitions were indistinguishable and that their previously known limitations were no longer observable as well.

Recoverable synchronization primitives The question of *recoverable mutual exclusion* have been studied by Golab and Ramaraju [147]. They proposed revised deadlock-freedom and starvation freedom progress properties in presence of independent process faults and an algorithm to implement recoverable mutex. Similarly, recoverable counterparts of non-blocking synchronization primitives such as *compare-and-swap* (CAS) were proposed as *persistent multi-word CAS* in [255] and [315], or as hardware-based *persistent CAS* for ARM in [316].

Recoverable consensus Herlihy's consensus hierarchy is a framework for ranking the synchronization power of various primitives for solving consensus in a concurrent setting

on shared-memory. Golab in [146] revisits this hierarchy in a model with crash-recovery failures, forming a new problem specification called *recoverable consensus*. They observed afterwards in [62] that Herlihy's universal construction results still carry over in a model with system-wide failures by placing shared variables in non-volatile memory and using *recoverable consensus* in place of *consensus*. Delporte-Gallet, in [119] extended these results to individual process failures by exposing additional conditions necessary and sufficient to solve *recoverable consensus* under this less restrictive failure model.

Detectable objects In addition to tolerating crash faults and staying consistent, the study of recoverable implementations consist in a new field of interest. The concept of *detectable objects* was introduced by Friedman et al. in [138]. *Detectability* ensures that in the wake of a crash, state of on-going operations at the time of the crash can be inferred, as either completed or aborted. In contrast to *durable linearizability* that elides operations concurrent to crashes, *detectable recovery* ensures that every crashed operation recovers and returns a correct response. In NRL (*nesting-safe recoverable linearizability*) [50], Attiya et al. consider a composable approach to constructing recoverable objects. Under their single-process fault model, operations are extended with a recovery routine that enables invocations to be prolonged until they return, for as many crashes as necessary. Attiya et al. describe in [51] a tracking method for *detectable recovery* of concurrent lock-free data structures. The space-complexity cost of abiding to the new *detectable* correctness condition was explored in [56] by Ben-Baruch et al. Li and Golab [214] have proposed a sequential specification for detectable shared objects (DSS), later refined by Moridi et al. in [235] with *unified-DSS*.

These more recent evolution in the realm of formal specification of persistent executions bring new opportunities for constructing valid concurrent recoverable programs. On one hand, *durable linearizability* defined a practical framework for studying safety and correctness of concurrent faulty executions, but gave no cues on how to ensure forward progress. On the other hand, recent *detectable objects* embed recovery routines in their specification, providing precise semantics to resolve post-crash states. A sequential specification for recoverable objects makes overall for a more portable framework; in that it can be combined with off-the-self criteria for concurrency, under shared-memory or distributed settings, with system-wide or individual process faults. Besides, the study of *recoverable consensus* opens up opportunities for universal constructions and automated translations of code into their concurrent crash-consistent counterpart. We detail practical universal constructions for NVMM in §2.7.2.

2.6.3 Persistence domains

The persistent memory programming model is undeniably complicated, as this lengthy section can attest. Looking for answers to (challenge 1) and (challenge 2) required more than a simple look over, and had us dig deep into subtle topics of the model. The deeper we dug, the less confident we grew of its practicality. (§2.6.2) As a matter of fact, it is unreasonable to expect any developer to pick up on fine grain store ordering and careful placement of barrier and flushes. Let alone hand proving their programs by reasoning at the level of assembly instructions. The matter is albeit of all concerns. Remember the dire consequences of persistency bugs: not only could they jeopardize data consistency, but also induce permanent corruption in a persistent heap. In worst cases, all data were to be forfeited and the program to start all over again from a fresh and empty persistent

heap. Permanent data loss is too big of a blow to endure, for that, data integrity can not be entrusted to such brittle designs. Therefore, the programming model must be revised, extended or abstracted in order to rule out bugs coming from hazardous manual flushes and fences.

2.6.3.1 Extending persistence to caches

Pelley (§2.6.1.1) showed us that *synchronous ordering* would stall the execution to order persistent memory writes, exposing higher latencies in the critical path. Even so, its compatibility with traditional consistency model is appealing. Another possible route to blend persistent memory in traditional memory models, lies in extending the persistency domain to further reduce latencies in the critical path. By introducing ADR, Intel typically extended the persistency domain of NV-DIMMs to the CPU's integrated memory controller, allowing for lower-delay handshakes. On power-fail, residual energy from the power supply is enough to completely drain the iMC write-pending queues and persist data.

eADR, for *Enhanced Asynchronous DRAM Refresh*, extends the persistency domain to include CPU caches. It was introduced by Intel in persistent memory specifications starting with the second generation of Optane modules. eADR is an optional platform requirement for persistent memory, and currently, no known server vendor have advertised an implementing appliance. The specification describes a *flush-on-power-fail* mechanism that guarantees CPU caches to be drained on system-wide crashes. This operation is expected to take more energy than the residual one typical power supply can provide. For this reason, manufacturers are free to introduce batteries or rely on uninterruptible power supplies for supporting eADR. This approach however introduces new hardware components (batteries) that require maintenance routines, are not easy to dispose of or recycle, and can also be faulty themselves. Assuming eADR, individual cache line flushes are no longer necessary, but `SFENCE` do remain, so as to order writes within persist epochs. For instance, any concurrent program verifying *linearizability* would be *durable linearizable* with eADR; because essentially, its persist points would match its linearization points [25, 346].

Research work equally tried simplifying the memory persistency model by introducing new hardware components and making part of the cache hierarchy persistent. **Whole-system persistence** [242] used the same flush-on-fail idea but also made CPU registers persistent, in order to provide suspend/resume semantics to processes in the event of a fault. We need not to re-detail why the concept of persistent processes is unattractive. **BBB** (Battery-Backed Buffer) [40] improves over eADR by reducing the energy required to drain by two orders of magnitude. **DPO** (Delegated Persist Ordering) [200], **HOPS** (Hands-off Persistence System) [240], or **StrandWeaver** [145] all proposed introducing some persistent buffer (battery-backed) alongside the CPU cache hierarchy, together with amendments to the memory model. From an external perspective, they still require the programmer to properly hint ordering constraints with special instructions, and in that provide no ground-breaking programming benefit over eADR.

2.6.3.2 Hardware transactions

In fact, the *ordering only* approach, that most of the research work considered when revisiting memory models for persistent hardware, has semantics that makes reasoning about recovery extremely cumbersome; as underlined by Gogte et al. [144] while making

their point for *failure-atomicity* of groups of writes. No code is solely made of non-blocking data structures that have *null recovery* when durable linearizable (otherwise called *log-free* [111], cf. §2.7.1). Failure-atomic blocks of code cover for crashes of arbitrary code and further help reducing the amount of software implementation required to clean up persistent state on recovery. We have seen interesting efforts in providing hardware-assisted failure-atomic constructions in the following recent work.

JustDo [173] is designed for machines with non-volatile caches, and proposes to resume interrupted or crashed failure-atomic sections to completion. A small log checkpoints in caches the program counter and live registers prior to every store. In the wake of a crash, the remainder of any interrupted lock-guarded failure-atomic sections can be executed. JustDo limits the write traffic because actual values and payloads are not flushed to media for logging purposes.

ASAP [30] leverages hardware-level logging (WAL), but allows for asynchronously persisting data after transactions commit. Ordering is preserved by tracking in hardware data interdependencies between failure-atomic regions.

PMEM-spec [178] introduces a novel hardware-software co-design to support failure-atomic regions; that allows *strict persistency* to rival with *relaxed persistency* models found in hardware. The idea is to allow optimistic access to PMEM (without stalling or buffering), detect potential ordering violation, then leverage atomic region aborts to recover any misspeculations as if they were power-failures. Their design requires no modification in CPU cache hierarchy (no persist buffer alongside caches), no changes in cache-coherence protocols, and no batteries. The only hardware change needed is for persistent writes to bypass CPU caches altogether and directly be sent to the PMEM controller. That is because they install inside the PMEM controller their own special *speculation buffer* data structure that handles most of their algorithm's logic. On the software side, the implementation of failure-atomic blocks must support an abort handler that discards data and re-executes the transaction from the start. It must also register a misspeculation handler that receives misspeculation detection signals from the OS.

2.6.3.3 Limits of hardware-based solutions

Finally to conclude, eADR, persistent caches or broadly speaking, any other approach that extends the persistence domain to even include CPU registers, are no silver bullet. Do not get us wrong, they are helpful in reducing the performance cost of flushes or barriers, but simply put, they are no permanent fix to the persistent memory programming model. The need for manual ordering remains, and delimiting persist epochs alone does not lift the burden of implementing recovery from the user. Moreover, by introducing new hardware components they also introduce new fault scenarios that software would have to cover for anyways. The promised performance gains are however interesting, software just need to stay aware that only power outage or any system-wide failure are covered, not individual process crashes. Further, we learn from novel hardware-assisted failure-atomic protected sections, that software involvement can only be reduced, never fully eliminated. Overall, exclusively hardware-based solutions will always come shorthanded when it comes down to making the persistent memory programming model more accessible. Meaning that the only way forward is through software constructs that provide relevant abstractions for programs to reason about persistence in an easier, more high-level way.

PMEM-spec is interesting, in that it does not claim to solve all PMEM programming issues in hardware, but reckons reducing PMEM programming hardships may require

appropriate software cooperation. We however do not want to assume hardware changes and going forward, will only consider software-based approaches to persistent memory programming. Going about hardware solutions to PMEM programming issues was necessary though: so as to understand their inherent limitations and ultimately, the definitive role of software in modern NVMM-based persistent memory. Understand that transitional steps are always welcomed, but above and whenever possible, we guilelessly prefer contributing on systems and abstractions that may last beyond uttermost hardware improvements.

2.6.4 Ensuring failure-atomicity

We realize that this section on PMEM challenges at the NVMM era might be a bit of a drag at this point - or that it might already have been starting few pages ago - but please, bear with us for a little more; methods for failure-atomicity are truly the corner stone of this new breed of persistent memory.

We gathered so far that ordering alone was sufficient for correctness of programming constructs, and that it fell onto software to properly order persistent writes. However, we made it clear that reasoning at the ISA-level and manually ordering persistent stores were extremely brittle and bug-prone ways of programming persistent memory; even more cumbersome when going about recovery.

Under these low level abstractions, programs would have to self-tailor recovery implementations. Not only to infer anterior program state from persisted data, but also to correct and clear any inconsistencies left by implicit writes. Which is as complex as finding out whether any operation was on-going at the time of a crash, and whether they successfully persisted. Then adopt the appropriate strategy to un-persist any outstanding inconsistent data originating from operations that could not be completely recovered.

Although this methodology might be applicable on a small scale, e.g., to a single data structure, it completely falls short when considering operations that are purely transactional. That is, whose effects apply on a bundle of data that can not be directly related by navigating existing in-memory references.

Def. *Failure-atomicity* is the term commonly used to denote operations to appear as *atomic*, i.e., to execute in a all-or-nothing fashion, with regards to persistence. Similarly to the *atomicity* property found in concurrent settings which stipulates that intermediate states of an atomic operation are invisible to concurrent observers; *failure-atomicity* prevents intermediate states from being visible to post-crash observers.

This notion is not exclusive to persistent memory, but a broader issue common to all abstractions for data persistence. Recall that any anterior design we deemed usable (§2.2, §2.3.2) had to feature an explicit way of denoting consistent application states that must persist. In doing so, the program was guaranteed to resume from one of these state, which made reasoning around recovery much simpler. In the systems we went over, failure-atomicity was assured by either explicit checkpoints or durable transactions. Both are compatible with the epoch persistency model of persistent memory, from an interface standpoint.

2.6.4.1 User-level APIs

Failure-atomicity is most often envisioned through • ***transactions***, or more broadly

speaking, • ***failure-atomic sections*** of code. Note we avoid referring to them as transactions when they only ensure failure-atomicity and not the classical ACID properties. Less commonly, explicit • ***checkpoints*** may also be used to create a separation between two persist epochs and identify persist points (recoverable states) in programs. Finally, individual • ***data structures*** and their operations might also appear as failure-atomic from an external viewpoint. Either because they may be *log-free* and have *null recovery*, or thanks to some internal clean-up code that is invoked first thing on recovery to re-attain a consistent state.

Failure-atomic sections of code are genuinely appealing for the general-purpose solution they provide to all three challenges we identified for persistent memory atop NVMM. Turning any arbitrary code sequence, identified as easily as some critical section, into a crash-consistent execution; where any undesirable intermediate state is guaranteed not to be recovered. Although this abstraction have been well tried for databases, we have already seen with RVM and Rio-Vista that durable transactions for main memory required to be reworked substantially. Similarly, NVMM introduces a singular programming model and peculiar performance characteristics, requiring protocols for failure-atomicity to be entirely revised.

2.6.4.2 Basic implementations

The two major hurdles to consider then when adapting techniques from spinning drives to NVMM, are: *(i)* the fact that nothing can prevent writes from becoming persistent, and *(ii)* that 8-byte stores are the largest to appear as *atomically persisted* at the hardware level, on Intel platforms. In particular, previous failure-atomicity schemes for traditional storage media could exploit memory volatility to host and traverse inconsistent program states, and explicitly persist changes on commit. For instance, only record the data changeset during the execution of failure-atomic sections of code, and reliably commit mutated media blocks with *physical logging*.

Conversely, with NVMM, inconsistent transitional states might be implicitly persisted with extremely fine grain while executing failure-atomic sections; which demands both versions of data (the working copy and consistent copy) to exist concurrently in persistent memory and to be atomically joined together on commit.

The resulting solutions typically involve forms of *logging* or leverage *out-of-place updates*, akin to database WAL and shadow-paging. Logging must be performed *ahead-of-time* or *on-the-go*, with appropriate fencing to avoid compromising the consistent version of data. Out-of-place updates require complex read and write instrumentation to access the working copy of data within failure-atomic sections.

Since at most 8-byte stores may be atomically persisted, all schemes working at the physical level are challenged by the prohibiting overheads of *tracking* and *logging* changes with such extreme fine grain. Systems either choose to work at a larger grain, e.g., 64B cache line granularity, 256B Optane block size, or even at a logical level. The latter commonly involves cooperation from the language runtime and compiler or stands as more programmatically intrusive. As an illustration, let us consider the most basic schemes we already discussed while covering databases: *undo logging*, *redo logging* and *copy-on-write*.

(Undo logging.) Every value is logged before being mutated, such that previous values can be recovered from the log and operations rolled back. This scheme requires a minimum of one flush-fence pair for every new entry in the log. The most

compelling benefit is that reads inside failure-atomic sections are regular, uninstrumented loads.

(Redo logging.) Every new value is logged on-the-go, such that previous values are left untouched at their home location. Only one fence is needed to mark the whole log complete, after which logged (new) values can safely be applied to their home location on commit. On recovery, the whole log is discarded when incomplete, or rolled forward when complete. The appeal for this technique comes from asynchronous writes throughout the failure-atomic section, which needs only to be ordered before starting to apply changes to data home locations. The main difficulty comes from reads that must be instrumented such that read-after-write accesses return the new value.

(Copy-on-Write.) Every value is duplicated and homed to a different location on first access in failure-atomic sections. An extra layer of indirection is required to direct subsequent accesses to the new home location of the data. On commit, new locations become the real locations, the old ones may be discarded and deallocated. New values can be allocated and written asynchronously and persisted with a single barrier on commit, but the extra indirection layer and the possible memory leaks on commit makes it a very situational and contextual choice.

These three basic schemes were extensively compared in [228], unfortunately, before real NVMM were available. Conclusions we can draw from them are:

- **They increase write traffic:** logging suffers from a *double write problem* and copy-on-write from a *write amplification problem* for too small updates.
- **They need more fences than the minimum:** one for each log entry with undo, generally four with redo (1 to log updates, 1 to mark log complete, 1 to apply logged values, 1 to mark log empty), at least two with copy-on-write (1 to persist new values, 1 to atomically update the index).
- **They require instrumenting reads and writes:** all three interpose on writes, while redo and copy-on-write redirect reads as well to tolerate their out-of-place updates.

Ensuring failure-atomicity of generic executions is thus a space of subtle trade-offs between the type of scheme employed and the granularity at which updates are tracked. As to optimize for multiple performance criteria such as the number of flushes and fences, *write amplification* (ratio of total bits written to NV-DIMMs for every bit of application payload), and access patterns favorable to Optane NV-DIMMs.

2.6.5 Summary

To conclude on PMEM challenges at the NVMM era, the PMEM programming model is a substantial hindrance for broad adoption of PMEM in the spectrum of interfaces for data persistence, as the length we went to even express the problems entails. Showing how (challenge 1) and (challenge 2) could be overcome from Intel's extensions was nonetheless necessary to motivate why this complexity had to be broken down for PMEM to be practical and eventually successful. With NVMM, proper ordering is a dead hard requirement to fulfill, yet vital before even thinking about renewing the meaningful abstractions and design ideas PMEM from the past have laid for us. All convenient support

for failure-atomic sections might be, in that it lends a well-tried programming abstraction that solves all three challenges; implementations for NVMM come with deterring overheads. Past persistent object systems could encompass failure-atomicity with a one-size-fits-all solution, such as checkpointing or durable transactions: whose overheads were minimal compared to media access latencies. Conversely, persistent memory with NVMM will be made practical by offering multiple programming abstractions for failure-atomicity. A complete *palette* for developers to compose with, depending on their needs toward guarantees, performance, and productivity.

We can already tell that failure-atomicity for NVMM is poised to hover between specialized recovery schemes for performance and general-purpose constructions for practicality. Data structures and objects may implement self-tailored and finely tuned failure-atomicity through precise ordering and on-point recovery, while being proven or verified correct locally. Whereas general purpose failure-atomic sections, due to their reduced performance, might be avoided except for purely transactional computations; perhaps also where productivity precedes performance. Consequently, language-level support for persistent memory will have to accommodate and arrange for all these relevant atomic programming abstractions to be implementable, but more importantly, composable together *and* composable with crash-consistent heap management (persistent memory allocation, deallocation, and permanent object reference management). That marks the consideration with which we will cover present work that propose programming abstractions for persistent memory in §2.7.

2.7 Persistent Programming Abstractions for NVMM

We just made the point for failure-atomicity to be the missing link between past persistent memory and its renewal in the NVMM era. The never-before-seen blending of media direct access with fine-grain persistence (flush & ordering) raises a new host of challenges for crash-consistent memory data structures atop NVMM.

Recall that fine-grain instructions make suboptimal (§2.6.4) the well-tried and easy-to-reason-about past programming abstractions for sound crash recovery. That are durable transactions and checkpoints, in regard to write-amplification and the minimal number of persistence instructions. All the while, direct access also incapacitates system software from interposing on the data path. Reasons for which answers to these challenges will rather come as programming help and support for failure-atomicity, namely, a set of libraries, tools, language compilers and runtimes.

The shape of these facilities however remains an active area of research. The broad outlines of which were traced by researchers starting couples years before the announcement of Optane NV-DIMMs. These prototype systems demonstrated usefulness of several programming abstractions, providing situationally-optimized solutions to failure-atomicity. They can be parted in four broad classes:

- **Data structures**. Either purpose-built and finely hand-tuned or mechanically ported on NVMM, persistent data types offer specialized crash-consistent operations at their interface. We cover both design options in §2.7.1 and §2.7.2.

- **Transactions**. NVMM has re-ignited interest in thin software-based durable memory transactions, with guarantees ranging from full-blown ACID to minimal failure-atomicity; that we discuss in §2.7.3.

- **Checkpointing**. A minimalist interface to denoting application consistent states, we relate in §2.7.4 its employability on NVMM.
- **Heap management**. Techniques for crash-consistent memory (de)allocation and persistent reference management will close this section in §2.7.5.

We do not plan on extensively surveying this ever-expanding body of work in this section. First and foremost, because a newly published (September 2022) survey by Baldassin et al. [54] already does an outstanding effort - more so than we ever could. Then, because our interests are more inclined towards requirements for language-level support of NVMM and appropriate heap extensions that may play well with library-level schemes for failure-atomicity.

2.7.1 Persistent data structures

First off, we must clarify: here we cover persistent data structures, as in designed for NVMM and persistent memory; not to be confused with the homonymous persistent data structures [126] from functional programming. This second kind designate history-preserving data types, that are immutable and keep observable former versions of themselves whenever modified.

Back to the matter, data structures are exceptionally useful constructs that aid organizing and arranging individual data objects in programs, or that capture relationships between multiple ones. Precisely, data structures are collection of values that define the external operations and interactions exercisable onto its data. Examples include searchable data types (e.g., B-trees, hash tables, skip lists) or sequential collections (e.g., queues, stacks). The former are used broadly for indexing in storage and optimize for fast lookup or insertion of (un)sorted values. The latter find use in any sort of event processing, as buffers that maintain inserted values in sequential order.

Hand-made non-blocking (lock-free, wait-free) concurrent data structures have been proposed in the literature for almost forty years. Being tailored for scalability in highly concurrent settings, they are key to designing efficient algorithms and therefore, are readily employed in demanding systems, including production-grade large databases (§2.5.3). On the flip side, they are optimized for a single data type and generally require a high expertise to understand the fine interleaving of operations that underpin their correctness.

Over the past few years, the same effort has been conducted to create persistent concurrent data types (PDT). We currently limit the discussion to PDTs that are manually built from bare `pwb` and `pfence`. Those are neither transactional nor log-based but carefully and finely tuned by brave experts, coming up with ad-hoc techniques for correct persist ordering and overall reduced number of persist barriers. Loads of research paper present new PDTs, mostly indexing structures: think tons of trees [44, 88, 198, 199, 252, 309, 339], hash maps [169, 223, 241, 243, 279, 353], skip lists [87, 91]; but also sequential data types, namely queues [130, 138, 282], where ideas from set-type structures hardly apply.

Bear in mind that Table 2.3 is not exhaustive and just scratches the surface, so much that research paper [212] even dedicates to comparing persistent indexes. We will therefore not present them individually, but rather try to extract and summarize key ideas and takeaways from PDT designs.

1. Leverage non-blocking (lock-free) structures for failure-atomicity.
In [111], David et al. stress out that "lock-free algorithms are a good fit for the NVRAM

Trees	CDDS B-tree [309], wB$^+$-tree [88], NV-tree [339], FPTree [252], Bztree [44], clfB-tree [198], PACTree [199]
Hash maps	NVC-Hashmap [279], Dalí [243], *Level hashing* [353], *Cacheline-Conscious Extendible Hashing* [241], Dash [223], Halo [169]
Skip lists	NV-Skiplist [87], UPSkipList [91]
Queues	*Durable lock-free queue* [138], OptUnlinkedQ & OptLinkedQ [282], PBqueue & PWFqueue [130]

Table 2.3: Notable persistent data structures

environment." Indeed, recall that lock-free linearizable algorithms always resume in a consistent state so long a thread stores persist in the linearization order (Friedman et al. definition in [138]). This property eliminates the need for *any form of logging* altogether, and allows executions to resume from a consistent state with *null-recovery*; as showed by Izraelevitz et al. in [174] with their mechanized transform for lock-free structures which abusively supplemented any store with a flush-fence pair.

David et al. expand on these ideas and come up with practical techniques for persistent pointer update and persistent memory allocation, that are sufficient to make lock-free structures failure-atomic. They amusingly coined such structures as *log-free*. The techniques themselves are *(i) Link-and-persist*, *(ii) Link-caching*, and *(iii) Epoch-based memory reclamation*.

- **Link-and-persist**. Non-blocking linked data structures commonly rely on atomic pointer update to make operations globally visible. We now understand that such operation may persist *after* being globally visible. For this reason, David et al. introduce a *mark* on newly added links, such that any *marked* links are logically interpreted as transient. Once both links and referenced data are guaranteed to have persisted, the mark may be removed safely. Were a thread to encounter a marked link on which depends the completion of its own operation, it would help persisting and un-marking it instead of blocking.
- **Link-caching**. Individually flushing and fencing link updates is costly. Even more so with the added *mark* metadata that must be lifted (and fenced) before data-dependent operations may proceed. In order to reduce the cost of *sync* operations in data structures, David et al. introduce a batching scheme for link updates. Newly added links are not immediately persisted, but rather tracked by a volatile set-like data structures, called the *link cache*. The link cache efficiently packs up to 6 links in a single of its cache-line-sized buckets, and flushes whole buckets (then at most 6 cache lines) with a single fence. When applied to a linked-list algorithm, the authors report that the link cache helped replace "writing back 4 cache lines one at a time by a single batch of 3 cache line write-backs"; that is, a single `pfence`.
- **Epoch-based memory reclamation**. Linked concurrent data structures extensively rely on dynamic memory management, to atomically insert, delete or update nodes. Special care is then necessary to avoid persistent memory leaks that could arise from crashes concurrent to operations that handle allocated data momentarily not linked anywhere in the data structure. This is usually taken care of with transactional allocation: before (de)allocating and (un)linking data, the intent is captured (durably) in a log record. Here, David et al. propose to reduce the book-

keeping overhead with coarser grain tracking of *active memory areas*. For short, each thread maintains for each allocator page the last "epoch" numbers at which it last allocated and deallocated memory from; such that on recovery, threads can inspect in parallel and clean up memory areas for which the recorded epoch number is high enough, relative to their own advancement epoch number. Then, on recovery, only active memory areas need to be traversed to detect and discard unlinked or marked nodes.

2. Persist in NVMM only data that are necessary for recoverability.
A recurring theme in achieving good NVMM performance is minimal bandwidth usage (especially writes) and minimal fencing instructions. In this regard, many PDTs thought of sidestepping the issue by allocating only recovery-critical data off NVMM, and reconstructing other metadata on recovery. Resulting as a whole in hybrid DRAM-NVMM data structures. The techniques is often referred to as *selective persistence*, and decouples the data layer from the search layer of data types. Allowing the latter to be managed at DRAM speeds with no overheads tied to failure-atomicity. That includes *(i)* structure traversal, *(ii)* concurrency control, *(iii)* structural modification operations (e.g., node balancing), or basically, *(iv)* any link update in the top layer. Selective persistence was first introduced by **NV-Tree** [339] and **FPTree** [252] that placed (data-holding) *leaf nodes* on NVMM but *inner nodes* on DRAM. In doing so, FPTree was also able to resort to hardware transactional memory (HTM) for concurrency control in its top (DRAM) layer, otherwise incompatible with NVMM-specific instructions; and fine-grained locking for the bottom (NVMM) layer.

Efficient lock-free sets, as proposed in [354], go even one step ahead and avoid persisting absolutely any link in NVMM. Indeed, they notice that most indexing structures are *link-free* as they boil down to sets. Meaning the whole structure can be reconstructed without any extra metadata, apart from knowing which record belongs to the structure. This applies to linked-lists, binary search trees, hash maps, skip lists and the likes. Each set then consists of a (thread-safe) data structure in volatile memory and a set of durable records in NVMM. Records are kept in special designated NVMM areas, which are scanned on recovery. Each record maintain membership information, that indicates whether it belongs to the set or whether its state were transient. Concisely, the proposed approach works as follows:

- *insert* first creates a record in NVMM, adds it to volatile structure then, if the prior addition was successful, marks it as valid.
- *contains* traverses the volatile memory, finds the matching record and return a reference to its content on NVMM.
- *remove* simply trims the record from volatile memory then marks it as invalid in NVMM.

During the last two operations, if a matching record is found while being inserted by another thread, then the inserting thread is helped. This ensures that the data structure remains durable linearizable. Interestingly, this algorithm is able to match the theoretical lower bound of one `pfence` per thread update, as established by Cohen et al. in [96]. Note also that, allocating and freeing nodes need not to be transactional as no leak may be caused: a node's validity scheme is used on recovery to reconstruct the node allocator state as well. In practice, only durable areas of a set need to be allocated crash-consistently and persistently referenced.

Still, hybrid data structures are ill suited when crashes are frequent or when reconstruction time matters. As one might have noticed by now, even though overheads are

reduced in crash-free executions, the approach forfeits opportunities for instantaneous recovery as well.

3. Optimize for current-gen NVMM hardware properties.
"Minimizing the number of persist instructions is an important topic, but not enough" [282]. As of 2022, three years after the launch of Optane NV-DIMMs, singular characteristics of real NVMM are now better understood. We remind that prior studies, as we discussed in §2.4.2, found:

- **Limited media bandwidth.** Quickly saturates under unrestricted parallelism.
- **Asymmetric read/write bandwidth and latency.** Mixed read/write workloads will most likely bottleneck first on write bandwidth, leading to underutilized read bandwidth.
- **Sequential accesses to hide media latencies.** Sequential reads benefit from CPU prefetching and the on-DIMM read buffer; similarly, sequential writes exploit efficiently the on-DIMM write-combining buffer.

PDTs [130, 199, 282] noticed persistence instructions to be more hurtful than first thought:

- **`CLWB` invalidates cache lines in current processor architectures.** As a consequence, persistence always incurs the cost of cache line invalidation; making especially expensive subsequent accesses to flushed content.
- **Contention leads to more severe performance degradation on NVMM.** Lengthy critical sections tend to be higher-latency operations, therefore are more susceptible to stalling concurrent threads, or make them retry on lock-free designs. Which might also translate into more wasted bandwidth.

Reading flushed content. In detail, the performance impact of *reading explicitly flushed content* is explored by Sela et al. in [282]. At first, their implementation of a durable concurrent queue did not clearly have an edge over the work of Friedman et al. [138], in spite of issuing optimal number of `pfence` (one per update operation, zero per read operation). However, their revised implementation that performs zero access to explicitly flushed content (**OptUnlinkedQ** and **OptLinkedQ**) improve throughput between two to threefold. Promisingly, by reasoning on the abstract universal construction of Cohen et al. in [96], they could show that any object might be implemented with zero access to flushed content.

Asynchronous concurrent index updates Comprehensive guidelines for efficient PDT implementations are also presented with PACTree [199]. They support various claims and patterns by measuring wasted bandwidth or excessive persist instructions. For instance in remote NUMA accesses to NVMM, or persistent memory allocation in the PMDK. Perhaps, their most notable idea is to *take off the critical path persistence of structural modification operations.*

Indexing data structures are susceptible to structural modifications (e.g., node splitting/merging, tree re-balancing) on key insertion or deletion, that can cascade to ancestor nodes. In persistent indexes, the performance blow is even more severe: these lengthy critical sections are made of high-latency persistent writes that will block or stall concurrent threads. Hybrid structures dodged the issue, by placing *internal nodes* on DRAM.

PACTree [199] conversely establish that, internal nodes can be kept persistent and compromising on recovery time to be unnecessary; when structural changes to the index

structure are made asynchronous. PACTree makes this possible by decoupling its search layer from its data layer, with a design that tolerates inconsistencies between the two. Both structures reside on NVMM for faster recovery, and optimize for their own doings. Precisely, the search layer employs a NVMM-consistent variation of a trie index because of reduced read bandwidth on traversal - only partial keys are stored and compared from internal nodes. The data layer then is a doubly-linked list of B^+-tree-like leaf nodes that conveniently enables sequential access for scan operations. When the data layer needs to be updated (node split or merge) to secure new key-value slots, only local changes happen synchronously. A logical persistent log tracks these changes to the data layer, so as to reflect them back to the search layer from a background updater thread. While the search layer might be out-of-sync, traversal operations would land on an adjacent node in the data layer. The inconsistency is then tolerated simply by navigating links to the correct node in the data layer. From their evaluation, placing the search layer in DRAM was not that beneficial (less than 10% throughput increase), thanks to the asynchronous background updates. Overall, the design of PACTree genuinely attests that hybrid persistent indexes can be built without impeding recovery or restricting the size of the index to that of DRAM.

In conclusion, persistent data structures provide tailored failure-atomicity at their interface. When hand-crafted, they may exploit custom schemes for failure-atomicity, finely intertwined with the data type's own characteristics. In that, being designed for a single-purpose lends them potential to outperform generic failure-atomic solutions.

Building a PDT from scratch however might feel like a daunting prospect: both expertise and original insight are necessary to design an efficient one. The task is unarguably harder than building volatile lock-free data structures. Leaving regular programmers with little design options were a structure not to fit their particular needs. Still, by going over multiple persistent data structure we found valuable insights and were able to extract ideas, findings and guidance applicable beyond the making of PDTs.

Notice that PDTs are often linked data structure and heavily rely on dynamic allocations, but do not solve the problem of memory management. To use them, it is necessary to integrate *(i)* their ad-hoc memory management mechanism, or *(ii)* to rely on an external solution. That second option usually means assuming a crash-consistent memory allocator, often transactional with *malloc-to*/*free-from* semantics. We cover in §2.7.5 other suitable techniques for management of persistent heap.

The way we have designed PDTs in our contribution (§4.5) is heavily inspired by *Efficient lock-free sets* [354], in that a mirror data structure indexes in DRAM the data nodes laid flat in a persistent array. By updating the persistent array before the volatile mirror structure, we guarantee that only persisted values are globally visible. Note that our DRAM mirror only holds *representant* objects that forward to the actual data nodes, to avoid data duplication in main memory.

Some authors [103] have also the questioned the efficiency of PDTs with regard to generic solutions. For instance, the queue implementation in [138] executes several `pfence` instructions, while the persistent transactional memory (PTM) proposed in [96] requires a single one. For this reason, we explore generic solutions to durable data structure implementation next (§2.7.2), and failure-atomicity of arbitrary code afterwards in §2.7.3.

2.7.2 General-purpose constructions for data structures

In the light of difficulty faced and dedication required so as to assemble new persistent data types; constituting a whole library of persistent container types (e.g., C++'s STL, Java's `java.util`) sounds to be quite the fairly serious endeavor.

As a more direct path, another body of work explored general-purpose frameworks to transition existing volatile data types into their NVMM-consistent derivative. Were proposed in research papers schemes to augment data type algorithms either from mechanical (manual) recipes that a programmer applies; or with automated transforms that work from instrumented code. In any instance, the programmer is relieved from non-trivial design decisions, while the resulting PDT has predictable efficiency and correctness guarantees.

Amongst the first representatives of these two kinds were the mechanical transform of Izraelevitz et al. from [174] that we already presented in §2.6.2.3, and the **ONLL** (*Order Now, Linearize Later*) universal construction of Cohen et al. [96]. Neither of the two were meant to be practical though: they rather served as theoretical proof material. Izraelevitz et al. showed with theirs that any lock-free object could be made *durable-linearizable*. Cohen et al. added a lower bound result on the number of persistent fence instructions per memory-updating operation and a detectable execution guarantee (cf. §2.6.2.4).

Then, over just the span of the last four years, numerous research papers have described methodologies that aimed at producing efficient converted implementations for real NVMM.

Capsules [57] proposes programmers to segment code into idempotent regions made of reads or private writes, called the *capsules*, and their boundary - a single CAS operation. Capsules are in effect processor-local checkpoints. The CAS at boundaries have to be replaced with the recoverable variant of Attiya et al. [50] for correct recovery of shared variables. Although quite a challenge implementation-wise, Capsules offer detectable recovery.

RECIPE [208] presents a manual conversion method for in-memory indexing structures. Based on three distinct conditions that DRAM indexes may match, they prescribe corresponding *conversion actions* to be implemented. In essence, these conditions capture non-blocking data structures that already possess appropriate mechanisms to avoid or tolerate crash-inconsistencies; and aid extending them for real NVMM with appropriate flush and fence placement. RECIPE was also extended for the making of UPSkipList in [91], by providing a mean of detecting post-crash inconsistency repair without relying on locks; so as to support algorithms with non-blocking and non-repairing writes. Lastly, RECIPE was found to yield incorrect persistent algorithms in some instances. Their revised version on ArXiv now adopts flush and fence instructions after every stores and most of loads in NVMM, almost as in the transform of Izraelevitz et al. [174].

PMwCAS [315] is a *persistent multi-word compare-and-swap* operation that provides CAS semantics across arbitrary words in NVRAM. The operation itself is lock-free and ensures that concurrent readers only observe persisted values. In that, the transition from volatile to persistent is seamless, nearly as easy as writing lock-based structures thanks to the multi-word interface. PMwCAS employs a *dirty bit* design where a mark is set on each word before being flushed and which is removed afterwards, to prevent concurrent read of uncommited data. On recovery, the whole pool of PMwCAS descriptors is scanned and each in-flight operation is rolled back or forward according to its own

progress. Correct memory management is simplified by enabling applications to install a callback to piggyback reclamation on PMwCAS's own memory recycling strategies. Although a powerful abstract base block, it was found to have limited scalability in write heavy workloads. [212] thoroughly analyzed and evaluated multiple persistent indexing structures, including Bztree [44] that internally rely on PMwCAS. Overall, Bztree had the highest count of flushes per insert/update/delete operation, mainly due to PMwCAS that incurs at least 3 flushes and fence when not contended.

MOD [159] observe that concurrent flushes (`pwb`) degrades the performance on Optane NV-DIMMs. To sidestep this problem, they propose to construct persistent data structures as functional ones. An update to a functional data structure is non-destructive [126]. Instead, it returns a new version of the object (for performance, this new version shadows the previous one). To inject data persistence, the recipe of [159] is as follows: before returning the new version, a single fence is needed followed by a flush on the pointer to the new version. We note that relying on functional data structures implies to massively copy in certain case the previous content (e.g., for a vector), leading to performance degradation. Moreover, it induces an overhead in maintaining all of the prior versions, as they report up to a 131x memory overhead.

NVTraverse [139] is a semi-automated transform that targets a specific class of lock-free data structures they named *traversal data structures*. They define traversal data structures as node-based tree-like whose operations first begin with a (possibly lengthy) traversal phase that locates nodes on which to perform the modifications. The key idea is to avoid flushing read locations during the traversal phase but only a few at the end to validate and persist the final location, then perform the operation. Although they describe how to automatically apply flush and fences to structures in appropriate *traversal form*, programmers would likely have to tune their concurrent volatile structures to fit their rigorous definition of a data structure in *traversal form*.

Pronto [232] is a software library that brings persistence to any volatile data types with minimal code changes. It achieves this through logical logging: the ASLs (*Asynchronous Semantic Logs*) capture operations and their arguments concurrently to the execution. The programmer needs only to wrap a data type and its operations with `begin`/`commit` frontiers, similar to transactions. Pronto maintains a volatile online image of the data structure against which runs the main execution thread. A background thread then records the logical logs on NVMM and synchronize on commit. The log is periodically compacted by taking consistent snapshots of the data structure, to reduce reconstruction time of the volatile image on recovery. The approach has little overhead over the unmodified data type, is trivial to use, and even outperforms hand-tuned structures; but has a large memory footprint and costly recovery. Basically, it mirrors the whole transient memory on NVMM and has to replay the execution from the last snapshot. Moreover, the approach doubles the number of maximum concurrent threads.

Persimmon [349] is an interposition mechanism to persist a state machine. [4] It relies on a persistent log to store the state-machine updates. These updates are then applied outside the critical path by a fork of the original program running in NVMM. To ensure failure-atomicity of the shadow NVMM copy, persimmon uses an undo mechanism. This mechanism instruments the program at runtime and persist every memory modification (by first saving its original content in the undo log). Persimmon does not handle concurrency, that is it only provides durability. Persimmon pays the cost of marshaling each state-machine update as well as blindly persisting the program heap in full.

[4] A state machine is a deterministic program without I/Os.

Applied to Redis, the authors show at best a 2x improvement over a file-system based approach (Redis AOF, fsync-ing every file modification).

CX-PUC [103] is the first universal construction to make a data type jointly persistent, linearizable and with wait-free progress. The approach is an adaptation of CX [104] for persistence, which is itself based on the combining technique of Herlihy [161] with a few modifications to enforce durability. The construction maintains $2n$ copies of a persistent object as well as a global mutation queue that establishes the linearization order. Upon executing an operation, a thread seals one of the copy, applies the missing mutations from the queue, then pending operation. After which, it tries to update the pointer to the most update-to-date copy. If this fails, the full procedure is repeated one more time. [5] The key interest of this approach is that recovery is immediate: the pointer to the latest copy being always consistent. However, it requires an upper bound on the number of thread (n), as well as persisting $2n$ full copies of the data. In addition, CX-PUC does not interpose on memory stores thus is only able to track changes with object granularity. A limitation they overcome by proposing a variant that combines with transactions for higher throughput, at the expense of additional annotations.

Montage [319] and **nbMontage** [77] provide the first transforms for buffered durable-linearizable structures of blocking or non-blocking types. Adopting buffered durable linearizability allows Montage to perform less than one fence per operation. They rely on a custom memory allocator [76] and epoch-based approach to recover an anterior image of the data type. The interface allows to selectively identify data nodes, thus avoids persisting data that can be reconstructed, reducing checkpointing overheads.

Mirror [140] works as an extension of the C++ *std::atomic* library in order to build hybrid data types. That is, it keeps a volatile replica of every persistent variables, to serve faster consistent reads directly from DRAM. The NVMM copy is accessed only on modifying operations to persist data. Intuitively, the transform updates crash-consistent variables before the volatile copy, and therefore, prevents reading *not-yet-persistent-writes*. Programmers need only to annotate types to use `patomic` instead of `std::atomic`, replace allocation calls with theirs; and for recovery, specify the root of the structure and a routine to traverse all nodes from the root. The mirror approach even outperforms NVTraverse by 4x on a linked-list for a write-heavy workload, and 10x for a read-heavy workload.

NAP [312] is a black-box approach that brings NUMA-awareness to hot items in existing persistent indexes. In essence, NAP introduces a wrapping layer that absorbs accesses to hot items, while cold ones are served by the underlying index. The NUMA-aware layer maintains a global view in DRAM, and a per-node view in NVMM; such that remote NVMM accesses are eliminated without inducing extra reads in NVMM. It is implemented in C++ and provide a wrapper template class for NVMM indexes, that takes around $\tilde{3}0$ lines of code to glue-on an existing type. NAP improves by up to 2x throughput in write-intensive workloads when using all four NUMA nodes of the machine.

PREP-UC [93] is a universal construction based on the node replication technique of Calciu et al. [78]. The technique was originally designed for NUMA-awareness. PREP-UC preserves that property and adds persistence with (buffered) durable linearizable variants. It utilizes logging and two persistence replica of the underlying sequential object. PREP-UC achieves significant performance improvements over CX-PUC, with lower spatial complexity.

[5] If the procedure fails two times, then another thread executes the operation and reports the result.

Tracking [52] presents a generic (manual) approach to derive *detectable recoverable* implementations of concurrent data structures. At a high-level, Tracking tags the structure's nodes with an operation descriptor that contains operation status, working set, and helping-relevant information. For recovery, every thread maintains a reference to its current operation descriptor. In spite of the larger (eyeballed) number of flush and fences their approach seems to entail, they found significant performance improvements over Capsules - the only other detectable transform to date. By disabling persist instructions and re-introducing them one by one (line after line), they could approximate the cost of each. Their analysis led to interesting findings: *(i)* persistent fences had nearly no impact on performance (with Tracking), and *(ii)* flushes to non-volatile memory had varying performance impact. In detail, they parted flushes in *low*, *medium* and *high* impact categories, and found *(i)* the low impact ones to match with private or freshly allocated variables that have not become shared yet, and *(ii)* the high impact ones to match with highly contended shared variables. With all that considered, Tracking exhibited mostly low impact flushes and some medium ones, which they used to conclude that most data being kept private explained the better performance.

PBCOMB and **PWFCOMB** [130] are two software combining protocols, blocking and wait-free, that can be employed to build data structures with *detectable recoverability*. Approaching persistence with combining is compelling, as it calls for a reduction in the number of persistence instructions and reduced contention on flushes. One crucial design decision for performance in PBCOMB is for the combiner threads to perform updates out-of-place, so as to maximize sequential and un-contended writes in NVMM before making the new copy globally visible. Their protocols leads to impressive throughput speedups when applied to sequential data structures (stack, queue, heap): severalfold better than with other UCs, PTMs or even hand-crafted ones. Combining however, restricts parallelism generally speaking. Even non-conflicting requests are serialized. For that, it might not be relevant in indexing structures. Additionally, out-of-place updates might result in a prohibitive copying cost as the structure grows larger.

We hope that this part could get us all a feel for how overwhelming persistent data structure can be. Devising conversion methods and automated transforms remains an active research area, as can attest the barrage of papers covering the topic. A perplexing balance between genericity, efficiency and correctness properties, that is indeed hard to get right as RECIPE illustrated.

To make matters worse, it comes unclear whether both principled, manual methodologies or automatic transforms are really applicable. Our main concerns are, that the preconditions expected by many techniques may necessitate non-trivial adaptations to the original data structure. Automatic transforms are not as immune to this as one may think: adhering to a specific API, using the right library calls or selectively making persistent allocations can also come out to be tricky to get right. In the end, it is definitely not a hands-off experience, but one with many tiny micro-adjustments that add up and turn out to be not so trivial for programmers. Additionally, the nearly generalized reliance on custom allocators, custom memory management or custom recovery fixed-points do not facilitate adoption in general-purpose persistent memory toolkits and libraries.

On the bright side, the aforementioned techniques provided us with experimentally-supported evidence for some efficient NVMM access patterns and situations they might be used in. Namely, the findings we take home are:

- **Use DRAM to avoid costlier operation in NVMM.** Pronto, Mirror, NAP, or even PCOMB all leverage DRAM to some extent in order to alleviate the cost of persistent instructions and slower NVMM.

- **Avoid persisting things read during a possibly lengthy traversal phases.** As entailed by NVTraverse, keeping a read-mostly phase and a write-heavy phase in sequence helps save NVMM bandwidth.

- **Use out-of-place updates to avoid small, fragmented writes.** MOD, or PCOMB notice that out-of-place updates do not always result in higher device bandwidth usage. Indeed, small fragmented writes with amplification may generate more write traffic, whereas creating a new copy promotes packed writes.

- **Avoid expensive flushes on highly contended data.** The analysis in Tracking clearly shows that not all flushes are equal. Low contention appears to be a recipe for faster explicit flushes, which promotes DRAM for highly contended objects.

The fact that these findings resonate with those we extracted earlier from hand-crafted data structures suggests we are indeed on the right track.

2.7.3 Persistent transactional memory

We previously discussed the need for *failure-atomic sections* (FASEs) and possible basic implementations in §2.6.4. In essence, FASEs form an all-encompassing solution to challenges of direct access to NVMM, bundled in the most well-tried and easy-to-grasp abstraction for failure-atomicity. Recall that we concluded however NVMM-specific implementations to be faced with inefficiencies when compared to special-purpose failure-atomicity. Namely, increased write traffic, extra fencing instructions and heavy load/-store instrumentation. FASEs might nonetheless be preferred for productivity reasons alone, since they immensely simplify coming by correct persistent code and going about application recovery. In any case, they are not entirely dispensable and remain necessary to atomically update any two piece of data that might not reference one another.

A versatility that is well-understood and welcomed as a failure-atomic abstraction: nearly all existing persistent object systems for NVMM include support for FASEs. Overall, when accounting for every research paper that proposes original implementation of FASEs, the body of work comes out as immense. Those that specifically present a novel logging technique, the others that propose one as part of a larger and complete persistent object system. As such and again, we will not be able to extensively present and compare them all. We remind however that the excellent survey from Baldassin et al. published this year [54] covers nearly them all, and provide a detailed comparison of decisive implementation aspects. Indeed, logging schemes can be distinctively tuned by working at different granularity, with varying support of thread-atomicity and crash-consistency properties.

The most common kind of FASEs found in the literature is undoubtedly transactions. Originating from *software transactional memory* (STM), a *persistent transactional memory* library (PTM) provides transactional semantics to programmers over regions of persistent memory. That is, within transaction boundaries denoted with `begin_tx`() and `end_tx`(), NVMM load and stores are ACID (or at least failure-atomic). We now examine notable PTMs.

Mnemosyne [310], is one of the first PTM designs dedicated to NVMM. It is built on top of the lock-based TinySTM transactional memory [133], and uses per-thread redo logs with global timestamps to preserve order when recovering transactions. Only four persistence fences are necessary to commit a transaction, regardless of the number of memory locations it modifies. Finally, since it has to log every memory word, it also provides automatic compile-time load/store interposition, to relieve the programmer from marking every transactional memory instruction.

PMDK [19], the official *Persistent Memory Development Kit* for Optane NV-DIMMs from Intel, contains an undo log implementation that reduces the number of persistent fences by aggregating all the modifications done on one memory object inside a transaction. PMDK transactions provide failure-atomicity, but not isolation. The persistent memory allocator is also highly optimized to reduce the number of `pwb` instructions. Programmers yet have to identify with manual special markings every memory load and store that are part of the transaction.

Romulus [102] is a lightweight PTM design that provides durable transactions through the use of two full copies of the data. Read-only transactions execute in isolation on one of two copies. Updates first execute on the most up to date version of the data (waiting its turn if necessary), then its modifications are played back on the second copy. For efficient play-back, mutations are recorded in a volatile redo log. If a failure occurs, any of the two copy holds a sound version of the data to restart from. Update transactions require at most four persistence fences, regardless of their sizes.

OneFile [267] ensures a stronger form of progress for updates, namely wait-freedom. To achieve this, the PTM uses a helping mechanism, a redo log for durability and a timestamp-based concurrency control. Though, every store in the sequential data structure require one double-word CAS to NVMM.

Redo-PTM [103] uses a variation of the CX universal construction [104] (CX-PUC) to construct durable linearizable data structure. As prior universal constructions, Redo-PTM relies on a queue of operations and multiple copies of the shared data. When executing a write, a thread first secures a copy then it applies the updates that were not played before returning. This approach has two main limitations: First, it requires to know beforehand the maximal number of threads (t) that concurrently access it. Second, it exhibits a write amplification of $O(t)$.

Since we have already partially discussed most common base logging schemes and their drawbacks on NVMM (§2.6.4), we propose to focus here on knobs available to PTMs for efficiency tuning:

- **Logging schemes.** We have established that: *(i) undo logging* needed one persist fence per log entry, *(ii) redo logging* needed 3-4 fences per transactions but at the expense of redirecting reads to in-flight updates, and *(iii) copy-on-write* required constant number of fences per transactions, avoided the double write problem of logging, but needed another intermediate indirection layer.

  Research work has then naturally sought to smooth out these sources of inefficiencies with variations of the techniques. For instance, **InCLL** [97] carefully lay out objects (data structure nodes) in memory to efficiently pack undo log entries in one cache line and save precious fence instructions; but is not applicable to generic FASEs. Alternatively, redo and copy-on-write are often combined one way or the other. First, as in Romulus to avoid full copies with CoW. Second, considering that Optane NV-DIMMs have a 256B block size, redo could also be made to work with

larger grain, while a shadow DRAM copy helps servicing reads during transactions. Last, hybrid logging strategies are also employed for HTM-PTM hybrids, as a way to sidestep limitations regarding persistence instructions in hardware transactions. In doing so, **Crafty** [142] is able to leverage HTM for isolation.

- **Tracking granularity.** Most often, updates are tracked at the word (8B) or object granularity with NVMM. Finer grain is often conceptually associated with fewer writes to media, while a coarser grain tend to reduce the overhead of the underlying mechanism. Remember that with spinning disks, we even encountered schemes that worked at page-granularity (4KB) with binary diffing techniques to reduce amplification. With the 256B internal block size of Optane NVMM however, finer grain might not always result in less effective bandwidth usage due to write amplification of small fragmented writes.

- **Thread atomicity.** In most of the above PTM designs, for the purpose of *isolation*, writes are serialized. At times, even resorting to a crude global shared lock. Such design decisions largely impede scalability on multi-core machines, especially hurtful in massively parallel workloads. By using per-thread logs and even deferring *isolation* to usual external synchronization primitives, scalability may be reattained. Special care is yet necessary in the overlap between critical sections and FASEs, as persisting partial updates in critical sections could for instance throw off durable linearizability.

- **Persistence guarantee.** Nearly all PTMs wait for persistence on commit, which are more natural semantics for programmers. With that, durable linearizability is obtained when they are designed jointly with concurrency control. Buffered variants are nonetheless still explored so as to amortize the cost of persistence instructions.

Lock-based FASEs Transactions, although dominant, do not consist in the only interface for FASEs. Among pioneering NVMM object systems, **Atlas** [83] proposed to infer FASEs from mutex-guarded critical sections. **NVthreads** [168] worked as a drop-in replacement for the `pthreads` C library and inferred consistent states from synchronization primitives. Atlas, NVthreads, and similar approaches might feel more natural to programmers for delimiting FASEs with already well-understood semantics of critical sections. Although such designs lack proper ordering in sequential executions, they might rather prove useful with the coming eADR-enabled architectures that have persistent caches. Where precise ordering is needed only for concurrent accesses.

In conclusion, durable transactions for NVMM are an essential component of any persistent memory support library. Existing implementations do not rival with the tailored failure-atomicity of data structures. That being said, by going over durable data types and FASEs, we accumulated a fair amount of clues on performance critical implementation aspects.

In particular, our contribution provides FASEs (§4.6) with only failure-atomicity, isolation being easily attained in Java with the *synchronized* keyword. Our implementation consists in a simple redo log, but working at 256B (Optane's block size) granularity. It is inspired by Romulus, except we avoid allocating eagerly the dual (back) space for the in-flight data version, mainly not to sacrifice half of our max heap capacity. Further details are available in §4.6.

2.7.4 Checkpointing

Checkpointing, or epoch-based persistence, is an alternative general-purpose abstraction that also provides a form of failure-atomicity. Checkpoints are often used for fault tolerance in environments with long computations such as in HPC, or basically, in any other situations where re-running computations is undesirable for time, cost, or energy reasons (e.g., small devices).

As we remember from early persistent memory and persistent object systems (§2.3.2), checkpoints were largely employed as an even easier way of denoting consistent application states for programmers. Assumedly, only explicit calls to the checkpointing facility are required to identify consistent recovery points that delimit persist epoch boundaries.

At the NVMM era, checkpointing remains quite attractive. Being a form of buffered persistence, it soothes out the cost of persistence instructions. NVMM low latencies may in addition largely help reducing runtime overheads of checkpoints.

ResPCT [193] provides a general checkpointing approach for fault-tolerance of multi-threaded programs on NVMM. They adopt InCLL (*In-Cache-Line-Logging*) of Cohen et al. [97] to store both variables and undo information in the same cache line. In doing so, they can update values during normal operation without issuing flushes or fences. Implicit flushes will not re-order undo information, since they are stored on the same cache line. ResPCT checkpoint period can be adjusted for performance. A 16ms period has negligible overhead versus a fully volatile execution with their hash map benchmark. With a 4ms period, throughput is still better than other generic other evaluated competing approaches. ResPCT performance are tied to its ability to allocate variables together with undo and epoch tracking information on the same cache line. However, it does not elaborate on how NVMM accesses are interposed, nor on whether its original memory layout for data object needs to be implemented manually. Moreover, with its specific memory layout, ResPCT at least doubles the NVMM footprint, just like Romulus [102].

Aside from general-purpose failure-atomicity, checkpointing on NVMM is also used to pursue *full-system persistence*. **Capri** [179] envisions an architecture/compiler co-design where their compiler automatically partitions persist epochs in programs. Similarly, OS-level persistence or single-level stores for NVMM, as in **LightPC** [207], **Aurora** [306] and **Twizzler** [65], use a form of checkpointing for transparent *whole thread state persistence*. We need not to elaborate again (§2.3.2.1) onto why this model is not the one we prefer to compose persistent programs.

Finally, **Arthas** [90] is a checkpointing mechanism for NVMM that covers up for hard faults. We already made the point several times that with direct access, transients bugs in software could potentially lead to permanent data corruption (hard faults) in NVMM. Then, in the manner of file system snapshots that enable recovering anterior images of the data, Arthas employs checkpointing for fine-grained versioning of NVMM states, such that the problematic ones could be rolled back.

Surprisingly, NVMM checkpointing for general-purpose failure-atomicity feels under-explored, in comparison to durable data structures or transactions. Though, forms of periodic persistence or checkpointing are found in the implementation of some *buffered* data structures or FASEs.

2.7.5 Memory and heap management

Correct memory management of a crash-consistent heap is the foundation of any persistent program. Evocatively, every single programming abstraction we discussed until now also requires a memory allocator that tolerates failures. Intuitively speaking, persistent variables are expected to remain reachable across multiple program executions, which entails that heap metadata stay sound as well.

Conventional memory allocators are optimized for fast allocation and reduced heap fragmentation, but are unable to maintain consistent states across potentially faulty executions. In addition to enforcing crash-consistency of their own metadata, allocators need to consider the effects of possible crashes while they hand-off memory responsibility to application code. For instance, a fault occurring before the application could durably reference newly allocated data would result in a persistent memory leak.

We have already identified issues tied to persistent heap management when we went over early persistent object systems in §2.3.2.2. NVMM-specific persistent heap management and allocators are also faced with the same ones, that we then recall here:

- **Dangling references.** Generally speaking, persistent references need to be updated atomically to avoid memory leaks or dangling pointers. This remains true when dynamically allocated memory is handed-over or reclaimed by the allocator.
- **Persistent pointers.** In modern OSes, a process' address space is only ephemeral. Pointer values stored from anterior executions are then garbage, unless the heap could be re-mapped at the same starting address. The generally accepted solutions are to *swizzle* pointers or, to store *relative offsets* rather than *virtual addresses*.
- **Referential integrity.** All pointers must stay valid at any time. In particular, persisting addresses to volatile objects will result in "wild" pointers in subsequent executions. Additionally, using relative offsets for durable references will collide when dealing with multiple persistent heaps.

Proper durable heap management and memory allocation are then expected to endow applications with sufficient facilities that prevent any of the aforementioned situation and the associated persistent bugs to occur. We now propose to present how these topics are tackled by present NVMM-specific persistent object systems, heap managers and allocators. For that, we will exemplify the most common techniques, by referring to the following work. **PMDK** [19], a production-grade persistent memory programming toolkit backed by Intel, and the de-facto library used by newcomers to NVMM programming. **Mnemosyne** [310] and **NV-Heaps** [92], two (relatively older) complete system for NVMM programming, designed even before the Optane craze. **NVAlloc** [108], **Makalu** [63] and **Ralloc** [76], crash-consistent allocators that are either log-based or use a form of garbage collection.

2.7.5.1 Memory leaks & Dangling pointers

Persistent memory allocators obviously need to remain *internally consistent* across potential failures. Otherwise, bugs such as double frees may occur in executions. More importantly, the user interface for memory allocation must also maintain a continuity between the allocator and the user code. Simply put, it must provide atomicity of (de)allocation and reference (de)installment in the user code. We identify the following three distinct semantics that prevent dangling pointers: *(i)* transactional allocation, *(ii) malloc-to*/*free-from* calls, and *(iii)* post-crash garbage collection.

(Transactional allocation.) In persistent programming models with support for FASEs, dynamic persistent memory allocations and reclamations can be enclosed within transaction boundaries. So long as the (de)allocation operation and the pointer update are part of the same atomic section, the underlying logging schemes guarantees no leaks or dangling pointers may result of failures. These semantics are supported in the PMDK, and tend to be more natural for programmers. However, this mechanism is expensive: allocator metadata are tiny, and small writes do not play best with NVMM; also the PMDK, which relies on undo logs, incurs six flush operations for one alloc-free pair [199].

(Malloc-to/free-from.) By extending the traditional allocation API to additionally supply a destination or origin pointer, the allocator can then assume the responsibility of atomically updating the memory attachment's point. This API was already supported in Mnemosyne then largely re-used with the PMDK; and was assumed even in latest academic work such as NVAlloc. Note that the underlying mechanisms for atomicity of allocation metadata and pointer update is still log-based in these systems.

(Post-crash garbage-collection.) Ralloc and Makalu are built around a garbage-collection (GC) phase on recovery, and can thus use the standard malloc()/free() API with no code modification. This offline GC pass enables them to reclaim unreachable memory block by traversing from the user persistent root objects. Those persistent roots provide tracing GC algorithm an obvious starting point. Conveniently, root objects are demanded by the programming model of persistent memory, to get users a fixed-point to recover from. Notice that this is similar to *transitive persistence* from §2.3.2.2 and §2.3.2.3, except that memory objects are not automatically migrated to NVMM here. Another attractive point, is that allocation metadata can be reconstructed on recovery. Thanks to GC and offline metadata reconstruction, Makalu and Ralloc pay almost no persistence cost and rival with transient memory allocators under normal (crash-free) operation. In a broader way, automatic memory management schemes are reassuring for programmers. Even more so when dealing with persistent memory, and the amplified consequences of its associated memory bugs.

Recovery time garbage collection is almost a clear winner here. Mainly for the (almost) null cost during normal operation and for simplifying the programming model. Although recovery may seem expensive for large heaps, remember that the procedure consists of a massively parallel read-only (long) tracing phase, followed by a (shorter) write-only phase for metadata re-population. Both have favorable NVMM access patterns, as we explained in §2.7.1 and §2.7.2.

2.7.5.2 Persistent pointers & Referential integrity

Preserving referential integrity is nonnegotiable: pointers must remain valid, even across consecutive faulty executions. With ephemeral virtual address spaces, persisting raw pointers will most likely lead to data loss. Likewise, persisting pointers to volatile objects will result in dangerous "wild" pointers in subsequent executions.

(References to volatile heap.) These kind of pointers can prove to be convenient when coding, but are totally unsafe. Most systems deem them illegal and forbid them. As in the PMDK and NV-Heaps, where they are statically prohibited using dedicated persistent types. Other systems (e.g., Espresso [329]) simply set them to *null* on recovery, to at least, raise an exception on next accesses in subsequent program runs, and avoid failing silently.

(References to persistent heap.) Storing raw pointers is possible, since the heap may be mmap-ed at the same starting address, but is unreliable generally speaking.

Swizzled pointers are safe to store, but costlier to de-reference. In Go-PMEM [143], raw pointers are stored. If the heap were to be mapped at a different starting address, all pointers would be re-originated on recovery. For that, they store along with heap metadata the previous heap starting address. Expectedly, re-writing all pointers comes at a high cost and this is not a common solution. More frequent, is the *offset-based pointer representation*, as found in Ralloc and NVAlloc. A simple arithmetic addition is needed when de-referencing a pointer.

(Cross-heap references.) When accessing multiple persistent heaps, as permitted in NV-Heaps and the PMDK, relative offsets will collide. Conceptually speaking, cross-heap references are only legal when the two heaps are loaded. That is the reason why such references are banned in NV-Heaps and the PMDK, with dynamic checking. Both systems use *fat pointers*, i.e. composite software structs that combine a heap identifier with an offset-based pointer. Additionally, an extra mapping table is necessary to map heap identifiers to their starting addresses. Needless to say that the de-referencing of fat pointers is a high cost operation.

In the end, referential integrity is crucial, with yet simple solutions. We note nonetheless that, this is the case only when opting for a model with dedicated persistent types. With implicit types, (costly) dynamic instrumentation is likely to be required, as with PS-Algol, PJama, or any other orthogonally persistent languages (§2.2.4). That also includes AutoPersist [288], a persistent flavor of Java for NVMM, that we review later in §2.8.4.2.

For reference, in our contribution, as motivated in chapter 3, we use in conjunction *(i)* recovery-time GC, as in Makalu [63], *(ii) offset-based pointer representation*, *(iii)* and dedicated persistent types to cover for unsafe volatile or cross-heap references.

2.7.6 Summary

To conclude, abstractions for failure-atomicity help break down the intrinsic complexity of the NVMM programming model. We now understand why any programming support for NVMM may want to include them. Through this section, we introduced, first, off-the-shelf durable data structures, either hand-tuned for performance (§2.7.1), or mechanically constructed for coverage and convenience (§2.7.2). Then, general-purpose FASEs as, basically, a sledgehammer to solving all NVMM programming challenges. Either in the form of durable memory transactions (§2.7.3), or epoch-delimiting checkpoints (§2.7.4). Finally, suitable techniques for NVMM heap management and data referential integrity (§2.7.5), which are essential building blocks towards any of the previous.

Among all the guidance and specific insights for performance and efficiency we gathered, two main themes seemed to emerge across the board. On one side, NVMM-native constructions and log-based approaches that seek out algorithms for lower cost of NVMM accesses and persistence overheads; with the underlying goal of benefiting from instantaneous recovery by staying consistent at any instant. On the other, algorithms that take advantage of hybrid memory architectures and leverage DRAM to eliminate most of the need for logging, crash-consistency and specific access patterns; at the expense of longer recovery time to reconstruct soft data. For this reason, including a recovery pass at the library level that orchestrates object-specific recovery code sounds like a thoughtful touch.

We must also mark, through these diverse solutions we encountered, comes out very clear that answers to NVMM programming challenges are very well explored as of now. Mature and reliable enough so that combining them all into a single cohesive program-

ming model seems like the only last thing left out to do. In fact, this is mostly what we did, by basing our PDTs from [354], our FASEs from [102] and our heap management from [63]. With detail, we explain their implementation later in §4.5, §4.6 and §4.4.

Typical developers are waiting to harness NVMM and persistence at the language-level. To some extent, the PMDK offers a complete set of tools, but remains fairly low-level, meaning cumbersome and difficult to use. In fact, all work we mentioned until now were implemented in native languages: C or C++. Presently, higher-level of abstractions are clearly overlooked and object-oriented languages miss out on the tremendous benefits of persistent memory.

In an effort of democratizing persistence and reaching the whole host of potential use cases for NVMM we detailed in §2.4.1, our interests are piqued by potential simplifications of the programming model made possible in the object-oriented model and managed language runtimes. Our intent is to unveil essential properties for efficiency, genericity and applicability of a programming model for persistence atop NVMM in managed-language ecosystems. To this end, we present in the coming section (§2.8) the state of persistence in the Java language, along with existing NVMM-specific exploration (industry and academic) work.

2.8 NVMM and Java

In the previous section, we noted that most prior works on persistent memory target programming languages that compile to native (C, C++, etc). We argue that due to the wide application range of NVMM, covering managed languages and object-oriented programming is equally as important. In particular, Java has been seen as a solid candidate for persistent memory ever since its first release, as related in §2.2.4 with the PJama project from Atkinson's group [49]. Today, Java stands as a major player in the big data ecosystem. Many modern data stores, data analytics or processing framework are written in Java [64, 70, 118, 204, 229, 239, 275, 318, 342, 343]. Unfortunately, to date, accessing efficiently NVMM in Java remains an open challenge.

Nonetheless, some recent academic research [143, 288, 329] , and industrial efforts [16, 20, 157, 299], offer solutions for managed languages. From the insight we gathered in previous sections, we propose to show how these recent solutions fail to address all core challenges. We could frame requirements for practicality, applicability and usability of persistent memory from early designs (§2.2, §2.3). Whereas programming abstractions for failure-atomicity (§2.4, §2.6, §2.7) got us guidance towards achieving balance between genericity and efficiency on NVMM. Consequently, the core challenges we then identify for language support of persistent memory are:

- **(User) Separation between transient and persistent data.**
- **(User) Identification of consistent program states.**
- **(System) Easily integrate in existing applications.**
- **(System) Prevent dangerous persistent to volatile references.**
- **(System) Avoid dangling references or memory leaks on recovery.**
- **(System) Lightweight management that do not restrict NVMM characteristics (bandwidth, access latency, heap size).**

- **(System) Offer tuned FASEs & PDTs, do not limit users in implementing their own.**

- **(System) Allow recovery-time code to clean inconsistencies in both system and user (meta)data.**

We then organize this section as follows: (§2.8.1) We remind the specificities of managed language environments, with an emphasis on garbage collection in Java. (§2.8.2) We present the state of persistence in Java, unrelated to PMEM. (§2.8.3) We relate recent *external* solutions to NVMM persistence in Java. (§2.8.4) We examine *internal* approaches that bring persistent memory in Java.

2.8.1 Managed languages & Garbage collection in Java

Briefly, we remind that Java does not compile to native code, and therefore can not easily benefit from all prior work we mentioned. Instead, Java compiles in architecture agnostic *bytecodes*[6]that are then interpreted by a Java *virtual machine*, its language execution runtime. Another key point of the Java runtime, is that it offers automatic memory management through garbage collection.

Execution environment. Loads of Java virtual machines (JVM) exist, on various platforms. Commodity servers usually run the Hotspot JVM (OpenJDK), a reference open-source implementation initially developed by Sun Microsystems. In particular, this implementation has no direct programming support for persistence instructions (`CLWB` or `SFENCE`) of recent Intel architectures. Along with Hotspot, modern JVMs do not simply interpret Java bytecodes, but embed a *just-in-time* compiler to output native code on-the-fly, in order to re-optimize frequently executed code. With details, Hotspot VM profiles bytecodes' execution, and use a tiered compilation mechanism to re-compile hot code first. Its *C1* compiler quickly outputs native code with a low degree of optimization, and instrumentation for online profiling. Its *C2* compiler outputs fully optimized native code, including dynamic optimizations. Adding new machine instructions in Hotspot is tedious for this very reason. One has to ensure that they are retained and executed in all three code representations. Above all, that they are properly inlined in the assembly generated by the C2 compiler. JEP[7]-352 [124] introduced a composite abstract instruction to flush and fence a memory range. Meaning that since 2020, from OpenJDK 14 and onward, persistence instructions (`pwb`, `pfence`, `psync`) are internally supported and properly compiled, but not externally exposed. First and foremost, we had to take on the task of backporting their, at the time, experimental patches to OpenJDK 8 and externally expose these individual instructions, as we report in §4.4.8.1.

Garbage collection in Java. The second topic on which Java differs from native languages, is memory management. Java users need not to allocate, free memory, or even manipulate pointers. Memory objects are transparently allocated when created, or freed when no longer referenced. In Java, the garbage collector (GC) is a JVM service that performs all heap management routines. As a consequence, blindly enhancing Hotspot VM for NVMM calls for a crash-consistent GC. Recoverable GCs were proposed nearly

[6] An abstract assembly language for the Java virtual machine.
[7] Java Enhancement Proposal

30 years ago [135, 201, 251], for the purpose of orthogonally persistent Java (§2.2.4) or persistent distributed shared memory (§2.3.2.3), but nothing at the NVMM era.

G1 [121] is a modern GC and the default in Hotspot VM, it has: *(i)* a generational heap, does *(ii)* concurrent marking, and *(iii)* parallel compaction. In detail, G1 orchestrates its collection phases around a cycle, and partitions its heap in multiple generations. G1 traces object reachability concurrently, but still need short pauses to promote (copy) them from one generational space to the other or to compact each generation. Objects that survive multiple collections are gradually promoted from the "eden" space to the larger "survivor" space that is less frequently inspected. G1 bounds the application pause time during collection phases by reclaiming regions with the lowest liveness first (fewer incoming pointers). Running frequent, generation-based small passes, ideally avoids G1 running expensive stop-the-world full-gc phases.

Turning such complex GC algorithm crash-consistent has yet to be done. Work that extended the Java heap for NVMM wittily avoided doing it, by favoring alternative GC algorithms (e.g., Espresso [329]), or by building on research JVM that are easier to tinker with (e.g., AutoPersist [288], JaphaVM [259]). Furthermore, nothing guarantees this conversion would be efficient. Evidences even point the opposite. In [340], the author establish that G1 constitutes a performance bottleneck when running atop Optane NV-DIMMs (in volatile mode, without persistent instructions) [192]. Their experiments point towards improper NVMM bandwidth usage to be the cause. Namely random small-sized mixed read-write accesses. They propose to rework G1 compaction algorithm with favorable NVMM access patterns in mind for transient executions (large memory use case). For that, they re-arrange small mixed accesses into distinct read-mostly and write-mostly phases.

ZGC [336] is a newer GC algorithm, declared production-ready since OpenJDK 15 (2020). It is designed with larger heap sizes in mind (up to 16TB) and sub-millisecond pause times (even as the heap grows). No data is available on its performance on NV-DIMMs. All considered, with its 25,000 source lines of code, modifying ZGC for crash-consistency will be at least as hard as G1.

Garbage collection atop NVMM. Extending GC to efficiently run on Optane was more explored for the volatile use of NVMM in heterogeneous memory systems. G1 and many modern GC, do copy-based compaction, meaning they relocate objects to reduce heap fragmentation. These continuous object movements prevent OS-level memory tiering mechanisms from tracking objects and finding which ones are more frequently accessed, as we had mentioned in §2.5.1. Work then implemented tiering mechanism as GC extension inside JVMs to detect hot and cold objects. **Panthera** [311] is designed for energy savings in big data processing. The system detects cold objects through code analysis and leverages the GC to migrate them to NVMM. *Write-rationing* garbage collectors, as in [35, 36], tackle the write endurance problem. **Kingsguard** [35] redesigns the Java heap to include new *mature* object spaces in NVMM and DRAM, and **Crystal Gazer** [36] profiles past executions to predict highly-mutated object allocation code sites and read-mostly ones, to then allocate them to *mature* NVMM or DRAM spaces accordingly. **PIMA** [34] allows to further reduce the DRAM footprint by trading-in write endurance for performance. It promotes *mature* objects successively to DRAM and NVMM, while sampling performance online. Then decides, based on collected profiling data, on placing them long-term in DRAM only when performance is improved. **GCMove** [215] very recently proposed to leverage a lightweight GC write barrier to dif-

ferentiate read-mostly objects. Their design splits the heap old-gen space in two parts, one residing in DRAM and the other in NVMM, with distinct memory management for NVMM.

2.8.2 Persistence in Java

Commonly, persistence in Java is achieved through specific interfaces, that *externalize* data out of the heap. Since heap management have ever been designed for traditional volatile memory, the role of these interfaces is, obviously, to read in and write out data, but also to translate between the in-memory Java object representation and the format of the persistency layer, that is, data marshaling. We first describe database object mappers (§2.8.2.1), then file system access (§2.8.2.2). After which we present additional interfaces when considering NVMM. That is, the one to call native code from Java (§2.8.2.3), and the one to do direct memory manipulations (§2.8.2.4).

2.8.2.1 Persistent object model

Object-relational mappers. We recall from §2.2.4, that Java included early-on in its specifications standardized support for external database connectors. In essence, to abstract away the "impedance mismatch" of database systems. The *Java Persistence API* (JPA) [120] and the *Java Data Object* (JDO) [182] both provide standardized bindings for object-relational mapping. In a nutshell, they translate objects into database entities and object operation into database queries. Hibernate [11] is a popular object-relational mapper and a notable JPA provider.

The Java Persistence API. The persistent object model of JPA is based on object-oriented idioms, to make persistence fit naturally within application code. In particular, any class can be persistent. Code annotations on the class, its attribute and methods, enable generating a mapping with the database. Retrieving "Entity" objects (annotated class) fetches and copies the data onto the Java heap and materializes it through a volatile representant object. Mutations are performed in volatile memory, by accessing the methods of the entity object. Database transactions can be orchestrated through the "EntityManager" instance. Entities are copied-out and pushed back to persistence on calls to the flush() method of the entity manager or on commit() of the on-going database transaction.

In all, JPA eradicates database queries from application code. Relational table rows are instantiated on-heap as plain Java objects. They can be manipulated without special attention, being a local copy of the stored data.

2.8.2.2 File system interface

File operation. Java posses file manipulation interfaces that matches with the common open/read/write/close and mmap system calls. In the JDK (*Java Development Kit*), a `FileChannel` is a channel connected to a file, that allows reading, writing or appending data. The data is supplied or retrieved in the form of a `ByteBuffer` instance that materializes a sequence of bytes. Alternatively, a file can be memory-mapped through its FileChannel. The resulting `MappedByteBuffer` specialized instance has an additional force() method. It flushes the changes made to the ByteBuffer onto the storage device containing the file.

Object serialization. The previous file manipulations are fairly low-level and cumbersome to use. Java also exposes higher-level facilities that read and write whole objects to files. To that end, Java has commodity classes (`ObjectInputStream` and `ObjectOutputStream`) that (un)marshal `Serializable` objects into byte sequences. In most cases, a class needs only to be "tagged" with the Serializable interface, without actually implementing any interface abstract methods. Object streams can be instantiated and connected to any FileChannel. The streams' `readObject()` and `writeObject()` methods make trivial persisting Java collections whole to files.

2.8.2.3 Native interfaces

Considering that most implementation efforts towards tools and libraries for NVMM programming focus on native languages, we briefly present the *Java Native Interface* (JNI) [249].

The Java Native Interface. JNI allows gluing C or C++ library code to Java applications. With it, from any Java class declaring `native` abstract methods, a C header file can be generated. Next, a C implementation of the header-declared functions must be manually provided. Usually, one bridges on existing C library code. After being compiled, the resulting native library file can be dynamically loaded in Java programs through the `System` class. At this point, calling any of the Java class `native` methods will then run the C code.

Limitations. JNI is yet notoriously known for several (bad) reasons. First, it is not easily integrated in Java build systems. Producing a self-contained jar artifact with both Java bytecodes and native dependencies can not be done elegantly. Additionally, native code is architecture dependent, meaning the produced jar artifact will no longer be portable. More substantially, JNI makes crossing the boundary between the Java world and native an expensive operation [156]. The JVM treats native code as a black-box. As such, `native` method calls do not benefit from compile-time code optimizations and can not be inlined. Equally as bad, the layer of communication between Java and native does (un)marshal any data exchanged. Accessing Java objects' data without memory copies is possible by calling object methods from native code. Invoking object methods necessitates to pass an object pointer through JNI, then a first lookup in a table to retrieve the function pointer of the object's method, and finally the actual function call. For all these reasons, JNI induces overheads that are prohibitive for small calls. Each JNI call can induce tens to hundreds-of-microsecond of additional delays.

Project Panama. Project Panama [5] is the next-generation native interface in OpenJDK. It is still experimental, but aims at remedying JNI shortcomings. Essential features include foreign memory access through a safe API, and foreign function calls. That is, direct memory operation to off-heap memory segments, and sharing of those segments with foreign functions. Presently, only the foreign memory API has reached an usable state. None of the two are deemed production ready yet.

2.8.2.4 Direct off-heap memory access

Off-heap for large memory. Accessing off-heap memory in Java have been an ancestral way of relieving the garbage collector. As the heap grows, garbage collection phases

become costlier [245]. Regardless of NVMM, several applications avoid running GC by storing part of their (large) datasets outside the Java heap. This is notably the case of modern data stores (e.g., Spark [343], Cassandra [204] or Infinispan [229]). Apache Arrow [1] addresses the problem of making such data structures portable. In [231], the authors propose **Oak**, an efficient volatile lock-free off-heap map. Oak allows modifying directly off-heap objects, that is, without copying data between the on- and off-heap spaces. Contrarily to our contribution, all of the above do store only marshalled objects off-heap (as arrays of bytes) for portability.

Direct ByteBuffer. Off-heap memory can be safely allocated and represented in Java by instantiating a `DirectByteBuffer`. It is a ByteBuffer implementation whose memory reside outside the garbage-collected heap. All of its on-heap and off-heap memory is reclaimed when the buffer instance is garbage collected. As any other ByteBuffer, Java provides only methods to read and write primitive types or byte arrays in the buffer. Consequently, storing Java objects in a direct byte buffer also necessitates marshalling and memory-copying them.

The Unsafe interface. A more permissive, but dangerous, interface exists in the JDK: `sun.misc.Unsafe`. This interface allows for low-level (and unsafe) memory manipulations, such as, manual (de)allocation of off-heap memory, or memory load/stores at arbitrary addresses (on- or off-heap alike). Because of the Java typing system, these manipulations are exposed as method calls in the form of `getCharAt(long addr)`, `putCharAt(long addr, char c)` and so on, for every primitive types. Unsafe also has methods to retrieve the virtual address of on-heap allocated Java objects, and an equivalent of `memcpy()`. Though, memory-copying Java instances is sketchy due to possible concurrent object relocation during GC pauses. All of these manipulations are directly inlined in the assembly generated by C2, similarly to the magic interface of JikesRVM [137].

In all, Unsafe is the only interface for direct-memory manipulation in Java. It can erase the dual data representation, and avoid wasting precious CPU cycles to convert or copy data back and forth between on- and off-heap.

Unsafe is the interface in which JEP-352 added its composite operation to flush whole memory ranges to NVMM. In spite of being extremely low-level, we also heavily rely on Unsafe in our contribution, as a building block for all off-heap memory management. Furthermore, we expose `pwb`, `pfence` and `psync` through Unsafe as well (see §4.4.8.1).

2.8.3 NVMM framework bindings for Java

In here, we examine *external* systems for NVMM persistence in Java. That is, those that place data and NVMM outside (off) of the Java heap. We notice that none of all work presented here is academic. Mashona revisits the Java file interface for NVMM (§2.8.3.1). PCJ and LLPL are Java bindings for the PMDK with two different programming models (§2.8.3.2, §2.8.3.3). MDS is a library of managed data structures for persistent memory with Java bindings (§2.8.3.4).

2.8.3.1 Mashona

Mashona [157], an experimental third-party solution from RedHat, offers NVMM-optimized `FileChannel` and `MappedByteBuffer` implementations. The library can open

a special file channel and memory-map a direct byte buffer from a file stored on a DAX file system. They are optimized, in the sense that, they both employ direct-access and user-space memory instructions instead of regular file system calls. The approach is similar to FLEX [334] (§2.5.2), it replaces in implementation read/write and fsync/msync system calls with `memcpy`, `CLWB` and `SFENCE`. Precisely, the Mashona channel and byte buffer use the flush-range operation from JEP-352 to force memcpy-ed data to the underlying NVMM DAX file.

(Applicability.) `FileChannel` and `ByteBuffer` alike in Mashona both strictly adhere to JDK specifications. They can as such be used as drop-in replacements in Java applications. Therefore, the ones that directly used a file system (which includes most of big data stores) can leverage NVMM with a single code line edit.

(Benefits.) Mashona is another "block device" approach to NVMM, that is, designed for compatibility of legacy applications. It does cleverly hide beneath the Java file system interface and is transparent to applications. Compared to NVMM-optimized file systems (§2.5.2), Mashona avoids the cost of system calls.

(Limitations.) Mashona is compatible with legacy applications, but very likely, those programs were optimized for rotational hard-drives. As a result, the file access patterns and protocols for crash-consistent updates are sub-optimal with NVMM, as seen in §2.2 and §2.3.2. More importantly, Mashona does not prevent the dual representation of data, nor marshaling operations. We show in the evaluation of our contribution (§5.3) the high impact of these software operations.

2.8.3.2 PCJ

The Persistent Collections for Java (PCJ) [16] is a library of persistent data types proposed by Intel and implemented with the PMDK. The programming model is similar to the PMDK. It differs though, by exposing collections (as in `java.util`) and boxing classes for primitive types, instead of C structs. Its interface is extendible with abstract classes that allow users to define their own persistent types. Under the hood, it uses JNI to call the PMDK and to manage its persistent heap. In particular, all updates to memory, for consistency, are encapsulated in PMDK transactions through JNI calls. As an optimization, non-transactional reads are serviced by Unsafe.

(Applicability.) PCJ data structures implement `java.util` interfaces for the most part. Theoretically, they can then be used in place of regular volatile Java collections. Except, things are not that simple. PCJ collections can only contain PCJ types (primitive boxes or user defined ones). Which means that applications have to: *(i)* identify which data types are durable, *(ii)* provide a PCJ-compatible implementation, and *(iii)* replace every occurrences in their code while making sure nothing else breaks.

With that put aside, the programming model is rather sound: crash-consistent data structures, PMDK FASEs, transactional allocations, and reference counting for memory reclamation.

(Benefits.) PCJ were the first to propose high-level data types and collections for NVMM in Java. Its programming model ticks all properties we established (§2.8), including explicit identification of persistent and transient data. It neglects recovery-time user code, but again, only FASEs are supported, hence it has always *null recovery*.

(Limitations.) A big shortcoming of PCJ is its reliance on JNI to manage its persistent heap. This alone contradicts all of our performance-related principles. We show throughout our evaluation (§5.2), that PCJ underperforms all across the board due to JNI. To the point where it is even slower than using a file system interface in

Java, with actual system calls.

2.8.3.3 LLPL

The Low-Level Persistence Library (LLPL) [20] is a low-level library for off-heap persistent memory access, by Intel and also implemented with the PMDK. Low-level, because unlike PCJ, it does not expose data structures but abstract memory blocks. Memory blocks are untyped and unstructured byte sequences that can contain arbitrary data and be linked together for structure. A purpose-built allocator provides crash-consistent management of the heap, that contains only memory blocks. Again, all updates are transactional (undo logging) as coordinated with the PMDK, through JNI.

(Applicability.) LLPL's low-level take on NVMM persistence sounds difficult and cumbersome. Truth is, the memory block abstraction is far easier to convert programs that used to rely on a file system. Memory blocks can host marshaled data, and be managed just like file chunks by programs. Intel actually released LLPL after PCJ. They did so for a port of Cassandra to NVMM [60, 61, 307], after realizing high-level data structures required too much heart surgery on the data store.

(Benefits.) Similarly to Mashona, LLPL is more suited as a compatibility layer for programs that used block storage. This time, with opportunities of revising application protocols for efficient NVMM patterns. Although using LLPL requires to almost entirely re-develop the storage management engine of a DBMS.

(Limitations.) For the exact same reasons as in PCJ, reliance on JNI is a deal breaker.

2.8.3.4 MDS

Managed Data Structures (MDS) [15, 299] were developped concurrently to PCJ by HP, and both are similar in essence. The library has cross-language support: C++, Python, and Java (through JNI). PCJ lacked in its ability to support existing volatile code. MDS provides class annotations to easily declare *Records* (data objects) and generate their memory layout on the persistent heap.

(Applicability.) MDS proposes an object and compute model separated from Java's, with strongly-typed persistent objects. For instance, records are specified by the user as an interface class, and instantiated with a static method. In fact, all record methods are static and attributes can only be accessed through their getters/setters. Their programming model is difficult to compose with regular Java objects, and unnatural for object-oriented programmers.

(Benefits.) Although MDS parts from regular Java, their point is to never bring any durable data on the Java heap. With that, they also aim at minimizing the number of JNI calls. Generic computations (lambdas) can be programmatically encapsulated in "Tasks", with or without isolation. Precisely, a whole execution graph of tasks can be built in the manner of Java's functional API. Then, only the origin arguments of the whole execution need to be communicated with JNI, along with a pointer to Java task objects.

(Limitations.) At least, MDS design avoids a lot more JNI overheads. They minimize the number of switches between Java and native, and reduce the amount of data transferred. All things considered, it severely parts from object-oriented idioms. Converting existing volatile Java code to persistence, or migrate disk management to NVMM, seems more than challenging with MDS.

2.8.4 Runtime-managed Persistence

We now examine *internal* systems for persistence in Java. That is, those that enhance the JVM for NVMM. Memory management is augmented for durability and recoverability of the Java heap. With little to no markings for persistence, the general theme is to support close-to-unmodified Java source code. All work presented here is part of academic research. Espresso introduces persistent allocation for Java objects (§2.8.4.1). AutoPersist brings orthogonal persistence back to Java with NVMM (§2.8.4.2). Go-PMEM, technically not for Java, but still incorporates persistence in a managed environment - the Go language (§2.8.4.3).

2.8.4.1 Espresso

Espresso [329] introduces the `pnew` operator to allocate persistent objects in Java. Their modified JVM, derived from Hotspot VM (JDK 8), is able to create transient or durable instances alike, simply by changing the `new` keyword into `pnew`. The Java heap is extended with a separate memory region on NVMM, treated as a non-generational space. The GC collects this persistent space as frequently as the "survivor" space, since both should only contain long-lived objects.

Relevant heap metadata are kept consistent across program restarts. Hotspot uses a bump pointer for fast allocations, they moved it to NVMM and issue a flush-fence pair on every `pnew`. For GC, they did not turn G1 crash-consistent, but the "parallel scavenge" GC (PSGC). Snapshots of the whole heap (data and metadata) are taken in NVMM right before *stop-the-world* compaction phases. A recovery mechanism detects crashes concurrent to heap compaction, and re-applies the last snapshot before retrying.

Programmers can use `getRoot()`/`setRoot()` new JDK calls to define persistent root objects. Persistent objects have extra `flush()` methods to persist whole or individual fields (inducing `pwb-pfence` pairs). Dangerous persistent to volatile references are not disallowed, but three safety criterion are described. User-defined, recovery zero-ing, and type-based safety. That last one is enforced by statically verifying (compile-time), that only references to JPA-like (durable) entities are held.

Espresso then also proposes a JPA-like API, named JPO, atop of the persistent heap abstraction. The goal is to re-use JPA annotations and calls to be compatible with traditional persistence in applications. For short, they heavily modified a JPA provider (Datanucleus [12]) to act as a local object store. FASEs are supported with JPA transactions API, though no implementation is described. Their design kind of breaks JPA semantics. JPA assumes that entities are local DRAM copies, and can be altered with no consequences until `entityManager.persist(entity)` or `transaction.commit()` are called. Whereas with NVMM, mutations are silently propagated. Programmers must then guard every single update with FASEs, a restriction absent in regular JPA.

(Applicability.) The programming model of Espresso allows any Java object to be persisted at allocation time. While a separation exists between volatile and persistent data, it is not possible to tell whether an object is persistent without tracking its origin point and allocation code site. We anticipate this to be error-prone and result in additional programming complexity.

Furthermore, constructing complex data structures with Espresso is not as simple as allocating a JDK HashMap with `pnew`. Rather, composite objects need to be manually edited. All their relevant attributes must also be allocated with `pnew`, with correct

manual insertion of `flush()` when mutated. Note that `pfence` are only available with `flush()` calls. A restriction that hinders the writing of efficient NVMM protocols with optimal flushes and fences. FASEs are also not accessible outside of the JPA-like entities and transaction interface.

Espresso is almost a no-abstraction approach, low-level, and semantically close to wielding bare `CLWB` and `SFENCE`; but with reduced malleability. Overall difficult to use, if possible at all. Related work [288] even report giving up on porting part of their experiments to the Espresso programming model: "we found it *much more difficult* to create a correct crash-consistent application in Espresso".

(Benefits.) Espresso is the first *internal* design for JVM-assisted persistence on NVMM. It successfully avoids dual data representation and marshaling.

Object types are not altered, meaning compatibility with legacy code is retained. That is, existing modules and libraries can work unmodified with persistent objects alike; so long they do not treat durability concerns. (*persistence independence* §2.2.4)

Heap management is crash-consistent, from allocation to reclamation. In particular, using a GC prevents dangling pointers and leaks. Cross-heap references are avoided, with either null-ing on recovery, or static persistent types. Though, nullifying on recovery could lead to unexpected null pointer exceptions.

Performance is hard to gauge, evaluations were performed on battery-backed DRAM (before Optane release), and the source code is unavailable.

(Limitations.) Espresso runs garbage collection over the persistent heap. Online GC pause times will inevitably restrict the maximum practical size of the NVMM region. Even more so since snapshots of the heap are made before each compaction phase.

GC is formidable for high-level object-oriented programming, but does not fully prevent memory leaks. GC approximates object liveness by reachability, meaning that failing to set a reference to null (coding error) will result in a permanent leak. These bugs are silent but can be detected with code analysis tools. However, fixing a corrupted persistent heap might not be so trivial.

Last, Espresso use native pointers inside the persistent heap. Therefore, it needs to retry mapping the heap until the OS accepts to use the exact same starting virtual address. If it fails repeatedly to do so, it gives up and re-originates all pointers from the previous base address to the new one. That is, Espresso traverses the whole heap and updates (then flushes) all pointers.

2.8.4.2 AutoPersist

AutoPersist [288] transparently migrates any Java object to NVMM as they become reachable from a persistent root, or back to the volatile realm when they no longer are. A persistent root is identified with a `@DurableRoot` annotation, hand-placed on a static object (a class-wide attribute). Scarce annotations and FASEs are the only code markings programmers need to add in their applications to support persistent memory with AutoPersist. The point is that all persistence-related operations are serviced by the JVM, with no programmer involvement.

Object migration is initiated by interposing on every `putField` Java bytecodes. The operation is augmented and first checks whether the object being referenced is already persistent. If not, then marks it as "converted" and recursively persists its whole transitive closure. Finally, it updates (and persists) the field.

Relocating objects also breaks existing inward references. They circumvent the issue with the same mechanism as in heap compaction. That is, *forwarding placeholder objects* that stand in the old location. When accessed, they force-update the reference of the caller to the new location. When they are no longer referenced, i.e., there are no more outstanding pointers to the old location, *forwarding objects* are garbage collected as any other object.

The persistent region of the heap is also garbage-collected. They extended GC to move back to DRAM objects that have become unreachable from the durable root. Objects are deallocated if not reachable at all.

For consistency, AutoPersist augments all non-transactional stores with `pwb` and `pfence`. Failure-atomic regions are also available through new JDK methods. Their implementation is a regular per-thread undo log.

AutoPersist is implemented in the Maxine JVM [322]. It is a research-oriented JVM, fully written in Java, which allows for faster and easier prototyping (compared to Hotspot VM), with reduced performance of course (up-to 50% slower).

(Applicability.) Individually denoting data that should reside in NVMM is no longer necessary. In truth, AutoPersist implements all three base principles of Atkinson's orthogonal persistence from PJama (§2.2.4): *(i) orthogonal persistence* — any type can be persisted, *(ii) transitive persistence* — the property of persistence is inherited by transitive reachability from a durable root, and *(iii) persistence independence* — bits of code handle transient or durable objects indistinguishably.

Although simple and seductive, this programming model seems to eagerly turn too much objects and data to NVMM. For that, they provide an additional `@Unrecoverable` annotation that denotes fields as *weak* references, not propagating persistence. Still, we question the relevance of the model. Real-world Java programs have tons of references, often circular. Understanding the structure of these humongous object graphs, in particular, their connected components is not easy. We anticipate that, in realistic cases, this model will not provide any coding simplification. Instead, it shifts positively marking persistent fields (Espresso-style), into negatively marking transient ones.

With Espresso, we argued that relying on GC could cause silent permanent leaks (programming bugs) when failing to unset a reference - back to null. In AutoPersist, misuse of the `@Unrecoverable` annotation will instead cause whole sub-graphs (connected components) of objects to leak silently. Put simply, dangerous references from persistent to volatile are not avoided, instead, volatile objects are made persistent to preserve referential integrity.

(Benefits.) AutoPersist is the first instance of fully-orthogonal persistence applied to NVMM. It avoids dangling references with GC, but it fails to satisfy every single other core properties we identified for language-level support of NVMM.

(Limitations.) For consistency, AutoPersist adds expensive flush-fence pairs after every single (non-transactional) store to NVMM. FASEs are implemented with undo logging, which duplicates every write and also adds expensive flush-fence pairs for every log entry (i.e., every store considering their logging granularity). Which means that in any instance, AutoPersist excessively adds flush-fence pairs to any memory store.

Persistence is managed transparently from within the JVM. To accomplish that, AutoPersist must interpose on every single Java bytecodes that incurs memory load or

store. Dynamically instrumenting previously inlined memory operations with complex Java methods comes at a price. They wrote a sibling paper [289] that explains compiler optimizations to turn that cost down. For short, without the optimization, their modified JVM ran 50% slower on test benchmarks with neither NVMM nor persistence - a fully transient execution on DRAM. With the optimization, the overhead is reduced to 10%. The optimization consists in verifying instance persistence firsthand, on every instrumented bytecode, with a single check (branching instruction). In most instances, the volatile implementation can be predicted right and inlined. Note that the fact this optimization works well, entails that code sites tend to mostly execute only on volatile or persistent data. Therefore, Atkinson's *persistence independence* is somewhat not reflected in the architecture of actual programs. Performance-wise, that means a fully volatile execution, which does not use FASEs or persistence annotations, is still 10% slower on their JVM.

Lastly, implementing AutoPersist in mainstream JVM such as Hotspot is far too ambitious. All of AutoPersist's complex algorithms directly augment Java bytecodes. Bringing support implementations throughout Hotspot tiered compilation is a severe programmer's trial. PJama [49] (§2.2.4) also was too complex to be actively maintained with the fast-evolving Java specifications; and ended-up being abandoned.

For reference purposes, AutoPersist is not the only orthogonally persistent Java for NVMM. **JaphaVM** [259] and **GCPersist** [330] are alternative research proposals. In particular, GCPersist provides a form of buffered durability by migrating transitively-reachable objects to NVMM during GC cycles.

2.8.4.3 Go-PMEM

Go-PMEM [143], is not for Java, but remains an effort to support NVMM natively in an object-oriented managed language. Go-PMEM is spiritually close to Espresso. Their only modifications to the Go programming model are: a `pnew` operator, and support for FASEs (undo logging). Crash-consistency outside of FASEs can be obtained by calling a new `persistRange(addr, size)` function from the Go runtime. Persistent pointers are swizzled on recovery as in Espresso, when the heap can not be mapped at the same virtual address. Volatile pointers can be stored in durable objects, but the swizzling pass zeroes out any of them on recovery.

(Applicability.) The programming model of Go-PMEM is almost identical to Espresso, perhaps with more malleable flush/fence primitives. It also does not use dedicated persistent types for compatibility. The authors admitted that: "as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory".

(Benefits.) This is the only NVMM-native interface for Go. The implementation is publicly available, which was not the case for Espresso or AutoPersist.

(Limitations.) As in Espresso, it does not support composite objects natively with `pnew`. This is even more obvious in Go-PMEM, where trying to `pnew` a builtin hash map will just fail and raise an exception.

We also noticed an implementation issue with GC and the persistent heap. Go-PMEM does not decouple GC triggers on the volatile heap from the persistent heap.

Meaning that a larger persistent heap (as is commonly the case) can postpone GC, to the point where garbage critically accumulate in DRAM, to eventually cause out-of-memory errors and an application crash.

2.8.5 Summary

In this section we presented the challenges tied to exposing NVMM in high-level environments such as Java; and articulated deficiencies of existing approaches found in academic and industrial designs alike. (§2.8.1) We reminded the specificities of managed programming languages, especially garbage collection. (§2.8.2) We exposed traditional APIs for persistence in Java. They all place durable data outside of the Java heap and build on object-oriented idioms to ease interactions with data.

(§2.8.3) We examined existing NVMM solutions that places data off-heap. They still can reuse object-oriented abstractions to successfully hide NVMM management. Their shortcomings mainly come from relying on native libraries (e.g., PMDK) for off-heap data and metadata management, which incurs impracticable JNI overheads.

(§2.8.4) We presented academic proposals that take advantage of NVMM to expose persistence as language extensions. The main interest is to offer identical programability opportunities for the two kinds of data. Hence, NVMM is directly managed by their (heavily-modified) Java runtime, with minimal additions to the language's standard libraries. All for supporting the full set of features offered by the programming language on persistent data. Let's just ignore for now performance limitations of assuring identical but crash-consistent heap management for NVMM objects; as we come back to the matter in §3.2.1 with experimentally-backed evidences. That aside, all three work have deficiencies in their programming model that stem from, roughly speaking, whole or part of persistence-related information being implicit, or hidden from programmers when they simply just stare at the code. Furthermore, we argued that persistent allocation alone (Espresso, Go-PMEM) was insufficient to build data types and make persistent programming high-level on Optane. Likewise, transitive persistence exacerbates consequences of software bugs, that is, persisting unnecessary (transitively-reachable) data when linking objects to a persistent root.

Overall, the only true differentiating factor between *external* and *internal* existing solutions, is that only the latter provides Java-native direct access to NVMM and totally avoids marshalling. As things stand right now, an ideal solution would be one that ticks all core properties we formulated at the beginning of this section, while combining off-heap NVMM with direct access through Unsafe, and hiding persistent programming complexity behind object-oriented idioms. In a nutshell, this is what we did with our contribution — that of the programming model is detailed in **??**.

2.9 Conclusion

With that last section, we finally could identify shortcomings in existing approaches to NVMM persistence in Java, and formulate the precise outline of our answer; meaning that we are now ready to bring this chapter to a conclusion.

(§2.1) We started out by defining persistence, as a property of data characterizing its lifetime. Pieces of data with the ability to outlive executions, to remain persistent for the extent of time during which they may be recalled and used by programs.

(§2.2) We reminisced historical ways of interacting with durable data. The time-attested success of file systems (§2.2.2) and database systems (§2.2.3), in spite of mis-

matched representations in main memory and storage. The attempt of single-level stores (§2.2.1) and persistent object systems (§2.2.4) at blurring the line between persistent and transient data. Abstracting persistence in systems continued being investigated, as a parallel line that could never overthrow file systems and databases.

(§2.3) As part of which was persistent memory; that we conceptually introduced as a programming abstraction. It strove at directing persistence through main-memory operations. We examined systems that implemented avatars of persistent memory, so as to understand what refrained them from becoming mainstream. Persistent operating systems (§2.3.2.1), object-oriented storage (§2.3.2.2), distributed programming languages (§2.3.2.3), and durable memory transactions (§2.3.2.4). All told us about the unattractiveness of transparent persistency, techniques for reference management in a persistent heap, or how software bugs are exacerbated with persistence. At all cost, we must avoid re-iterating conceptually broken designs from the past, with the advent of non-volatile main memory.

(§2.4) Actual NVMM is upon us. (§2.4.1) System software support is already laid for us to harness its direct byte-addressability. (§2.4.2) Commercially available Intel Optane modules, though, exhibit very different performance profiles and hardware properties when compared to regular DRAM. (§2.4.3) Optane and alternative NVMM technologies are poised to become central hardware components of next generation computing platforms.

(§2.5) The anticipated use cases for NVMM are plenty. They include building giant volatile memory systems (§2.5.1), accelerating file storage (§2.5.2), recovering databases in seconds (§2.5.3), or exploiting fine-grained persistence to bring fault-tolerance in a wide range of computing platforms (GPU kernels, on-device network caches, or even energy-harvesting servers) (§2.5.4).

(§2.6) NVMM however do not strictly provide a hardware-based support for persistent memory. Challenges for persistence with byte-grain direct access arise from current computer architectures, in particular, their volatile CPU caches and relaxed memory consistency models (§2.6.1). (§2.6.2) The extended memory model for Optane requires persistent stores to be explicitly marked and ordered with `CLWB` and `SFENCE` instructions. The resulting programming model is too low-level and error-prone. (§2.6.3) Even with persistent CPU caches, software will have to carefully enforce memory order for consistency. (§2.6.4) Software-based failure-atomic regions of code could provide an all-encompassing and easy-to-apply solution to the difficult programming model. However, typical protocols known in database systems for generic failure-atomic updates, when ported on NVMM, are somewhat impeding latency and bandwidth of the media.

(§2.7) Abstractions tailored for NVMM persistence then appear as a remedy to programming complexity and crash-consistency. Just as with concurrency, persistence has been granted a host of data structures (§2.7.1) and universal constructions (§2.7.2). Failure-atomic sections (FASEs) also flourished as renewed transactional memory (§2.7.3) or checkpointing (§2.7.4), tuned for NVMM specifically. All these well optimized techniques mean nothing if corruption spreads in the persistent heap after faults. (§2.7.5) Crash-consistent memory allocation and heap management are thus paramount to guarantee all pointers remain valid and no memory location leaks.

(§2.8) Managed language, as Java, are widely employed in the design of modern data stores and big data processing frameworks. (§2.8.1) Injecting NVMM in these systems then necessitates a gateway to persistent memory in Java. (§2.8.2) However, the state of NVMM persistence in managed environment is severely lacking. Nearly all of industrial or academic work support only native languages. The handful of them dedicated to

Java, either, *(i)* manage NVMM as *external* memory (§2.8.3) and rely on inefficient native interface calls, or *(ii)* manage NVMM as *internal* memory (§2.8.4) and end-up exposing a deficient programming model for persistence with largely inefficient NVMM handling (persistence instruction overuse or bad access patterns in heap management).

The contribution we present in this thesis then aspires at designing a proper Java-native solution to accessing NVMM. One that does not induce any overhead when reaching persistent memory locations. One that knows no limitation tied to NVMM large capacities and heap management. One with minimal performance impact in crash-free executions. One that is safe and empowering for programmers, that leaves them in-control, while it abstracts the complexity of failure-atomicity with object-oriented idioms. We present this contribution in the next chapters; (§3.1) starting with its programming model and experimental evidences that back our choices.

Finally, we close this chapter with Table 2.4. It displays a synoptic view of NVMM programming systems and abstractions we examined so far. It compares them in relationship to the core properties for language-level support we extracted throughout this chapter.

	Support		Failure-atomicity			Applicability			Efficiency					
	Model type	Language	Impl.	Granu.	Persistency	Static types	FASE	PDT	Dual rep.	Marsh.	NVM perf.	Heap mgmt	Heap size	Recovery
***File systems* (§2.5.2)**														
Ext4-DAX [2]	file system	any	user	page	imm.	no	no	no	yes	yes	− − −	N/A	N/A	− − −
NOVA [332]	file system	any	CoW	page	imm./buff.	no	no	no	yes	yes	−−	N/A	N/A	− − −
Mashona [157]	library	Java (native)	append	word	imm.	no	no	no	yes	yes	−	N/A	N/A	− − −
***Persistent data type* (§2.7.1)**														
SOFT [354]	data type	C++	log-free	op.	d. lin.	yes	no	yes	no	no	+ + +	+ + +	+ + +	++
***General-purpose constructions* (§2.7.2)**														
Capsules [57]	method	C++	dual-copy+CAS	word	detect. rec.	no	no	user	yes	no	++	N/A	N/A	+ + +
RECIPE [208]	method	C++	log-free	op.	imm.	no	no	user	no	no	++	N/A	N/A	+ + +
PMwCAS [315]	method+library	C++	log-free	word	imm.+linea.	no	no	user	no	no	++	N/A	N/A	+ + +
MOD [159]	method+data types	C++	shadow pag.+CAS	op.	imm.	no	no	yes	no	no	++	N/A	N/A	+ + +
NVTraverse [139]	method+library	C++	log-free	op.	d. lin.	no	no	user	no	no	++	N/A	N/A	+ + +
PRONTO [232]	library	C++	sem. log+chkpt.	op.	d. lin.	no	yes	no	yes	no	++	N/A	N/A	−
CX-PUC [103]	library	C++	$2N$ replicas+CAS	object	d. lin.	yes	yes	no	no	no	+	++	++	+ + +
Montage [319]	library	C++	CoW+epoch reclam.	payload[1]	buff. d. lin.	yes	yes	no	no	no	+ + +	+ + +	+ + +	++
Mirror [140]	library	C++	log-free atomics	word	d. lin.	yes	no	yes	yes	no	+ + +	+ + +	+ + +	+
PREP-UC [93]	library	C++	2 replicas+ sem. log	cache[2]	(buff.) d. lin.	yes	yes	no	yes	no	++	+ + +	++	++
Tracking [52]	method	C++	log-free	op.	detect. rec.	no	no	user	no	no	++	N/A	N/A	+ + +
PCOMB [130]	method	unspe.	CoW+CAS	object	detect. rec.	no	no	user	no	no	+ + +	++	+ + +	+ + +
ResPCT [193]	library	C	InCLL(undo+epoID)	epoch	buff. d. lin.	yes	yes	no	no	no	+ + ++	++	++	+ + +
***Persistent transactional memory* (§2.7.3)**														
Romulus [102]	library	C++	dual-copy+vola. redo	word	d. lin.	yes	yes	no	no	no	+	+ + +	++	−−
OneFile [267]	library	C++	redo	word	d. lin.	yes	yes	no	no	no	+	++	+ + +	+ + +
Redo-PTM [103]	library	C++	$N+1$ replicas+CAS	word	d. lin.	yes	yes	no	no	no	+	++	++	+ + +
Crafty [142]	library	C	undo	word	buff.	no	yes	no	no	no	N/A	++	+ + +	+ + +
***Persistent heap managers* (§2.7.5, §2.8.3, §2.8.4)**														
Mnemosyne [310]	library+compiler	C or C++	redo	word	imm./buff.	no	yes	no	no	no	+	++	+ + +	+ + +
Atlas [83]	library+compiler	C	undo	word	buff.	no	yes	no	no	no	+	++	+ + +	+ + +
PMDK [19]	library	C & C++	undo	object	imm.	yes	yes	no	no	no	+	+	+ + +	+ + +
NVthreads [168]	library	C	CoW+redo	page	buff.	no	yes	no	no	no	N/A	++	+ + +	++
NV-heaps [92]	library	C++	undo	object	d. lin.	yes	yes	no	no	no	N/A	++	+ + +	+ + +
Makalu [63]	library (allocator)	C	undo	16B	imm.	no	no	no	no	no	+	++	+ + +	++
Ralloc [76]	library (allocator)	C++	user	8B	recoverable	no	no	no	no	no	+ + +	+ + +	+ + +	++
PCJ [16]	library	Java (JNI)	undo	object	imm.	yes	yes	yes	yes	yes	− − −	− − −	+ + +	+ + +
LLPL [20]	library	Java (JNI)	undo	object	imm.	yes	yes	no	yes	yes	− − −	− − −	+ + +	+ + +
MDS [299]	library	Java (JNI)	none	cache[2]	flush-on-fail	yes	yes	no	no	yes	−−	++	++	+ + +
Espresso [329]	compiler+runtime	Java (native)	undo	word	imm.	no	yes	no	no	no	+	+	− − −	+
AutoPersist [288]	compiler+runtime	Java (native)	undo	word	imm.	no	yes	no	no	no	−	−	− − −	+
Go-PMEM [143]	compiler+runtime	Go	undo	word	imm.	no	yes	no	no	no	+	+	− − −	+
***This thesis* (§4.3, §4.5, §4.6)**														
J-NVM [210]	library	Java (native)	user	user	user	yes	yes	yes	no	no	+ + +	+ + +	+ + +	+ + +
J-PDT [210]	library	Java (native)	log-free	word	imm.	yes	yes	yes	no	no	+ + +	+ + +	+ + +	++
J-PFA [210]	library	Java (native)	redo	block	imm.	yes	yes	yes	no	no	++	+ + +	+ + +	+ + +

Table 2.4: Comparison of various NVMM programming support libraries and systems.

[1]payload: NVMM allocation payloads returned by the allocator.
[2]cache: the whole CPU cache hierarchy is invalidated and flushed to (persistent) memory.

Chapter 3

The Programming Model of J-NVM

This chapter presents the programming model of J-NVM and the rationale behind our design choices.

In what follows,

(§3.1) We briefly describe the key design aspects of J-NVM.

(§3.2) We present evidences that garbage collection is dispensable for data stores and hurtful on large persistent heaps. Current GC algorithms do not scale to the larger size datasets (§3.2.1) and there are too few object deletion sites in the code base (§3.2.2).

(§3.3) We detail the object persistence model of J-NVM and argument for explicit static persistent types.

(§3.4) We present how J-NVM goes about recovery to ensure liveness of persistent objects.

(§3.5) We finally demonstrate J-NVM usage with a basic example.

3.1 Overview

J-NVM decomposes a persistent object into a persistent data structure and a volatile proxy. Persistent data structures live in NVMM, outside the Java heap. Proxies are regular Java objects that intermediate access to the persistent data structures. They implement the `POBject` interface, are instantiated on-demand (e.g., when a persistent object is dereferenced) and managed by the Java runtime.

The above decoupling principle avoids running a garbage collector on persistent objects. Based on it, J-NVM implements a complete developer-friendly interface that offers failure-atomic blocks. To construct this interface, J-NVM reuses ideas and principles proposed in prior works, but assembles them differently. Our framework uses a class-centric programming model, that is the property of durability is attached to a class, and not to an instance. As common with prior frameworks (e.g., Thor [217]), a persistent object is live by reachability from a set of user-defined persistent roots. J-NVM garbage collects the unreachable persistent objects at recovery, but avoids running a GC at runtime for performance. Instead, objects are explicitly freed by the developer.

3.2 Memory management

Running a GC at runtime has a cost. For volatile objects, this cost is balanced by the usefulness of the GC: the GC avoids many bugs and simplifies substantially the code

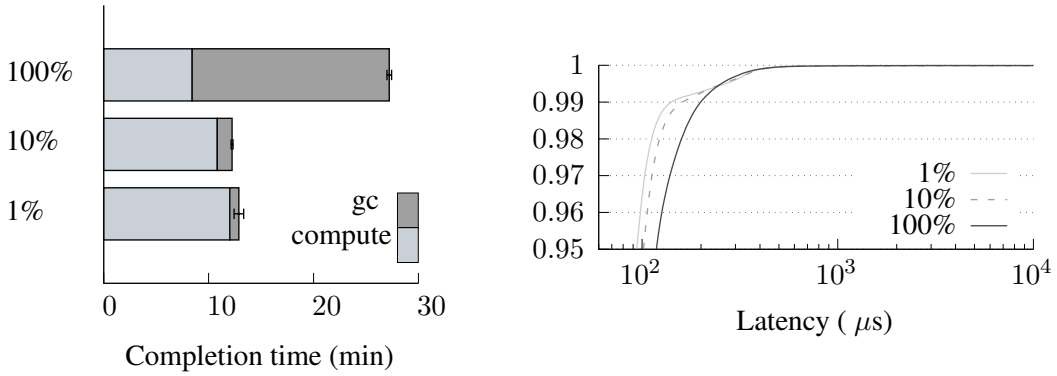


Figure 3.1: YCSB-F in Java with different cache ratios.

base. This section shows that this is no more the case with persistent objects in the context of a data store.

3.2.1 GC Overhead

First, we measure the cost of running a GC at the scale of a large persistent dataset. We consider G1, the default GC of Hotspot [121], then the tri-color concurrent marking algorithm of Go-PMEM [143], a recent persistent framework for the Go language.

G1. In this experiment, we evaluate the cost of collecting a large dataset with a state-of-the-art GC algorithm. We focus on the performance of a GC for volatile memory, because GCs for NVMM are not as optimized yet [143]. We evaluate G1, a modern GC that features many optimizations (generations, concurrent marking and parallel compaction). In particular, G1 pauses the application when it compacts the heap, but bounds the pause time by compacting the regions that contain more garbage first.

We evaluate G1 when running the YCSB-F workload with Infinispan. The experiment uses 15M persistent objects, which amounts for 15 GB of user data. Infinispan stores these objects in NVMM through the file system interface (DAX ext4). The workload is a mix of read and read-modify-write operations (50/50). In total, 10 threads executes 100M operations.[1]

To evaluate the performance of G1, we change the ratio of volatile cache in Infinispan. Infinispan uses this cache to avoid costly accesses to the file system. For each ratio, by testing different sizes, we configure the size of the Java heap for the best performance (20 GB, 30 GB and 100 GB for respectively 1%, 10% and 100%).[2]

Figure 3.1(left) reports the completion time of YCSB-F for the three cache ratios. With a cache of 100%, the completion time is roughly multiplied by two. When we analyze this result, we observe that the compute time alone becomes better when the cache ratio increases. This is expected: with a bigger cache, we decrease the number of accesses to the file system, which boosts performance. However, we also notice that, when every object is cached, 69% of the time is spent in GC, erasing the advantage of a larger cache (see breakdown in Figure 3.1(left)). This shows that, even with a modern optimized GC, collecting a large dataset comes at severe performance cost.

[1] The YCSB benchmark is fully detailed in §5.2.

[2] 100 GB may sound large compared to the 15 GB of user data. Yet, the volatile cache occupies 80 GB when it is populated with all data.

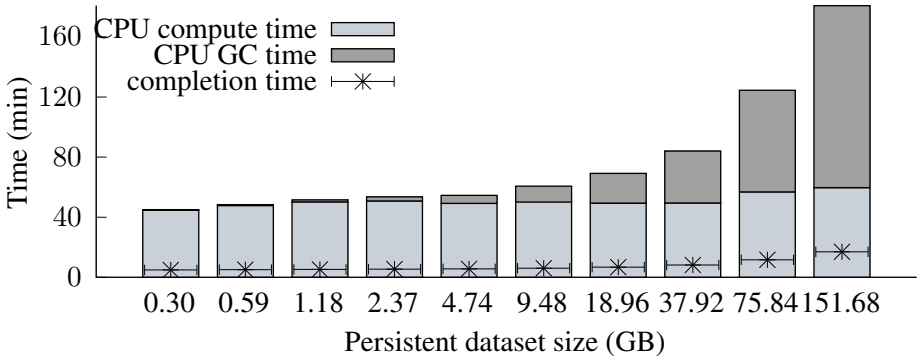


Figure 3.2: YCSB-F in Go-PMEM [143] when varying the size of the persistent dataset.

We verified that the overhead is caused by the size of the dataset and not by other phenomenons. Especially, with 100 GB, G1 has to use two NUMA nodes on our machine (see §5.1). G1 has also to use *uncompressed ordinary object pointers* (OOP) of 8 bytes while it can use *compressed* OOPs of 4 bytes with heaps up to 32 GB. For each ratio, we tested three NUMA configurations (interleaved or first-touch – through `numactl` – while the experiment presented in §3.2.1 uses the NUMA-aware G1 collector). For the 1% and 10% ratios, we also compared the performance with and without compressed references. The overhead does not come from these phenomenons: fluctuations between the configurations are insignificant.

Figure 3.1(right) indicates that a large heap also significantly harms performance stability. In this figure, we report the tail of the latency distribution for the YCSB operations. Above the 0.9999 percentile (10,000 operations), a small cache of 1% is 50x faster than the largest cache. This experimental result shows that managing a large heap can additionally incur rare yet impactful slowdowns.

Go-PMEM. The Go-PMEM framework [143] offers access to NVMM in Go following an integrated design. Its GC collects jointly the persistent and volatile heaps, using a tri-color concurrent marking algorithm [123] adapted for NVMM. The heap is not compacted. It is automatically resized after each collection. Because of a flaw in the resizing policy, applications may end with an out of memory error. To avoid this problem, we had to force a collection every 10 GB of allocation.

Figure 3.2 reports the performance of YCSB-F when using the go-redis-pmem data store. This data store is a feature-poor version of Redis [13] written by the authors of Go-PMEM. We execute the workload of Figure 3.1 and test different sizes for the persistent dataset. The black line in Figure 3.2 shows the completion time for each run. With a small dataset, YCSB-F lasts 5 min. The same experiment takes 3.4x more time (17 min) with a large dataset. To understand this drop of performance, Figure 3.2 shows the accumulated time spent by all the threads in GC (dark gray) and in compute (light gray).

In Figure 3.2, the compute time is relatively stable. This is expected since the exact same amount of operations is executed in each run. However, increasing the size of the persistent dataset also drastically increases the time spent in GC. With a small dataset, this time is negligible and accounts for less than 1% of the total CPU time. This shows that collecting the heap after 10 GB of allocated data is sufficient for the workload. With the largest dataset, despite that the number of operations remain the same across all

Data store	SLOC	# sites
infinispan (this paper)	603,800	4
cassandra-pmem [17]	334,300	1
pmem-rocksdb [7]	314,900	4
pmem-redis [6]	55,900	1
pmemkv [22]	25,600	2
go-redis-pmem [143]	8,400	2
pmse (MongoDB)[18]	4,800	3

Table 3.1: NVMM-ready data stores rarely delete persistent objects.

runs, the CPU time spent in GC reaches 67% of the total time.

This degradation comes from the fact that each GC pass visits all the persistent objects. As the size of the dataset increases, so does the cost of this computation. It would be possible to reduce the time spent in GC by adding more volatile memory to decrease the collection frequency. Unfortunately, this solution is not satisfactory because it just moves the cost of collecting NVMM from the CPU to the volatile memory.

3.2.2 Deletion sites

The previous experiment shows that collecting a large heap has a non-negligible cost. Paying this cost is only interesting if the GC actually helps the developer, either by simplifying the code or by avoiding bugs. Table 3.1 reports the number of deletion sites in several NVMM-ready data stores. We observe that even for data stores with a voluminous code base, a handful of deletion sites exists. This shows that garbage collecting persistent objects at runtime has a limited interest to ease programming.

From what precedes, we pragmatically consider that the performance penalty of garbage-collecting NVMM in a data store outweighs its benefits. This key observation motivates our design choice, where persistent objects live outside the Java heap.

3.3 Data persistence model

J-NVM exposes what we call a *class-centric* programming model. With this model, durability is a property attached to a class: in J-NVM, a class is persistent if and only if it implements the `POBject` interface.

On the contrary, AutoPersist [288] and Espresso [329] attach the durable property to each instance of a class. We call this approach the *instance-centric* model. Espresso relies on the `pnew` keyword to allocate an instance of a given class directly in persistent memory. Similarly to Thor [217], AutoPersist first allocates the object in volatile memory, and once it becomes referenced by another persistent object, it is moved to persistent memory.

The instance-centric model is appealing because the programmer can use the same class to instantiate a volatile or a persistent object, which is not the case with the class-centric model. As discussed below, the instance-centric model, however, raises two fundamental problems.

Reliability. By hiding durability from the type system, the instance-centric model provides a form of orthogonal persistence [48]. Hiding durability can be harmful because neither the developer nor the compiler can easily identify the persistent state of

the application. This problem is also underlined by George et al. [143]: "as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory". The developer can make mistakes by thinking that an object resides in persistent memory while it resides in volatile memory, which leads to data loss [183]. Conversely, the developer can also move to NVMM more objects than necessary, leading to memory leaks. Because these bugs only happen at runtime, identifying them is difficult [92, 220].

The class-centric model of J-NVM decreases transparency but makes the code clearer and thus less error-prone for the developer. The type of a variable (or a field) directly indicates whether it resides in persistent or volatile memory. The class-centric model trades the code simplicity of the instance-centric model for better reliability, exactly as a statically-typed language trades the code simplicity of a dynamically-typed language for better reliability.

Cross-heap references The instance-centric model raises a second problem related to the way it manage cross-heap references. Such a reference, from the persistent to the volatile heap can appear because the same class serves to allocate objects in both persistent and volatile memory. In particular, a reference stored in NVMM can indifferently refer to a persistent or a volatile object. When a crash occurs, because the volatile heap is emptied, a cross-reference from persistent to volatile memory becomes dangling, that is referring to an invalid location.

AutoPersist avoids cross-references by instrumenting the code. When the application writes a reference to `b` in an object `a`, if `a` is in NVMM and `b` in volatile memory, AutoPersist transparently migrates `b` to NVMM. The code instrumentation in AutoPersist has a non-negligible cost. Even if the application does not use NVMM, it induces an overhead of 51% (9% with the QuickCheck optimization [289]).

Espresso does not instrument the code to detect cross-references, thus they may appear. To deal with them, Espresso can nullify a dangling reference at recovery. However, this approach is not satisfying: a dangling reference can appear because the developer thought erroneously that the referenced object is persistent, while it is volatile. Instead of silently losing data, the runtime should provide help to prevent such situations. Alternatively, Espresso proposes to rely on annotations to prevent cross-references. The type of an annotated reference becomes incompatible with the type of a volatile object. This mechanism is similar to a class-centric solution.

J-NVM avoids the problem of cross-references at all by relying on the class-centric model: an application may store a reference in NVMM only if the referenced object implements `PObject`.

3.4 Liveness by reachability

Programming with NVMM requires to deal with two fundamental concerns. First, after a crash, an object may be lost, which leads to a memory leak. This may happen when an object is allocated but not yet reachable from a persistent root. Second, after a crash, a reachable object may be partially deleted and thus unusable. Such a situation occurs when an object is freed but not yet removed from the reachable graph. To avoid both problems, J-NVM focuses on simplicity. It considers that once created, a persistent object remains alive as long as it is reachable from a persistent root. This approach is called liveness by reachability and commonly found in many frameworks (e.g., [217, 310]).

Liveness by reachability is implemented in J-NVM using a recovery-time GC, similarly to Makalu [63]. The GC traverses the graph formed by the persistent objects when the application resumes after a crash. This does not happen at runtime to limit the impact on performance (see §3.2.1). During the collection pass, if J-NVM finds a reference to a partially deleted object inside the reachable graph, the reference is nullified. Then, to prevent memory leaks, the persistent objects that are unreachable from a persistent root are deleted.

3.5 Example usage

As illustrated in Figure 3.3, programming with J-NVM is straightforward. Any class annotated with `@Persistent` is durable. For instance, this is the case of `Simple` in Figure 3.3 (line 1).

To run the application, the developer compiles the sources as usual, then passes a code generator over the bytecode files (the `.class` files). Any class marked with `@Persistent` is transformed.[3] The code generator replaces the volatile fields with persistent ones (lines 3 and 4 in Figure 3.3). Accordingly, accesses to such fields are replaced by persistent accesses (lines 8, 9, 12, 27 and 28). If a field is marked `transient` (line 5), the code generator keeps it in volatile memory, making no transformation. The developer may use transient fields to optimize the application, e.g., to deduce a volatile value from the persistent state (see §4.2).

In addition to the above transformations, the code generator also wraps methods into failure-atomic blocks. The `fa="non-private"` argument of `@Persistent` at line 1 specifies that each non private method has to be wrapped.

In the `Main` class of Figure 3.3, the application manipulates a `Simple` object. It starts by creating (or retrieving) a persistent memory region of 1 MB called `"/mnt/pmem/simple"` (line 17). A persistent memory region contains by default the persistent map `JNVM.root`. This map associates names with the root persistent objects used by the application. The `main` method uses this map to retrieve the persistent object associated with the name `"simple"`. If the object does not exist (line 19), the method allocates a new `Simple` object and records it in the map (line 20). Further, `main` retrieves the `simple` object, increments its `x` field, sets its `y` field and prints its content (lines 22-28). Line 30 creates a second `Simple` object and inserts it in the root map. The code then frees the first object still referenced by the local variable `s` (lines 31-32).

3.6 Methodology

J-NVM offers a general framework to inject durability in a Java application. Starting from a set of legacy classes to persist, the developer annotates them to generate appropriate proxies. Once transformed, the methods of the proxies are failure-atomic, that is they execute entirely, or not at all, despite system failures. Because J-NVM exposes a class-centric model that favors reliability over re-usability, the developer cannot directly use the volatile classes from the Java runtime in a persistent object (e.g., native arrays, or `java.lang.String`). Instead, the developer should use the drop-in persistent replacements provided in the J-PDT library (such as `PString` at line 9 in Figure 3.3).

[3] If the sources are unavailable, instead of relying on the `@Persistent` annotation, the tool takes as input an explicit list of classes to transform.

With J-NVM, the developer has to manage differently the life cycle of a persistent object than a volatile one. Liveness by reachability requires a persistent object to be reachable from the root map (`JNVM.root`). This restriction is commonly found in prior frameworks, such as the PMDK library [19]. Persistent objects have also to be explicitly deleted where appropriate. As illustrated at lines 31-32 in Figure 3.3, explicit deletion makes the code slightly more complex than with an integrated design. However, as shown in §3.2.2, such events are rare in persistent data stores.

```
1  @Persistent(fa="non-private")
2  class Simple {
3   PString msg;
4   int x;
5   transient int y;
6
7   Simple(int x) {
8    this.x = x;
9    this.msg = new PString("Hello, NVMM!");
10  }
11
12  void inc() { x++; }
13 }
14
15 class Main {
16  static void main(String args[]) {
17   JNVM.init("/mnt/pmem/simple", 1024*1024);
18
19   if(!JNVM.root.exists("simple"))
20    JNVM.root.put("simple", new Simple(42));
21
22   Simple s = (Simple)JNVM.root.get("simple");
23
24   s.inc();
25   s.y = 42;
26
27   System.out.println(s.x);
28   System.out.println(s.msg);
29
30   JNVM.root.put("simple", new Simple(24));
31   JNVM.free(s.msg);
32   JNVM.free(s);
33  }
34 }
```

Figure 3.3: How to use J-NVM.

3.7 Summary

In this chapter, we presented the programming model of J-NVM and how it allows for scalable and safer management of a persistent heap, while remaining intuitive for object-oriented programmers. (§3.1) We sidestep online garbage collection for persistent objects, by splitting them into an off-heap structure (NVMM) and an on-heap volatile "proxy" Java object (DRAM). (§3.2) Our decision of relying on explicit deletion (free) for persistent objects, as opposed to volatile objects that are garbage-collected online, is motivated from the pragmatic observations that online GC is presently incompatible with large heap sizes and not so beneficial in data store applications. (§3.3) We also stress out the necessity of having strong typing for persistent objects, that jointly, *(i)* eases code comprehension and avoids programming mistakes, *(ii)* or prevents dangerous persistent-

to-volatile references at compile time i.e, without costly online instrumentation. (§3.4) J-NVM further enforces referential integrity with recovery-time GC to reclaim memory or repair stall references left from faulty executions. (§3.5, §3.6) We demonstrated the usage of J-NVM from a concrete programming example and recapped the overall methodology to follow.

In the next chapter, we dive deep into the system design of J-NVM and the implementation of its key components, plus the high-level abstractions built on top: J-PFA and J-PDT.

Chapter 4

J-NVM:
Off-heap Persistent Objects for Java

This chapter covers the system design of J-NVM and implementation aspects of the persistent heap, J-PDT, J-PFA, and the code transformer.

It organizes as follows.

(§4.1) We describe the structure and generation process of volatile proxy objects.

(§4.2) We detail the specific life cycle of persistent objects, and how volatile proxy objects can represent them with only a limited lifetime.

(§4.3) We present J-NVM low-level interface and the help it provides to developers that do not want every memory accesses to be encapsulated in FASEs.

(§4.4) We continue with the structure and management mechanisms of the NVMM heap.

(§4.5) We explain our methodology to build recoverable data structures in J-PDT.

(§4.6) We present the redo-log algorithm of J-PFA and how proxies help us interpose on transactional reads or writes.

(§4.7) We finish by detailing the bytecode transformation process that JNVM-Transformer automatically performs to turn regular java objects into their persistent counterpart (NVMM data type + proxy).

4.1 Proxy objects

J-NVM exposes NVMM through persistent objects. A persistent object consists of a persistent data structure and a volatile proxy. The proxy implements the `POobject` interface. It contains the methods of the persistent object and defines an interface to access the persistent fields of the object, which are stored in NVMM, outside the Java heap.

Figure 4.1 shows a simplified version of the persistent object generated by the code generator of J-NVM from the example in Figure 3.3. The code generator is based on the ASM framework [74, 75], which provides helpers to simplify the rewriting of bytecode. To obtain Figure 4.1, the code generator first adds the `POobject` interface, which marks objects of the `Simple` class persistent, then it removes all the non-transient volatile fields. Any access to a non-transient field (e.g., lines 8 and 9 in Figure 3.3) is then transformed into a call to a generated method that accesses the persistent data structure (lines 7, 8 and 15 in Figure 4.1). Because of the `non-private` arguments at line 1 in Figure 3.3, the code generator wraps each method into a failure-atomic block. Such blocks (lines

6-10 and 14-16 in Figure 4.1) are delimited with `faStart()..faEnd()`.

Additionally to the above setters and getters, the generated code contains methods to manage the life cycle of a persistent object. The sections below detail these methods and then explain how the developer can use them jointly with the low-level interface to create custom persistent data types.

4.2 Life cycle

Association. By design, J-NVM decouples the persistent data structure of an object from the proxy that represents it in the Java world. J-NVM has thus to maintain an *association* between a proxy and the persistent data structure it gives access to. The `addr` field (line 20 in Figure 4.1) maintains such an association: it contains the address of the persistent data structure associated with the proxy. The getters and setters use this address to execute operations over the persistent fields. For instance, `x` is the second persistent field of `Simple` in Figure 4.1. As a consequence, `getX` returns the integer located at offset 8 in the persistent data structure at address `addr`.

Allocation The data structure of a persistent object is stored in NVMM. J-NVM allocates it in the constructor, using `JNVM.alloc` (line 7 in Figure 4.1). Once allocated, the persistent data structure is associated with the corresponding proxy. The method `alloc` takes two arguments: the Java class of the proxy, which is used during resurrection (see details below), and the size of the data structure.

Persistent references and resurrection A persistent object may hold a reference to another persistent object. As with primitive types, J-NVM provides the `readPObject` and `writePObject` methods to manipulate them (see for instance lines 27-28 in Figure 4.1). Because these methods manipulate proxies, the Java type system ensures that NVMM contains only references to persistent objects, and not to volatile ones.

To store a reference to an object `a`, `writePObject` stores `a.addr` in NVMM. Upon dereferencing `a`, `readPObject` dynamically creates a proxy associated to its persistent data structure. In detail, `readPObject` reads the address of the persistent data structure in NVMM, retrieves the name of the proxy class in its header, then allocates the corresponding proxy class. Once the proxy is allocated, `readPObject` calls the constructor generated at line 22 in Figure 4.1, before returning the proxy to the caller.

We call this constructor the *resurrect constructor*.[1] The resurrect constructor first associates the proxy with the persistent data structure (line 23). Then, it calls the `resurrect` method (line 24). This method, if overridden, may initialize transient fields upon resurrection (e.g., the `y` field in our example).

Free As seen at lines 31-32 in Figure 3.3, calling `JNVM.free` frees the persistent object. This method takes a proxy as argument. It frees the persistent data structure associated with the proxy and writes 0 in its `addr` field, which makes the proxy invalid: after a call to `free`, accessing the proxy throws an exception.

[1] In our implementation, the resurrect constructor uses a signature that cannot collide with a user-defined constructor.

```
1  class Simple implements PObject {
2   // transformed code
3   transient int y;
4
5   Simple(int x) {
6    JNVM.faStart();
7    this.addr = JNVM.alloc(getClass(), size());
8    setX(x);
9    setMsg(new PString("Hello, NVMM!"));
10   JNVM.faEnd();
11  }
12
13  void inc() {
14   JNVM.faStart();
15   setX(getX()++);
16   JNVM.faEnd();
17  }
18
19  // added code
20  long addr; // the persistent data structure
21
22  Simple(long addr) {
23   this.addr = addr;
24   this.resurrect();
25  }
26
27  PString getMsg() { return (PString)JNVM.readPObject(addr, 0); }
28  void setMsg(PString v) { JNVM.writePObject(addr, 0, v); }
29  int getX() { return JNVM.readInt(addr, 8); }
30  void setX(int v) { JNVM.writeInt(addr, 8, v); }
31  long size() { return 12; }
32 }
```

Figure 4.1: A generated persistent object.

4.3 Low-level interface

Prior research [57, 138, 174, 223, 354] shows that constructs such as failure-atomic blocks and transactions are costly, and in many cases not required by the application. For this reason, J-NVM also exposes a low-level interface that trades the simplicity of the high-level interface for better performance.

To use the low-level interface, the developer omits the `fa` argument in the `Persistent` annotation at line 1 in Figure 3.3. In that case, the code generator performs the same transformation as described above, but it does not wrap methods in failure-atomic blocks. To still create such blocks, the developer can call `faStart()..faEnd()` explicitly. It is also possible to fine-grained manage data persistence without failure-atomic blocks at all, as we detail next.

Outside a failure-atomic block, the field accessors behave differently. To achieve this, J-NVM maintains a per-thread counter that tracks the nested level of failure-atomic blocks. At runtime, J-NVM checks this counter when it loads or stores a field. If the counter is strictly greater than 0, J-NVM instruments the load/store to ensure that the failure-atomic blocks execute atomically. Otherwise, it grants a direct access to NVMM without mediation. We measured that checking this counter for each access has almost no performance impact because the counter is always in the L1 cache of the processor and the branch predictor makes correct predictions. For that reason, we have not investigated static analysis methods to eliminate them.

Accessing NVMM without mediation requires to ensure data persistence by hand.

The remainder of the section is devoted to describing the interface that J-NVM provides to help the developer in this task.

4.3.1 The `recover()` method

If an object does not use failure-atomic blocks, it can be in an inconsistent state at recovery. To prevent such a situation, the developer needs to override the `POBject.recover()` method. At recovery, before the application resumes, this method is called for each live object encountered during the collection pass.

4.3.2 Cache line management

A system crash can happen at any point in time. When such an event occurs, the application needs to find out which operations were applied before the crash. This requires to control the propagation order of the CPU cache lines to NVMM. With failure-atomic blocks, J-NVM transparently takes care of the cache line management. In particular, the system ensures that all the persistent stores of a block are propagated to NVMM at the end of the block.

When using the low-level interface, J-NVM does not enforce any propagation order, allowing the developer to make optimizations. J-NVM exposes three operations to control propagation to NVMM: `pwb`, `pfence` and `psync`. These operations implement the architecture-agnostic instructions defined by Izraelevitz et al. [174]. We adapted them to work with the Java memory model [176, 227], as also proposed in JEP 352 [124].

In detail, `pwb(addr)` adds the cache line of `addr` to the write pending queue of the processor. Because of the Java memory model, `pwb` may be reordered with other instructions. A call to `pfence()` prevents such a situation: it ensures that the preceding `pwbs` and stores to (both persistent and volatile) memory are executed before the succeeding `pwbs` and stores. The method `psync` behaves as a `pfence` and additionally ensures that the cache lines in the write pending queue are also propagated to NVMM.

J-NVM exposes `pfence` and `psync` directly in the `POBject` interface. `pwb` is accessible using the methods generated in a persistent object: `pwb()` flushes all the cache lines of the object, and `pwbX()` flushes the cache lines that hold field `x`.

4.3.3 Validation and recovery

Calling `pfence` prevents out-of-order execution inside the processor. This drastically decreases the instruction-level parallelism. As a consequence, reducing the number of `pfences` in the application is paramount for performance [102].

Unfortunately, liveness by reachability requires that each reachable object is always in a consistent state. For instance, if a newly-created object becomes reachable, its fields must be voided to prevent reading random values at recovery. It follows that a `pfence` should always precede a store that would make an object reachable. Enforcing liveness by reachability is thus harmful for performance.

To reduce the number of `pfences`, J-NVM does not consider an object to be alive when it is just reachable. Instead, the object has also a valid status stored in its persistent header. J-NVM considers that only both reachable and valid objects are alive. Internally, an object is allocated in the invalid state, and it only becomes valid after a call to the `validate` method. Similarly, J-NVM atomically deletes an object by invalidating it before recycling its memory. The valid state is totally transparent to the high-level

```
1  @Persistent
2  class LowLevel implements PObject {
3   PObject o;
4
5   LowLevel(String name) {
6    o = new Other();
7    o.pwb();
8    o.validate();
9    pwb();
10   JNVM.root.wput(name, this);
11  }
12
13  static void main(String[] args) {
14   a = new LowLevel("a");
15   b = new LowLevel("b");
16   pfence();
17   a.validate();
18   b.validate();
19  }
20 }
```

Figure 4.2: The low-level interface.

developer through the use of failure-atomic blocks. However, with the low-level interface, the developer may call directly `validate` to minimize the number of `pfences`.

Figure 4.2 illustrates how the developer may use the validation mechanism internal to J-NVM. The optimization consists of deferring validation after the pfence at line 16, to allocate and make reachable several objects with a unique `pfence`. In detail, the code allocates two objects `a` and `b` (lines 14-15), which themselves allocate a sub-object each (line 6). The two objects are added to the root map (line 10) using `wput`. This operation is weak, in the sense that it does not rely internally on a failure-atomic block, and thus does not executes `pfences`. The calls to `pwb` at lines 7 and 9 ensures that the cache lines of `a`, `b`, `a.o` and `b.o` are all added to the write pending queue. The call to `validate` at line `8` validates `a.o` and `b.o`. This call does not execute `pfence`: it just changes the `valid` state of the object and adds the cache line of the header to the write pending queue.

If a crash happens before line 16, since `a` and `b` are invalid, J-NVM will free them at recovery, even if they are already reachable from the root map. J-NVM will also delete `a.o` and `b.o` because they are not reachable from a live object. As a result, in case of a crash before line 16, all the allocated objects are deleted. Thus, by not executing any `pfence` before line 16, the code is correct. The unique `pfence` at line 16 ensures that if `a` (resp. `b`) is valid (lines 17 and 18), then so do `a.o` (resp. `b.o`).

In what follows, we detail the implementation of J-NVM. After a description of the persistent heap (§4.4), the next sections present the algorithm used to ensure failure atomicity (§4.6) and the library of persistent data types (§4.5).

4.4 The persistent heap

4.4.1 Memory allocator

Designing a persistent heap requires to address fragmentation: after multiple object allocations and releases, the free space is divided into small pieces, making allocation of

id *(15 bits)*	valid *(1 bit)*	next *(48 bits)*	state
class	0	any	**valid**
class	1	any	**invalid**
0	0	any	**free** or **slave**

Table 4.1: The block header and its associated states.

large objects impossible. Solving this problem is essential for a persistent memory since, by definition, it is long lived.

Usually, we eliminate fragmentation in managed languages such as Java by executing a compacting phase during a GC cycle [181]. However, J-NVM avoids the use of a GC at runtime to deliver better performance. To deal with fragmentation, J-NVM relies instead on a memory layout inspired by the work of Pizlo et al. [260]. This layout splits the heap in blocks of fixed size, exactly as we do with the blocks of a file system. If a large object does not fit into a single block, J-NVM creates a linked-list of blocks to store its content.

Using blocks of fixed size eliminates the fragmentation problem by design since we can always allocate large objects. However, this memory layout also increases the complexity of accessing large objects. J-NVM hides this complexity behind proxies. Instead of keeping a single address for the persistent data structure (line 20 in Figure 4.1), the proxy actually contains an array that holds the addresses of all its blocks. The array is populated during the association between the proxy and the persistent data structure. Once the proxy is initialized, retrieving the block that contains a given field simply requires a division.

4.4.2 Block header

A block starts with a header of a single word (8B) that provides its state (see Table 4.1). For allocated blocks, `next` gives the next block that belongs to the object. When `id` is not equal to 0, the block is the first block of a persistent object, called the *master block*. In this case, J-NVM uses `id` as an index in a persistent array to retrieve the name of the proxy class during resurrection (§4.2). Otherwise, when `id` equals 0, `valid` is necessarily equal to 0. We have then two possibilities. The block can be a *slave* block, which means that it belongs to a persistent object but it is not the first block. Alternatively, the block may also be free in which case it does not belong to any object.

4.4.3 Block allocation

J-NVM allocates a free block using a bump pointer stored in persistent memory and a free queue stored in volatile memory. The free queue is implemented with a concurrent queue to scale with the number of threads. To allocate a block, J-NVM tries first to obtain one from the free queue. If this fails, J-NVM creates new blocks by bumping the bump pointer. When J-NVM allocates a block, except when it uses the bump pointer, it only accesses volatile memory and never updates NVMM. The task of initializing the block (as a master or a slave) is delegated to higher levels of the software stack.

4.4.4 Recovery

At startup, J-NVM executes a recovery procedure. To create the volatile free queue, this procedure uses a volatile bitmap. For each block, the bitmap indicates with a bit

```
1 void updateO(PObject n) {
2  n.validate();
3  pfence();
4  setO(n);
5 }
```

Figure 4.3: Atomic update.

whether the block is free or not. Starting from the root map, J-NVM traverses the live object graph. As any graph traversal, this procedure has thus a complexity linear in the number of live objects. When it finds a reference to a valid object, J-NVM marks its blocks as alive in the bitmap, and calls the `recover` method of the object. Otherwise, since the referenced object is invalid, the reference is set to null.

At the end of the traversal, J-NVM populates the free queue with the blocks marked as free in the bitmap. In doing so, J-NVM writes 0 in the valid bit of each free block to ensure that a newly allocated block is necessarily in the invalid state. Once the recovery procedure terminates, J-NVM triggers a `pfence`.

4.4.5 Object allocation

When J-NVM allocates an object, it first allocates its blocks using the free queue and the bump pointer. Then, J-NVM writes its `id` in the master block and links appropriately the slave blocks. During the allocation of an object, J-NVM does not use any fence since a master block is necessarily in the invalid state.

4.4.6 Object deletion

To delete an object, the application explicitly calls `JNVM.free`. This method invalidates the master block and adds all its blocks to the volatile queue. J-NVM does not execute a `pfence` in `JNVM.free`, which allows a developer that uses the low-level interface to use a single `pfence` to free a graph of objects. For instance, to free `a` and `a.o` in Figure 4.2, the developer marks `a` as invalid by calling `JNVM.free` and then explicitly triggers a single `pfence`. In case of a crash after the `pfence`, J-NVM will delete `a` because it is invalid, and `a.o` because it is not reachable by valid objects. Executing a `pfence` in `JNVM.free` for `a.o` is thus useless.

4.4.7 Atomic update

For a developer that uses the low-level interface, J-NVM provides a method that atomically updates a reference. This method ensures that the collection pass executed at recovery cannot nullify the reference. As shown in Figure 4.3, the implementation of this method is straightforward. It simply validates the new reference, executes a `pfence` and updates the reference. Calling `pfence` ensures that the new object is valid before being reachable. The code generator creates this method for each field that references a persistent object. It also generates a second helper that updates a reference and additionally frees the old referenced object atomically. Our NVMM portage of the Infinispan key-value store (see §5.1) uses these methods to ensure at all time a sound association between a key and its values.

4.4.8 Implementation details

This section completes the presentation with implementation details.

4.4.8.1 NVMM access

J-NVM leverages the `Unsafe` interface to access NVMM, as indicated in §2.8.2.4. This interface directly inlines assembly instructions in the generated code, similarly to the magic interface of JikesRVM [137]. J-NVM uses this interface to read and write NVMM. We also implemented `pwb`, `pfence` and `psync` with this interface. For the recent Intel architecture used in our experiments, we implemented `pwb` with the `clwb` instruction, and, even though they are conceptually different, we implemented both `pfence` and `psync` with the same `sfence` instruction. As a result, by using the intrinsic mechanism and adding three new instructions, J-NVM accesses NVMM at nearly native speed (see our evaluation in §5.3.5).

4.4.8.2 Small immutable objects.

As presented above, J-NVM stores objects in blocks of fixed size. Consequently, the system is subject to internal fragmentation, that is each object consumes a whole block regardless of its size, potentially wasting NVMM. To avoid internal fragmentation for small immutable objects, e.g., `PString` in Figure 3.3, J-NVM uses memory pool allocators built atop the default one (§4.4). These allocators are able to pack several objects of the same size in a single block.

Memory pool allocators handle only immutable objects. This comes from the fact that the failure-atomic algorithm described in §4.6 works at the block level and not at the object level. To understand why, consider that two threads execute each a failure-atomic block that modifies an object. If the two updated objects are located in the same block, the block will be replicated twice, and the content of the two replicas will diverge. As reconciling these replicas requires complex algorithms, J-NVM avoids such a situation by packing only immutable objects in the same block.

4.4.8.3 Heap relocation.

J-NVM ensures that the persistent heap is relocatable. For that, instead of storing absolute addresses in NVMM, it stores only offset relative to the beginning of the heap. This is similar to the *offset-based pointer representation* in NVAlloc [108] or Ralloc [76] (§2.7.5.2).

4.5 J-PDT

J-PDT is a stand-alone library of persistent data types built on top of the low-level interface. Similarly to those in §2.7.1 and §2.7.2, these types provide failure-atomic operations without resorting to logging mechanisms. This section gives an overview of the main data types present in the library.

4.5.1 Persistent arrays

J-PDT provides arrays of fixed sizes. An array contains its length at offset 0 and the elements afterward. This class provides a constructor to initialize its content appropri-

ately, accessors to retrieve the elements, and methods to flush either an element, or the array in full. In addition, J-PDT provides extensible arrays similar to the `ArrayList` class of the standard Java library. To extend an array, we rely on the low-level atomic update methods described in §4.4.7.

4.5.2 Maps and sets

J-PDT includes several set and map abstractions. Implementing these data structures is more challenging than implementing the arrays and extensible arrays detailed previously. However, the decoupling principle introduced by proxies offers a general pattern of solution: the content of a persistent object is stored in NVMM, while its logic remains in volatile memory. We largely identified this recurring pattern in §2.7. With proxies, the separation is seamless: all data are kept and accessible from a single Java class. Note that we absolutely avoid storing duplicates of the data in proxies, transient fields only hold control data necessary to implement the logic of the structure.

Overview. We first implement a persistent set as a persistent map that associates each key with itself. Then, to construct a persistent map, J-PDT stores the references to the persistent key/value pairs in a persistent extensible array. In the proxy, J-NVM maintains two volatile data structures: a *free queue* that stores the empty cells in the persistent array, and a *mirror map* that mirrors the persistent array in volatile memory. The mirror map implements the logic of the data structure. For instance, for a hash table, we use a Java `HashMap`, and for a persistent binary tree, we use a Java `TreeMap` (a red-black tree). In §5.3.4, we provide an evaluation for a persistent HashMap, TreeMap, and SkipListMap we built with this method.

During resurrection (see §4.2), J-PDT inspects each cell of the persistent array. If it finds a non-null reference to a pair `(k,v)` at index `n`, it adds the mapping `(k,n)` to the volatile mirror. Otherwise, `n` is added to the volatile free queue. To add a key/value pair, J-PDT first removes a free cell index `n` from the volatile free queue. If the queue is empty, the persistent array is extended and the queue populated accordingly. Then, J-PDT allocates a new pair in persistent memory, and writes its references at index `n` in the persistent array. To remove a key/value pair, J-NVM adds its index, say `n`, to the volatile free queue. Then, it writes a null reference at index `n` in the persistent array.

The persistent data structure is always in a consistent state because modifying it incurs a single write to NVMM (at the right index in the persistent array). Note that as in Mirror [140] or SOFT [354], the persistent layer is updated before the volatile *mirror map*, and reads always go through the volatile map first. Therefore, only persisted states can be read.

Concurrency. J-PDT provides concurrent persistent maps and sets. For that, a persistent data structure is mirrored with an appropriate concurrent class from the Java runtime. For instance, the mirror of the concurrent persistent hash table of J-PDT is the volatile concurrent hash table of the Java runtime. Internally, the data structure relies on methods of the Java runtime able to execute a closure when a key is inserted or removed. The Java runtime ensures that the execution of the closure is linearizable. J-PDT leverages this feature of the runtime to update appropriately the persistent array when a key is either inserted or removed.

For instance, to insert a new key/value pair in a concurrent hash map, J-PDT calls the `compute` method of the `ConcurrentHashMap`. If the key is absent, the closure

(i) finds a free slot in the volatile free queue, *(ii)* allocates a new persistent key/pair, and *(iii)* writes a reference to it in the persistent array.

Base, cached and eager maps and sets. Resurrecting a persistent object has a performance cost. Indeed, J-NVM needs to allocate a proxy, traverse the linked-list of blocks of the persistent object, and, for some object, deduce a volatile state from the persistent state.

To avoid this cost for values stored in maps and sets, J-PDT proposes different implementations of maps and sets with different trade-offs between performance and memory consumption.

The default implementation presented above is called the *base* implementation, and it favors memory consumption. For each key in persistent memory, the base implementation keeps a proxy in volatile memory, but it systematically allocates a new proxy when the application retrieves a value associated with a key.

Both the *cached* and *eager* maps and sets trade memory consumption for better performance. They maintain a cache of the proxies to the values. The eager implementation populates the cache during resurrection, while the cached implementation populates the cache on demand. In our implementation, the cache contains all proxies but it would be possible to extend this code to include only the hottest proxies.

```
1  public interface Convertible<V extends PObject> {
2    V copyToNVM();
3  }
4
5  public class AutoPersistMap<K extends PObject,
6                 V extends PObject,
7                 I extends Convertible<K>,
8                 J extends Convertible<V>>
9      extends AbstractMap<K,V>
10     implements Convertible<RecoverableMap<K, V>>, RecoverableMap<K,V> {
11
12    public V putConvert(I key, J value){...};
13 }
```

Figure 4.4: Declaration of the AutoPersist-like persistent map.

AutoPersist map. We also implemented a persistent map that mimics the automatic migration algorithm of Autopersist [288]. To this end, we introduce a new `Convertible` interface. This interface provides to objects a self-defined method (`copyToNVM()` in Figure 4.4) to convert themselves into some pre-defined persistent type.

In a nutshell, the AutoPersist-like map is a wrapper for any of our `RecoverableMap` we described above, and additionally offers a `putConvert()` method. This method takes as input two convertible objects and automagically makes them persistent as they are inserted inside the map. As in AutoPersist, the map migrates the whole underlying graph behind inserted objects, by propagating through any persistent fields that is also `Convertible`. The migration algorithm strictly implements the one described in AutoPersist. For instance, it uses a thread-local work queue and pointer queue, to avoid large recursion graphs, or infinite recursion due to potential circular references.

Of course, because it implements the deep migration protocol with static types, it can not reproduce the dynamic instrumentation costs of AutoPersist. Compared to our others persistent maps, it only pays an additional NVMM copy on `put()`. The interesting

```
1  // Entry proto-interface
2  private interface Entry extends PObject {
3    void apply();
4  }
5
6  // Log entry types
7  public static class CopyEntry implements PObject, Entry {
8    private MemoryBlockHandle orig;
9    private MemoryBlockHandle copy;
10
11   CopyEntry(long orig, long copy) {
12     this.orig=MemoryBlockHandle.fromAddr(orig);
13     this.copy=MemoryBlockHandle.fromAddr(copy);
14   }
15
16   public void apply() { MemoryBlockHandle.copy(getOrig(), getCopy()); }
17 }
18
19 public static class ValidateEntry implements PObject, Entry {
20   private MemoryBlockHandle orig;
21
22   ValidateEntry(PObject pobj) { this.orig=pobj.getMasterBlock(); }
23
24   public void apply() { getOrig().validate(); }
25 }
26
27 public static class InvalidateEntry implements PObject, Entry {
28   private MemoryBlockHandle orig;
29
30   InvalidateEntry(PObject pobj) { this.orig=pobj.getMasterBlock(); }
31
32   public void apply() { JNVM.free(getOrig().getAddr()); }
33 }
```

Figure 4.5: Log entry type for the redo log.

bit compared to AutoPersist, is that from the typing of the map methods, we always know whether the returned object is persistent or volatile.

4.6 Failure atomic blocks

As indicated in §3.5, the high-level programming model of J-NVM provides failure-atomic blocks. Many systems already offer such a construct [19, 92, 102, 103, 143, 232, 310] (see §2.7.3). Our algorithm is not new by itself. It is inspired by Romulus [102] and adapted to our persistent memory layout. We have implemented failure-atomic blocks not to advance the state of the art, but instead to verify that we can build a developer-friendly system based on our decoupling principle.

The log. J-NVM implements a standard redo log. At a high level, during the execution of a failure-atomic block, J-NVM adds all the modifications (allocations, writes and frees) to a per-thread persistent redo log, leaving original data intact. When reaching the end of the failure-atomic block, J-NVM commits the log then applies its modifications to NVMM. Note that the entries constituting the log contain no data, only pointers to the affected blocks. As illustrated in Figure 4.5, a copy entry records the original block and in-flight block, and (in)validate entries record the master block of the affected object.

```
1  public class PRedoLog implements PObject {
2
3    /* Entry type declaration */
4
5    //The log
6    private PArray<Entry> log = new PArray<>();
7
8    //Default constructor
9
10   //Logging methods
11   public void logCopy(long orig, long copy) {
12     log.add(new CopyEntry(orig, copy));
13   }
14   public void logValidate(PObject pobj) {
15     log.add(new ValidateEntry(pobj));
16   }
17   public void logInvalidate(PObject pobj) {
18     log.add(new InvalidateEntry(pobj));
19   }
20
21   /**
22    * Invoked either after a crash, or at the end of an FA block.
23    *  PRE _ entries written to log (validated)
24    *   1 _ fence and validate log
25    *   2 _ apply entries
26    *   3 _ fence, invalidate log
27    * POST _ nothing, entries are cleared at the start of next FA block
28    *
29    */
30   public void redo() {
31     this.fence();
32     //Commit log
33     this.validate();
34     this.fence();
35     log.forEach(Entry::apply);
36     this.fence();
37     //Erase log
38     this.invalidate();
39     this.fence();
40   }
41
42   /**
43    * Invoked at the start of an FA block.
44    */
45   public void init() {
46     //free all previous log entries
47     log.forEach(e -> e.destroy());
48     log.clear();
49   }
50
51 }
```

Figure 4.6: The redo log.

The protocol. Before committing the log, J-NVM does not uses any `pfence` as data in NVMM is unchanged [102]. As illustrated in Figure 4.6 (line 30-39), to commit the log, J-NVM first executes a `pfence` to ensure that the state of the log is persisted. Then, it marks the log as committed and executes a second `pfence` to ensure that its new status reaches NVMM. Finally, it applies the operations recorded in the log without `pfence`. If a crash occurs during this last step, the log will be replayed.

Recording object mutations. To update a persistent object, J-NVM considers two cases. If the object is invalid, J-NVM directly modifies the object. This may happen,

for instance, if the object is allocated then modified in the same failure-atomic block. Modifying an invalid object is safe because it is deleted at recovery when a crash occurs before commit. Now if the object is valid, J-NVM maintains two versions of each modified block of the object, an original one and an in-flight one. Upon reading, J-NVM uses the in-flight block if it exists, and the original block otherwise. Upon writing, if the in-flight block is already present, J-NVM directly performs writes to it. Otherwise, it allocates an in-flight block, adds the pair (original, in-flight) to the log then performs writes to the in-flight block.

Replay log entries. At the end of a failure-atomic block, J-NVM marks the persistent log as committed then plays the operations recorded in it (line 33, 34 in Figure 4.6). If it finds an allocation, J-NVM transparently validates the new object, which makes the object alive if and only if it is reachable (line 24 in Figure 4.5). If it finds a deletion, J-NVM calls the `JNVM.free` method (line 32 in Figure 4.5). If it finds an update, J-NVM copies the in-flight block into the original block (line 16 in Figure 4.5).

Log recovery. After a failure, J-NVM first handles the per-thread logs of failure-atomic blocks, then it executes the recovery procedure (see §4.4.4). If a crash occurs before the block was fully played, J-NVM replays its operations. If the crash occurred before the block was committed, J-NVM aborts it. J-NVM erases the log and lets the recovery procedure garbage collect the in-flight blocks as well as all the objects allocated in the block (these objects are still in the invalid state).

4.7 Bytecode transformer

JNVM-Transformer is an off-line tool that generates proxies for J-NVM from regular Java objects. It works as a post-compilation plugin integrated in the maven build system, and performs bytecode-to-bytecode transformations on annotated classes. The bytecodes (.class files) of the target classes are automatically enhanced to implement the decoupling principle of J-NVM and methods imposed by the framework.

Key aspects of the transformation process were already indicated in §3.5 and §4.1. This section describes with more detail how it transforms a single class in isolation (§4.7.2), then how it accounts for inter-class dependencies in class hierarchies (§4.7.3).

4.7.1 Proxy generator

Related work. Annotation processing for bytecode enhancement is an extremely common theme in Java. In industrial applications for instance. Hibernate [11], as a JPA provider, transforms entity classes at class loading time[2]. It does so to generate internal utility methods for the framework, declarations for the Lucene indexes [4] or Protobuf [308] message types. Older research on orthogonally persistent Java attempted inserting read and write barriers in objects without tempering with the JVM for better portability. Marquez et al. [230] applied bytecode transformations at class loading time for a portable orthogonally persistent Java. Aspect-oriented [195] frameworks were also proposed [37, 38, 258] with the same goal in mind.

[2] A JVM agent is installed to intercept classes right before a class loader instantiates them.

Proxy building. In J-NVM, proxies are specific objects that must abide to a set of rules dictated by the `PObject` interface. While hand-coding proxy objects is possible, the task requires good knowledge of J-NVM internals. Further, we notice that proxies contain no additional information compared to plain Java objects, meaning that a static transformation between the two is operable. JNVM-Transformer automatically operates the transform at post-compilation time, to avoid the recovery overheads of rewriting class bytecodes when classes are loaded at runtime. In all, a proxy generator was not absolutely necessary, but come as a boon for developers' productivity.

Practical aspects. JNVM-Transformer is written entirely in Java, and relies on two third-party frameworks to implement its bytecode enhancements. ByteBuddy [323], which provides a high-level declarative API and enables reasoning about bytecode transformations from Java source concepts rather than bare Java bytecodes. ASM [74, 75], a lower-level library that reads compiled class files and proposes a visitor pattern to rewrite matching bytecodes on the fly. For ease of use, JNVM-Transformer is fully integrated in the maven build system as a post-compilation plugin. Therefore, it transparently enhances user-defined classes when building code artifacts.

4.7.2 Base case: POJO bytecode enhancement

We now describe the core steps in the transformation process that turns regular Java objects into J-NVM proxies.

1. **Define and initialize a class-wide `classID` static field.** J-NVM safely dereference persistent pointers, meaning it needs to call the reconstructor of the appropriate proxy class. For this reason, `classID` are stored in NVMM block headers, and are internally mapped to proxy reconstructors (method handles). For that purpose, J-NVM has to be aware of user-defined persistent classes in order to attribute them non-conflicting `classID` values. As such, we require that proxy classes define a `classID` static field that must be populated at the static instant with a call to `JNVM.registerUserKlass()`. Internally, J-NVM maintains a persistent map between Java class name strings and `classID`, such that the identifiers stay valid across multiple executions.

2. **Compute the off-heap memory layout of the persistent object.** This step is crucial to issue NVMM allocations of the right size and to note the offsets of each field within persistent objects. We iterate over all fields defined in the class, skipping `static` or `transient` ones. We also check that non-primitive fields implement the `PObject` interface, if not, we bail with an exception because persistent references must point towards persistent objects. If all is in order, we simply accumulate the total NVMM layout size as we iterate and along, we remember each fields' offset in the layout. We also mark `final` fields to later adjust their setter visibility to `private`.

3. **Define and implement getters and setters for persistent fields.** We generate getters and setters for each persistent field. Their implementation calls `PObject` helper methods that read or write Java fields to NVMM from field offsets. We use the naming convention of Java EE entity beans[3] for getters/setters. If the class already defines them, we instead generate private accessors with non-conflicting names (leading underscore character). The original getters/setters are then augmented to call the private accessors in place of getField/putField Java bytecodes.

[3] get+fieldName or set+fieldName, with fieldName first character as uppercase

4. **Substitute accesses to persistent fields with calls to the generated getters or setters.** Actually, we need all instances of getField/putField bytecodes on persistent fields to be replaced with a method call to their getter or setter. We only exclude our private accessors from this substitution pass for obvious reasons.

5. **Define and implement the tracing method for the recovery-time GC.** We also automatically provide the tracing method for our recovery-time garbage collector. We first call a `PObject` helper method to apply the GC "mark" on the current persistent object, then, for every field that holds a reference to a persistent object, we recursively call their implementation of the GC tracing method.

6. **Remove any non-transient field.** We do not want persistent fields to have a volatile mirror or duplicate in their proxy. For this reason, we strip from the class all declared fields that are neither `transient` nor `static`. Stripping them is safe to do at this stage. We already generated our NVMM layout definition. In addition, all the field accesses were already substituted with the getters/setters of persistent fields.

7. **Define and implement a "reconstructor".** Reconstructors are essentials to proxies. They let them be re-instantiated from their underlying data structure. Their implementation is straightforward: they take as argument a `Void` and `long` parameter. The first argument is only useful to avoid conflicts with other user-defined constructors. The `long` argument is our persistent pointer type: an offset from the base address of the heap. The implementation of a reconstructor simply calls the default constructor to properly initialize the proxy object, then delegates the proxy re-association to a J-NVM helper method.

8. **Wrap non-private methods with failure-atomic blocks.** For all user-defined methods that are not: private, accessor methods, static methods, the default constructor, the "reconstructor", or overridden from `PObject`; we add library calls to J-NVM `faStart()` and `faEnd()`, at the beginning and end of the instrumented methods.

9. **Add `PObject` interface and implement all abstract methods.** We add class decorations that indicate the instrumented class implements the `PObject` interface. Then, we use an ASM visitor to copy all methods from J-NVM proxy template classes (abstract classes) into the instrumented class. We choose the appropriate proxy template based on object layout size, to support either mono- or multi-block objects. We skip abstract methods or the ones we implemented above. Proxy template constructors and "reconstructor" are transformed into private methods with different names. The static initializer methods from the template and instrumented classes are combined into a single one.

10. **Enhance constructors for NVMM allocation.** Proxy constructors must perform an NVMM allocation of the correct size, except for the default constructor and "reconstructor". This ensures that subsequent initialization of persistent fields can be satisfied. For that, we use an ASM visitor that adds a call to the proxy template constructor right after the `super()` call of the instrumented constructor. In Java, a call to `super()` must always be the first instruction in constructors, so we cannot insert our NVMM allocation before.

11. **Remove `@Persistent` annotations.** At last, all transformations were performed on the instrumented class. We may now remove the `@Persistent` annotation since it has become a fully-fledged J-NVM proxy class. We use a simple ASM visitor that matches our annotation and returns *null* instead.

To briefly summarize, JNVM-Transformer enhances class bytecodes to: *(i)* replace volatile fields with persistent ones, *(ii)* substitute field access with instrumented getters/setters, *(iii)* leave out `transient` fields, *(iv)* wrap public methods with failure-atomic sections, and *(v)* add `POBject` interface and necessary method implementations expected by the framework. As a result, plain Java objects become proxy types compatible with J-NVM.

4.7.3 Inheritance: transformation in an arbitrary class hierarchy

We just described the automatic enhancement steps for POJOs, but in any realistic scenario, we can not limit the process to isolated classes. Our maven plugin enlists for transformation all `@Persistent` annotated classes, when sources are available, or classes listed in a configuration file when they come from pre-compiled external artifacts. These classes can absolutely be related to each other with user-defined inheritance schemes.

Supported class hierarchies. J-NVM only requires persistent objects to implement the `POBject` interface, and *does not restrict in any way user-defined class hierarchies*. In contrast, PCJ [16] institutes that all persistent types descend from a common "Persistent Object" ancestor class. This is also the case in Espresso [329], but is only visible from their internal representation of classes in the JVM.

Instead in J-NVM, we noticed we did not have to be this restrictive, rather, that we could accommodate for any inheritance hierarchy. In essence, all types descend from "Object" in Java, persistent objects in J-NVM are no exception. Moreover, we allow the direct parent class of a persistent one to be transient or persistent regardless. In the end, the hierarchies that we support always begin from "Object" as usual, then can be an arbitrary interleaving of transient and persistent classes with no additional restrictions.

Additional transformation steps. Tackling these kind of class hierarchies necessitates to operate a dichotomy in the transformation steps depending on whether a class already has a persistent ancestor. When instrumented classes have no persistent ancestor, the transformation steps are exactly the ones described in §4.7.2. However, when persistent parents already exist, the process is not as trivial as adding calls to super methods everywhere. We list below the extra crucial steps to instrument classes with persistent parents. In particular, the reader should pay attention to changes that are necessary to allocate NVMM properly.

1. **Persistent parent identification.** We first need to crawl back up the hierarchy through declared parent classes. We stop upon finding a class implementing `POBject` or when we reach Java's "Object" class. If we find a `@Persistent` annotated class, we instrument it before the one currently being treated, such that layout information are available when we process the child class. This step allows us to deduce whether the class being instrumented is the first persistent one in its hierarchy, and to adjust the following transformation steps.

2. **Extend implementation of the GC tracing method.** When a persistent parent exists, we first need to make a super call to its own tracing method, to cover persistent parent fields as well.

3. **Specific "reconstructor" implementation.** We do not call the default constructor and J-NVM helping method, but the "reconstructor" from the super class, followed by the user-defined callback for transient field reconstruction.

4. **Skip proxy template class copy.** Proxy template methods are already available through the super class, hence we do not need to use again our ASM visitor to copy implementations of the mandatory methods in `PObject`.

5. **Extend parent NVMM layout.** In Java, the memory layout of a child class is simply its own concatenated to its parent's. We adopt the same approach, and begin counting field offset values from the last one in the parent class. The child class allocation size in NVMM is then the sum of its own layout length and that of its parent.

6. **Propagate NVMM allocation size through parent constructors.** We previously noted that proxy constructors needed to be enhanced for NVMM allocation. They allocate NVMM right after their super call – which is the earliest possible. Now consider that, Java forces every constructor to call `super()` first thing, meaning all constructors in the hierarchy are traversed. However, we only need to perform a single NVMM allocation. Incidently, the only possible location to allocate NVMM, hierarchy-wide, is right after the super call of the topmost persistent ancestor (the one that do no have any persistent ancestors above). The issue though, is that this ancestor do not know the allocation size of the child class whose constructor initiated the chaining of `super()` upcalls.
We solve this by adding the specific allocation size of the child class as an extra argument in persistent object constructors. In doing so, the allocation size can be propagated upward to the first ancestor that performs the NVMM allocation. To stay consistent with constructors defined by the user, we add this argument in package-protected constructors only – not the public ones. In detail, we duplicate user defined constructors in package-protected ones and add an extra argument for allocation size. We pass this argument to the first `super()` call in the constructor implementations. We leave untouched the rest of their implementation. Afterwards, we overwrite user-defined implementations in public constructors to delegate to the package-protected ones we just crafted. The extra argument is statically set to the layout size of the instrumented class.

4.8 Summary

With this chapter, we made a thorough tour of J-NVM design and functionality.

Proxy objects. In the first part of this chapter, we presented the fundamental decoupling principle we apply to persistent objects (§4.1). The split between the NVMM data structure and volatile proxy object allows us to separately manage data on the persistent heap. This separation further dissociates the permanent lifetime of persistent data from that of its ephemeral representant object (§4.2). We abstract these divergences by extending proxy objects to seamlessly interact with NVMM management. In particular, NVMM allocation is performed first thing in proxy constructors, and a specific "reconstructor" re-instantiates a proxy simply from the direct address to its underlying NVMM data structure. From a developers standpoint, proxies are then fully transparent. The "reconstructor" is only used internally, and the remainder of their life cycle feels natural in object-oriented programming. Precisely, persistent objects are durably allocated with `new`, and are either retrieved from the *root map* (global object directory) or when de-referencing a persistent pointer.

Low-level interface. The J-NVM framework also permits disabling automatic FASEs interposition for expert programmers that want to implement highly optimized structures

(§4.3). The persistence instructions (`pwb`, `pfence` and `psync`) are then exposed (§4.3.2). A `recover()` callback method can be overridden for algorithms that rely on a recovery phase to correct their state before resumption (§4.3.1). Finally, a "valid" bit in persistent object headers can be flipped manually with the `validate()` method to defer the instant at which objects become logically durable (§4.3.3). Unless they have this bit is set, the recovery GC recycles persistent objects. This mechanism can be leveraged to make *persistence by reachability* more flexible and efficient (`pfence` grouping), as it tolerates that objects are not yet fully flushed and persisted before they first become reachable from a persistent root.

The second part of this chapter was dedicated to describing implementation aspects of the persistent heap (§4.4), and the high-level utility libraries that provide: J-PDT (§4.5), J-PFA (§4.6), and JNVM-Transformer (§4.7).

Heap management. (§4.4.1) Our memory allocator is designed to be extremely simple and fast. It avoids fragmentation by only allocating 256B fixed-size blocks. Either from a shared bump pointer, or by recycling freed blocks maintained in a concurrent volatile free-list (§4.4.3). Objects larger than 256B are accommodated on multiple blocks. For recovery purposes, these are arranged as a linked-list by installing a `next` pointer in each of their blocks' header (§4.4.2). The blocks need not to be physically contiguous, since their proxy maintains all associated addresses in a volatile array, to translate field offset values into actual memory locations with just an extra arithmetic division. The allocation process itself is free of any persistent flushes and fences, since all metadata living outside of data blocks can be fully reconstructed on recovery from the GC pass (§4.4.4).

On top of our block allocator, object (de)allocation (§4.4.5, §4.4.6) simply have to populate the header of the object's first block. A *classID* durably identifies the structure's type in NVMM and eliminates user type-casting mistakes. Flipping the *valid* bit of a block's header is the only atomic step required to make the associated object transient or durable when reachable from a persistent root; thanks to the recovery time GC (§4.4.7).

Persistent data types. (§4.5) J-PDT provides a library of hand-tuned set-type recoverable data structures for NVMM, compatible with the JDK collection interfaces (`java.util.*`). Our data structures are NVMM-DRAM hybrids that decouple their data layer (NVMM) from their search layer (DRAM). The data layer consists of the same recoverable extensible array (§4.5.1) which replicates Java's `ArrayList`. It is implemented from our base persistent arrays – persistent replacements for Java's native arrays of any primitive type. (§4.5.2) The search layer is essentially a volatile data structure from `java.util` that indexes proxies. It is always updated *after* the data layer, such that only persisted content are visible. In the manner of *Efficient lock-free sets* [354], the volatile data structure is fully reconstructed on recovery simply by iterating over the persistent data array.

Failure-atomic blocks. (§4.6) J-PFA is the FASE implementation of J-NVM. It works as a redo-log but with block granularity, inspired by Romulus [102]. Instead of logging every single word-sized persistent accesses and mutations, heap blocks are duplicated when first mutated within a FASE. Only the addresses of duplicated blocks

are appended to the redo log, not the data. During the remainder of the FASE, all reads/writes are easily directed to the new duplicate block, thanks to our proxy objects. At the end of the FASE, the redo log is committed, then every log entry is replayed in order. That is, every in-flight block is copied back to its original location. The log is then marked as "clear", meaning all in-flight blocks will be recycled by the recovery time GC. With this protocol, until the log is committed, original blocks are never mutated. Once the log is committed, original blocks are overwritten, but the FASE effects are fully recorded and can be replayed again and again.

Code generator. (§4.7) JNVM-Transformer is a Java bytecode-to-bytecode post-compilation tool that enhances regular Java objects (POJOs) to become persistent objects compatible with J-NVM. It fully integrates with the maven build system and works its magic from a single `@Persistent` annotation (§4.7.1). It successively transforms the attributes and methods of an object to work with J-NVM (§4.7.2). In short, it computes the object's NVMM layout from its fields, then deletes the fields, and generates getters and setters that call J-NVM low-level framework to read or write NVMM locations. All field accesses are also substituted with calls to their getter or setter. Constructors are enhanced to request memory from our allocator, and a "reconstructor" is fully generated to resurrect the proxy. Last, all other necessary methods (abstract) from the `PObject` interface are generated. (§4.7.3) We mark that the bytecode transformation process do not restrict in any way the hierarchy of user classes. That is, we accommodate for the transformations to work in any circumstances. We do not require to inherit from a "persistent base class", and `@Persistent` can enhance any class regardless of its parent type (volatile or persistent alike).

To conclude, J-NVM offers a feature-rich interface to NVMM in Java. It was especially designed to reduce persistence overheads in crash-free executions. Thanks to its flexible low-level interface, J-NVM can easily accommodate for other algorithms or protocols found in the literature (§2.7). The fact that the framework orchestrates recovery, and only requires programmers to reason about it on a per-object basis, makes it easy to grasp and feels extremely natural in object-oriented programming.

In the next chapter, we present a detailed evaluation of J-NVM, J-PDT and J-PFA, and how they perform when applied to an industrial-grade data store.

Chapter 5

Evaluation of J-NVM

In this chapter, we present the performance of J-NVM across a variety of workloads and provide a detailed comparison against other existing approaches. It organises as follows.

(§5.1) We describe our experimental setup: hardware, test applications and competitors.

(§5.2) We present our main benchmark and its synthetic results.

(§5.3) We conduct a comprehensive analysis of J-NVM performance. That includes impact of the workload (§5.3.1), of multi-threading (§5.3.2), performance of recovery (§5.3.3), of our data types (§5.3.4) and of the physical accesses to NVMM in with J-NVM (§5.3.5).

5.1 Experimental setup

Hardware and system The test machine is a quad-Intel CLX 6230 hyperthreaded 80-core server with 128 GB of DRAM and 512 GB of Intel Optane DC (128 GB per socket). It runs Linux 4.19 with gcc 8.3.0, glibc 2.28 and Hotspot 8u232-b03 (commit c5ca527b0afd) configured to use G1. The patch for Hotspot that adds the three NVMM-specific instructions to `Unsafe` (namely, `pwb`, `pfence` and `psync`) contains 200 SLOC. Besides this patch, J-NVM, J-PDT and J-PFA that all together implement our NVMM object-oriented programming framework, encompass about 4000 SLOC.

NVMM runs in App Direct mode and is formatted with the ext4 file system. In this mode, software has direct byte-addressable access to NVMM.

Infinispan Our experiments use Infinispan, an open-source industrial-grade data store maintained by Red Hat. Infinispan exposes a cache abstraction to the application that supports advanced operations, such as transactions and JPQL requests. We use Infinispan version 9.4.17.Final [229], which contains around 600,000 SLOC (see Table 3.1). Infinispan runs either with the application (embedded mode), or as a remote storage (server mode). Unless stated otherwise, we use the embedded mode during our experiments and cache up to 10% of the data items. As seen in §3.2.1, a larger ratio would significantly harm performance. Accordingly, we also cap the volatile heap to 22 GB. This size gives the best performance with our YCSB workload on a file system backend atop NVMM (precisely, less then 3.7% of the total time is spent in GC in Figure 5.1).

For each experiment, we report the average over at least 6 runs along with the standard deviation.

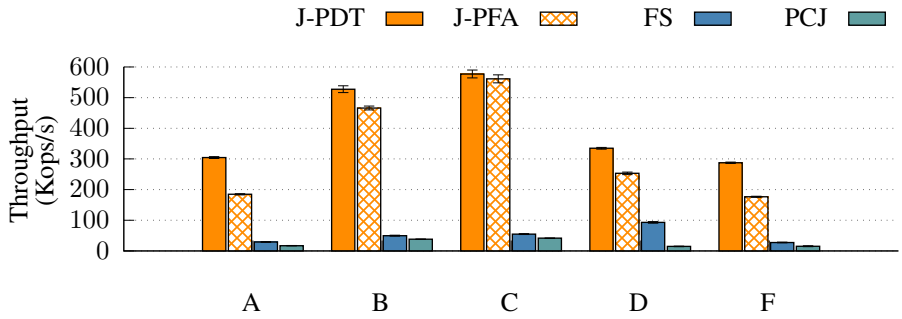


Figure 5.1: The YCSB benchmark.

Persistent backends We evaluate different NVMM-ready persistent backends for Infinispan: ***(J-PDT)*** A backend using the J-PDT standalone library. ***(J-PFA)*** A backend built with the failure-atomic blocks of J-NVM. ***(FS)*** The default file system backend of Infinispan using NVMM formatted in ext4. ***(PCJ)*** An implementation that relies on the Persistent Collections for Java library [16]. PCJ uses the native PMDK 1.9.2 library [19] through the Java Native Interface. For reference purposes, we also consider the following dummy backends without persistence: ***(TmpFS)*** A file system stored in volatile memory. ***(NullFS)*** A virtual file system that treats read and write system calls as no-ops [27]. ***(Volatile)*** A configuration in which persistence is simply disabled. Volatile behaves as NullFS, except that the marshaling/unmarshaling phase is avoided.

The persistent backend using PCJ is 274 SLOC long. The J-PFA and J-PDT backends use the same code base which contains 271 SLOC.

5.2 YCSB

Benchmark. We compare J-NVM against the other approaches by running version 0.18 of the Yahoo! Cloud Serving Benchmark (YCSB) [99] YCSB is a key-value store benchmark that consists of six workloads (A to F) with different access patterns. A client can execute six types of operations (`read`, `scan`, `insert`, `update` and `rmw`) on the key-value store. Workload A is update-heavy (50% of `update`), B is read-heavy (95% of `read`) and C is read-only. Workload D consists of repeated reads (95% of `read`) followed by insertions of new values. In the workload E, the client executes short scans. Workload F is a mix of read and read-modify-write (`rmw`) operations. We evaluate all workloads except E. Infinispan only provides `scan` through the JPQL interface, hence workload E is not comparable with the others that use a direct interface. If not otherwise specified, YCSB executes in sequential mode (single-threaded client).

YCSB associates a key with a data record that contains fixed length fields. Unless otherwise stated, we use the default parameters of 3M records, each having 10 fields of 100 B. YCSB runs with the default access patterns (namely, zipfian and latest). Compared to a uniform distribution, these patterns improve the cache hit ratio, and makes thus the FS backend more efficient.

J-PDT, J-PFA and PCJ all require to use persistent keys and values in YCSB. To achieve this, we modified the Infinispan client, which represent less than 30 SLOC from the vanilla version.

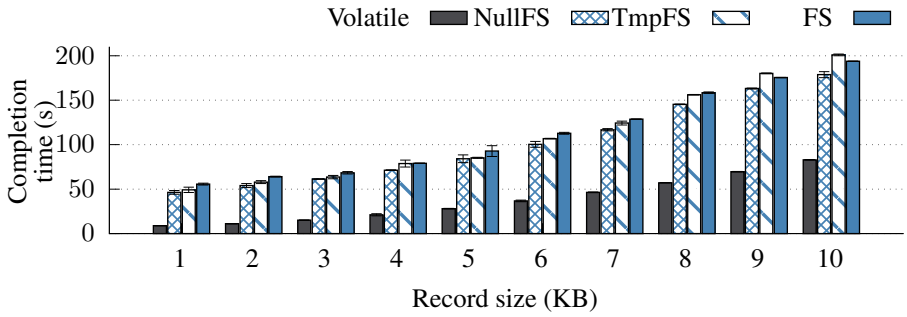


Figure 5.2: The price to access NVMM from the file system.

Results. Figure 5.1 presents the throughput of the YCSB benchmark with the different persistent backends. In this figure, we observe first that J-PDT systematically outperforms the other approaches. Except in workload D, J-PDT is consistently 10.5x faster than FS. In comparison to PCJ, the difference ranges between 13.8x and 22.7x faster. In workload D, J-PDT executes at least 3.6x more operations per second than FS and PCJ.

The low performance of FS comes from the cost of marshaling persistent objects back and forth between their file system and Java representations. Figure 5.2 highlights this phenomenon. In this figure, 1 KB corresponds to Figure 5.1. Compared to Volatile, the three file system backends (NullFS, TmpFS and FS) have similar performance. The completion time is between 2.11-6.26x higher than the volatile base line. In particular, NullFS, which fully ignores reads and writes, is just slightly faster than FS. This shows that the main cost comes from data marshaling and not from the file system itself.

In Figure 5.1, the lower performance of PCJ is due to the Java native interface that requires heavy synchronization to call a native method [249]. J-NVM avoids this cost by leveraging the `Unsafe` interface (§2.8.2.4, §4.4.8.1), which does not have to synchronize the whole JVM to escape the Java world.

Overall, the results in Figure 5.1 outline that NVMM drastically changes the way to access persistent data from the Java runtime: while JNI calls or marshaling/unmarshaling operations were negligible with slow storage devices, this is no more the case with NVMM. They must be avoided where possible.

In Figure 5.1, J-PFA also systematically outperforms FS and PCJ for the same reasons as mentioned above. Nevertheless, J-PDT is still up to 65% faster. This result shows that hand-crafted crash-consistent data structures can be more efficient than a generic approach.

5.3 Performance analysis

This section provides a comprehensive analysis of the performance of J-NVM. We detail the importance of the workload, how J-NVM scales with the number of threads, the time to recover from a crash and the performance of the low-level interface.

5.3.1 Sensitivity to the workload

In what follows, we analyze how J-NVM reacts to workload variations. We use the settings presented in §5.1, but change one parameter at a time. Figure 5.3 presents our

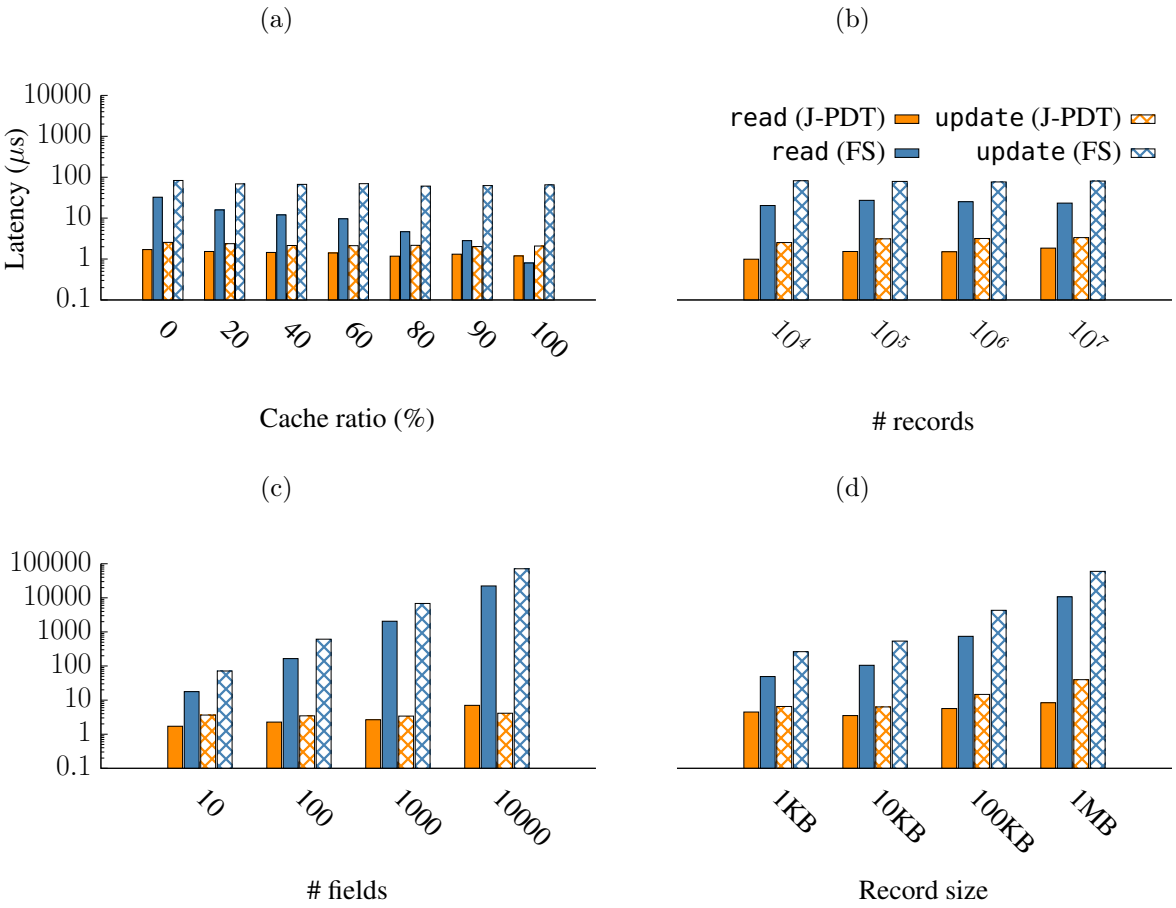


Figure 5.3: Impact of the cache ratio, the number of records, their composition and size (from left to right).

results with YCSB-A. This figure reports only the performance of J-PDT and FS since, as presented earlier, they are respectively faster than J-PFA and PCJ.

Caching. Figure 5.3a measures the impact of caching persistent data in Infinispan. In this figure, we observe that changing the cache size does not much impact the performance of J-PDT. This observation holds for both reads (from 1.7 μs to 1.2 μs) and updates (from 2.6 μs to 2.1 μs). With J-PDT, only proxies are kept in the cache. In particular, J-PDT never marshal/unmarshal the persistent data structures themselves. As caching brings almost no performance benefits, it is disabled in all our experiments using J-NVM as a backend for Infinispan.

For FS, improving the cache size has almost no performance impact on updates. This comes from the fact that, as Infinispan uses a write-through policy for durability, updates need to access the file system in the critical path. On the contrary, having a larger cache benefits to reads (from 32.5 μs to 0.8 μs) . At 0%, a read systematically fetches data from the file system, which becomes less and less likely when the cache size increases.

With a cache of 100%, J-PDT pays the cost of reading NVMM through proxies, while FS directly uses volatile objects stored in the cache. As a consequence, FS is slightly

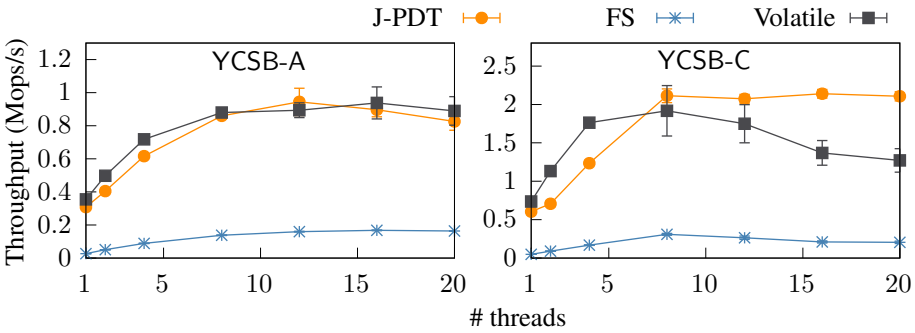


Figure 5.4: Multi-threaded performance.

better than J-PDT for reads in this case (0.8 μs versus 1.2 μs). In this experiment, the dataset is small (3 GB) and thus garbage collection has a limited impact on performance. As underlined in §3.2.1, this caching policy would be problematic with a larger dataset.

Number of records. We now turn our attention towards the impact of the dataset size on performance. Figure 5.3b presents the access latency when the number of records increases. Overall, we observe that the performance of both J-PDT and FS is stable. The number of records does not impact performance because each operation works on a single record at a time. Furthermore, as neither J-PDT nor FS use a GC to collect the persistent state, the overheads of Figure 3.2 are avoided.

Record composition In this experiment, we consider additional fields (Figure 5.3c) and larger fields (Figure 5.3d). In both cases, we adjust the number of records to keep a constant dataset size.

We first observe that changing the record composition only moderately impacts the performance of J-PDT. For reads, the latency grows from 1.7 μs to 7.0 μs with more fields (Figure 5.3c), and from 2.4 μs to 4.0 μs with larger fields (Figure 5.3d). With updates, the latency grows from 3.6 μs to 4.1 μs with more fields (Figure 5.3c), and from 3.2 μs to 14.6 μs with larger fields (Figure 5.3d). This slight increase in latency comes from the fact that J-NVM has to resurrect more fields, or larger ones.

In Figures 5.3c and 5.3d, the read performance of FS significantly degrades when the number of fields increases (from 17.7 μs to 22.3 ms in Figure 5.3c). This is also the case when the size of each field increases (from 17.5 μs to 1.6 ms in Figure 5.3d). Updates show a similar pattern: from 71.3 μs to 71.3 ms with more fields, and from 71.0 μs to 6.5 ms with larger fields. As in Figure 5.2, this degradation comes from the increasing cost of marshaling/unmarshaling voluminous records.

5.3.2 Multi-threading

This section evaluates how J-PDT behaves when the persistent objects are accessed concurrently. Figure 5.4 presents the throughput achieved in YCSB-A and YCSB-C using 1M records when the number of threads increases from 1 to 20. For both J-PDT and FS, accesses to the persistent state are protected by the locks of Infinispan. Notice that since Infinispan runs in embedded mode, a YCSB thread is also an Infinispan thread.

In Figure 5.4, the peak performance of J-PDT is slightly higher than Volatile in the two workloads. This surprising result comes from the increased pressure on GC in the

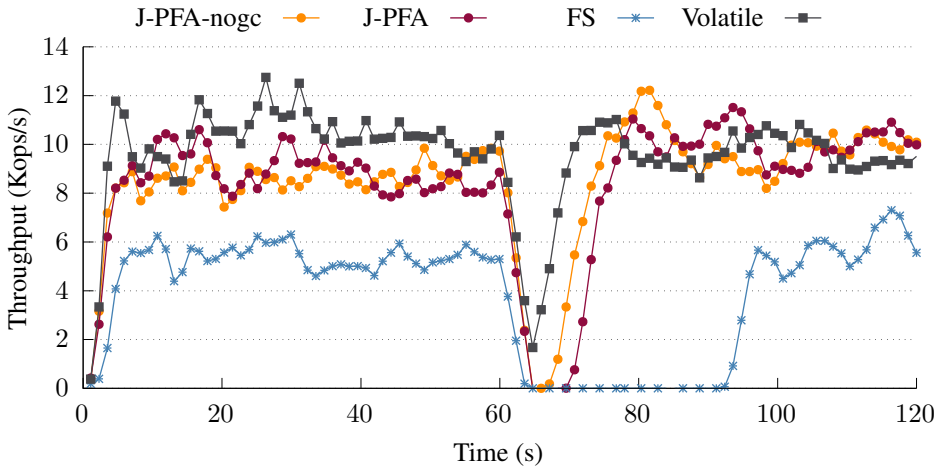


Figure 5.5: Recovery time with a TPC-B like workload.
Single REST client executing bank transfers (5.10^6 accounts). A fault is injected after one minute and the application server restarted. Throughput corresponds to a 5-second moving average.

Volatile implementation. In YCSB-A, J-PDT saturates Infinispan with 12 threads, while Volatile needs 16 of them. In YCSB-C, J-PDT and Volatile both saturate Infinispan with 8 threads. These results show that J-PDT, with its design based on proxies to access NVMM, does not introduce additional scalability bottlenecks with respect to the volatile implementation. Figure 5.4 also shows that FS, with a realistic cache ratio of 10%, scales up to 16 threads in YCSB-A and up to 8 threads with YCSB-C. In both workloads, at its peak performance, FS remains more than 5x slower than J-PDT.

5.3.3 Performance of the recovery procedure

In this experiment, we evaluate the time to recover from a crash failure. Figure 5.5 presents our results. We use a bank application inspired from the TPC-B benchmark [9, 134]. The bank server holds 10M accounts of 140 B each. It provides a single operation to execute a transfer between two accounts in a failure-atomic block. The server runs in a container and exposes a REST interface to remote clients. In Figure 5.5, the load injector continuously performs transfers between two randomly-selected accounts. It runs on the same machine as the bank server.

After a minute, the container holding the bank server is crashed with SIGKILL, then immediately restarted. Volatile, which only stores the state in DRAM, resumes processing requests after 2.4 s. Because the server restarts from a blank state, accounts are recreated on demand with a 0€ balance after recovery. Volatile is back to its nominal throughput (9.5K ops/s on average) 5.5 s after the crash.

J-PFA restarts processing requests 8.5 s after the crash, and returns to its nominal throughput (8.8K ops/s on average) a few seconds later. J-PFA needs 6.1 s more than Volatile to restart because the recovery procedure has to run the recovery GC over the 10M accounts. Volatile does not exhibit this cost because the bank server simply restarts from an empty state.

For completeness, Figure 5.5 also includes J-PFA-nogc. With J-PFA-nogc, the recovery procedure does not trigger the traversal of the object graph to delete invalid reachable objects. Instead, the recovery procedure only inspects each block, adding invalid ones to the volatile free queue (see §4.4.4). Avoiding the graph traversal is correct in this

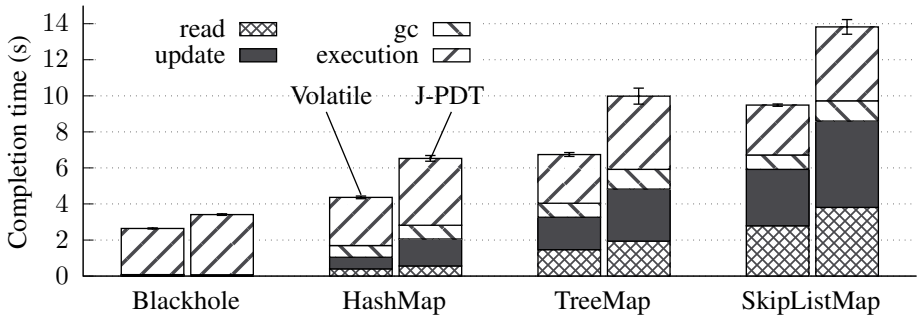


Figure 5.6: Persistent vs. volatile data types.

experiment because the application can not create invalid reachable objects: the server executes both the allocation and the insertion of an account in the database in the same failure-atomic block. We observe that without the graph traversal, J-PFA-nogc restarts processing requests 2.8 s faster than J-PFA.

In Figure 5.5, FS takes 28.8 s to restart and 34 s in total after the crash to return to normal (4.7K ops/s). This long delay comes from reconstructing eagerly the cache in memory. Upon restart, Infinispan reloads 10% of the accounts (1M) from NVMM. When this occurs, J-PFA pays a lower price because it creates proxies instead of reloading data in full.

5.3.4 Persistent data types.

In Figure 5.6, we compare the persistent maps available in J-PDT against their volatile counterparts in Hotspot (`java.util.*`). Three data types are considered: a hash table, a red-black tree and a skip-list map. In total, this code base covers 629 SLOC.

In Figure 5.6, we run YCSB-A directly on the data types themselves, without Infinispan. The "Blackhole" histogram in Figure 5.6 corresponds to an execution in which the operations are not applied. In other words, this histogram measures the time spent by the benchmark to inject the workload. Figure 5.6 shows that J-PDT is 45-50% slower than a volatile implementation. The rationale behind this drop of performance is the following: *(i)* J-PDT handles crashes which requires `pfences` in the critical path; *(ii)* NVMM is slightly slower than volatile memory [175]; and *(iii)* J-PDT relies on proxy objects to access NVMM.

5.3.5 Block size and NVMM access performance

During our experiments, we use a block size of 256 B. We measured that this size provides the best overall performance, because NVMM uses internally also a cache line of 256 B. A YCSB record contains 10 fields. With small fields (100 B) the NVMM space lost due to the block headers and the internal fragmentation accounts for 21.2% per record. This reduces to 9.4% with larger fields (10 KB).

Table 5.1 presents the throughput to access blocks of 256 B using J-NVM and C. For writes, the benchmark triggers a `pwb` after each CPU cache line (64 B) and it executes a `pfence` after a full block. In Table 5.1, J-NVM is at most 24% slower than C, except with random reads where it is 2.8x slower. These results show that the `Unsafe` interface allows most of the time to access NVMM at nearly native speed.

	Sequential		Random	
	Read	Write	Read	Write
J-NVM	3.21 GB/s	0.74 GB/s	0.71 GB/s	0.38 GB/s
C	4.01 GB/s	0.78 GB/s	1.94 GB/s	0.40 GB/s

Table 5.1: Access to a persistent 256 B-long block.

5.4 Summary

In this chapter, we experimentally validated the design of J-NVM, as able to efficiently access and manage NVMM from Java.

Experimental setup. (§5.1) We evaluated the performance of J-NVM by implementing several durable backends for the Infinispan data store [229]. Namely, two backends using J-PDT and J-PFA, another one made with PCJ [16] (PMDK); that we compared to the existing file backend (FS). We used FS with an Ext4-DAX file system on a NV-DIMM device (NVMM block-mode), not to introduce any hardware variation.

Overall performance. (§5.2) We measured latency and throughput in crash-free executions with the YCSB benchmark, and recovery times with a TPC-B-like workload [9]. From these, we could attest that J-PDT consistently led to 10× increased throughput over FS, in a real-world industrial-grade database and across all YCSB workloads. Interestingly, J-PDT can be over 65% faster than J-PFA in write-heavy workloads, showing that hand-tuned persistence remains more efficient than generic approaches, even outside of micro-benchmarks.

Marshaling. We further showed in Figure 5.2, that the lower performance of FS came from marshaling persistent objects back and forth between file and Java representations. The performance impact of system calls or file system operations turned out to be negligible when compared to software procedures that Infinispan had to do upstream (marshaling).

Caching. (§5.3) The in-depth performance analysis of J-NVM led us onto some notable insights. With J-NVM, the data is both durable and CPU addressable from a unique copy, therefore, no data movement are necessary during normal operation. (§5.3.1) Incidently, we observed constant operation latencies when varying: *(i)* the DRAM caching ratio, or *(ii)* the dataset size – number of records or different record compositions. Precisely, the gap in read latencies from Figure 5.3a, between J-PDT (NVMM-direct reads) and FS (DRAM-direct reads[1]), is only 0.4 μs. For such a small delta, DRAM caching could be considered obsolete with NVMM. In turn, by eliminating caching, operation latencies become resilient to variations to the size of the data set, or in the access distribution (uniform, zipfian, etc).

Multi-threading. (§5.3.2) Multi-threaded performance reached saturation between 8-to-12 threads with J-PDT, similar to the DRAM-only execution. This shows that J-NVM does not impede the application's scalability, nor the parallel performance of

[1]FS directly reads from DRAM, since 100% of data is in the cache

Optane modules; which seemed to saturate with 8 threads as well for a single module – that we recall from the micro-benchmarks in §2.4.2.

Recovery. (§5.3.3) We measured the performance of J-NVM recovery in faulty executions. We compared here J-PFA and FS (10% cache) to a fully volatile backend that recovers nothing (Volatile). The database was populated with 10M objects (140B payload each). J-PFA took only 3sec more than Volatile to reattain its nominal throughput. When disabling the GC phase on recovery[2], J-PFA-nogc recovered 2.8sec faster than J-PFA, which is almost as fast as Volatile. This was expected: no data need to be copied over, recovery then only accounts for the GC pass[3], which turned out to be around 3sec for 10M objects. Conversely, FS took 29sec more than Volatile to get back to its nominal throughput. Reason being that it had to ingest 10% of the data eagerly before starting to process requests again.

Micro-benchmarks. (§5.3.4) We also ran micro-benchmarks that put head-to-head J-PDT against `java.util.*` volatile data structures. Turned out J-PDT was consistently 45-50% slower on a YCSB-A workload. Just within the ballpark of the performance gap between the two media: NVMM and DRAM.

(§5.3.5) We finally had a micro-benchmark that compared raw NVMM access performance from either Java through Unsafe or native language (C). J-NVM was slower only in the read phase, by 25%. Overall, `Unsafe` allows for near-native access speeds to NVMM in Java.

The conclusion we can draw on J-NVM from this performance evaluation, is basically, that it successfully allows NVMM to be accessed in managed languages (Java) with no performance penalty over native languages (C).

In the next and final chapter, we conclude this thesis with a complete summary of our work and suggestions for future research directions.

[2]Skipping GC on recovery is possible in this specific instance (without compromising correctness), since the workload is purely transactional and only uses J-PFA. No recovery is needed apart from sorting-out the redo-log. The allocator free-list is reconstructed from a linear (parallel) scan of the heap, simply inspecting "valid" bits in block headers.

[3]Recall that the GC pass is a massively parallel read-only operation to NVMM – rather favorable to the media.

Chapter 6

Conclusion

This thesis has presented J-NVM, a principled approach to directly access NVMM outside the Java heap with volatile proxies. Our evaluation using micro-benchmarks and the Infinispan data store shows that J-NVM delivers better performance than other existing solutions.

In this chapter, we conclude the document by first briefly summarizing its content, then mentioning the limitations of our contributions and future work.

It organizes as follows.

(§6.1) We recall background, motivation and related work behind our contributions.
(§6.2) We summarize our contributions, relative to the original problem statement.
(§6.3) We examine limitations of our approach.
(§6.4) We detail some suggestions for future research directions.
(§6.5) We present some last and closing thoughts.

6.1 Related work

Background. We saw firsthand in chapter 2, that managing persistent data structures directly from the application through mapped files is an old subject largely studied in operating systems literature [58, 218, 250, 296, 328] (§2.2, §2.3). This line of research was fully renewed with the arrival of NVMM (§2.4). Several file systems tailored for NVMM exist (e.g., NOVA-Fortis [332], SplitFS [184] or Strata [203]). As seen in §5.2, using NVMM as a file system leads to costly marshaling/unmarshaling operations.

NVMM challenges. The full potential of NVMM and its numerous benefits can only be unlocked by applications through direct-access. However, the persistent memory programming model, and low-level reasoning it requires, abidingly leads to brittle persistence in programs (§2.6). Any misplaced flush or fence instruction may put the whole data at risk, with silent bugs that can now cause permanent heap corruption. Even though expert programmers might feasibly handle these new responsibilities, ensuring consistency of NVMM content in the wake of faults or crashes remains an ordeal for their patience and skills. Even more so as we recall the latent complexity of candidate applications and use cases for NVMM (§2.5).

Consequently, NVMM tremendous potential will not shine until applications can soundly change the way they go about persistence to easily adopt persistent memory. This necessitates programming support from languages and tools, for failure-atomicity or recovery of memory objects.

While presenting unique opportunities for fine-grained persistence, NVMM direct access also burdens programs with the integrity, consistency and validity of their data.

NVMM in native languages. Lot of prior research focus on offering to the developer a transactional interface inspired by databases [19, 92, 102, 103, 143, 232, 310] (§2.7.3). Others directly deal with the low-level NVMM semantic [265, 266] to implement specific data types [57, 138, 159, 174, 223, 354] (§2.7.1). A third category of works propose recipes to build persistent data types upon the prior knowledge about volatile constructions [159, 174, 208, 349] (§2.7.2). Overall, these works show that general techniques and hand-tuned persistent data types have their pros and cons. In particular, as confirmed by our comparison between J-PFA and J-PDT, failure-atomic blocks are easy to use but often less efficient than hand-tuned data types (§5.2).

NVMM in Java. All these prior works target native programming languages and they cannot be readily used in a managed language such as Java. Some recent efforts try to fill this gap by using an integrated design [143, 288, 329], or an external design [16, 20]. We discuss Espresso [329], AutoPersist [288] and Go-PMEM [143] in §3.2 and §3.3. We show experimentally that their approaches lead to collecting very large datasets, which negatively impacts performance. In chapter 5, we evaluate PCJ [16], which is in essence similar to LLPL [20], and show that it performs less efficiently than a file system interface.

Off-heap in Java. Regardless of NVMM, several applications avoid running a GC by storing part of their datasets outside the heap. This is notably the case of modern data stores (e.g., Spark [343] and Cassandra [204]). Apache Arrow [1] aims at addressing the problem of making such data structures portable. In [231], the authors propose an efficient volatile lock-free off-heap map. The map allows to modify directly off-heap objects, that is, without copying data between the on- and off-heap spaces. Contrarily to J-NVM, it only manipulates arrays of bytes, and thus requires to marshal/unmarshal its content.

6.2 J-NVM: Scalable, Safe and Quick Persistence in Java

This thesis has focused on the fundamental problem of providing support for NVMM direct-access in managed languages. Java and other high-level object-oriented programming languages are largely used for their productivity gains, and are especially popular choices in intensive data-processing programs. We deemed, in §2.8, that proper support for NVMM should neither restrict the expressiveness of the language, nor be limiting NVMM relative to its performance characteristics.

Core idea. Our key insight is to keep persistent data (NVMM-resident) *outside* the Java heap to avoid costly garbage collection, while preserving NVMM direct access (with `Unsafe`). Our contributions propose a novel "decoupling" principle for Java objects, that separates the data structure of a persistent object from its methods. This principle is key to run dedicated memory management in the NVMM heap. The use of a representant "proxy" object, unifies the representation of persistent objects under Java's object model. Proxies are data-less and ephemeral (instantiated lazily) on the Java heap, hence they put low-pressure on the GC.

Contributions. We presented four contributions in this thesis: J-NVM, J-PDT, J-PFA and JNVM-Transformer. Put together, they form a complete system that hides the complexity of NVMM programming behind regular Java objects. In detail:

- **J-NVM.** The low-level and core component of the framework. It is a pure Java library, with a builtin NVMM memory allocator, and methods to access NVMM locations or insert persistence instructions. It implements the decoupling principle, that is, the bare logic to instantiate or destroy persistent objects and efficiently access their fields.

- **J-PDT.** A high-level library of recoverable data structures. They are hand-crafted collection types for NVMM (e.g., arrays, maps, trees) which provide failure-atomic operations. Internally, the logical state of the structure is maintained in a volatile (DRAM) index from `java.util.*` that only holds on proxies; and a persistent set type materializes the data tuples in NVMM.

- **J-PFA.** The high-level support for failure-atomic blocks of code (FASEs). They enable any code sequence to execute atomically with respect to concurrent crashes. We implemented them with per-thread redo logs that work at a 256B granularity.

- **JNVM-Transformer.** The post-compilation tool that generates persistent objects and proxies from annotated Java objects. It saves a lot of manual effort that would otherwise be necessary to code proxy classes. In a nutshell, it restructures a POJO to comply with our framework and injects the appropriate J-NVM library calls for data manipulation and life cycle management.

Our evaluation on the Infinispan data store with the YCSB benchmark showed that J-NVM had superior performance all across the board, by at least one order of magnitude. Since detailed summaries of our contributions are already available at the end of preceding chapters (§3.7, §4.8, or §5.4); in what follows, we propose to examine how J-NVM fulfilled the core challenges we identified for language support of persistent memory in §2.8.

1. **(User) Separation between transient and persistent data.** J-NVM exposes a class-centric model, meaning all persistent objects are clearly identified with specific types. In detail, all classes that implement the `PObject` interface have persistent instances. Therefore, programmers can very easily verify off-line whether an instance is persistent, simply by looking at its class definition. Conversely, in *instance-centric* models, such as Espresso, the programmer has to backtrack to the object's allocation site and check whether it was instantiated with a `pnew`. In systems with *transitive persistence* (§2.2.4), such as AutoPersist, it becomes almost impossible to statically identify whether an object is durable or transient. Programmers would have to trace the whole flow graph of an instance and look for whether it was ever attached to an object that was itself (in)directly reachable from a persistent root. In all, the *class-centric* model of J-NVM saves a lot of hassles from our perspective. Generally speaking, clear and available information at glance leads to less coding bugs.

2. **(User) Identification of consistent program states.** With J-PFA, or essentially all systems proposing FASEs, inconsistent (transitional) program states are identified by programmers and encapsulated with transactional semantics. Our implementation of FASEs do not avoid the *double write problem*, but at least, incurs logging

overheads only per 256B blocks mutated – and not every single 8B stores. In comparison, Espresso, AutoPersist or Go-PMEM all use undo logging, which aggressively emits persistence instructions: 1 flush-fence pair for each new log entry.

3. **(System) Easily integrate in existing applications.** The *class-centric* model of J-NVM prohibits re-use of existing types, meaning parts of an application code base have to be rewritten for persistent objects. In situations where only some types have to permanently become persistent, JNVM-Transformer can automatically enhance these class definitions. *Orthogonal persistence* and *persistence independence* (§2.2.4), as found in Espresso or AutoPersist, seem more easily applicable on first thought. On second thought though, this is only the case for conversion of volatile code in applications. Whereas most real-world applications waiting to benefit from NVMM already use conventional file system calls for persistence. Even with *integrated* approaches, these programs would still need significant re-design efforts to integrate NVMM.

4. **(System) Prevent dangerous persistent to volatile references.** Again, the *class-centric* model of J-NVM statically forbids dangerous cross-heap references. Any volatile reference in persistent objects have to be marked as `transient`. This way, they are stored in the proxy but not in the NVMM data structure. In doing so, they remain volatile-to-volatile references, and persistent objects are free to initialize them in `recover()` or proxy "reconstructor". Note that the dichotomy between proxy fields and NVMM fields is entirely transparent when working with persistent objects. We discussed in §2.8.4.1 how Espresso proposed to nullify volatile fields on recovery, or in §2.8.4.2, how AutoPersist does not solve the issue. Instead of preventing these bugs from happening, AutoPersist migrates volatile objects referenced by persistent objects to the persistent heap. In all, it trades in "wild" pointers on recovery, for more frequent memory leaks – a bug for a bug.

5. **(System) Avoid dangling references or memory leaks on recovery.** The recovery-time GC of J-NVM can make up for persistence bugs originating from crashes concurrent to object free or references de-installation. The atomic-update utility method (§4.4.7) further covers for dangling pointers resulting of reference updates outside of FASEs. Competitor systems (e.g., PMDK) usually prevent these issues with transactional allocation, transactional pointer update, or with the alloc-to/free-from specific allocator API. These mechanisms often require a larger number of flushes and fences than with garbage-collection. Though, online GC comes with other limitations on NVMM.

6. **(System) Lightweight management that does not restrict NVMM characteristics (bandwidth, access latency, heap size).** J-NVM can perform most of its memory management operations without persistence instructions, or even any access to persistent metadata. NVMM allocation, free of persistent object, or even reference updates, need not to account for potential inconsistencies in heap metadata following a crash. It is the recovery-time GC of J-NVM that fixes these crash inconsistencies, and permits heap management operations not to issue persistence instructions (for heap consistency purposes) in crash-free executions. In all, little access if any at all to NVMM for heap management operations. Conversely, Espresso, AutoPersist and Go-PMEM all employ online garbage collection that severely impedes the amount of live data on NVMM heaps. Moreover, *orthogonal persistence* in AutoPersist requires dynamic instrumentation of Java bytecodes, which has a deterring cost as well.

7. **(System) Builtin FASEs & PDTs do not limit users in implementing their own.** With J-PFA and J-PDT, J-NVM offers both. Additionally, the low-level interface provides building blocks for programs to assemble their own data structures if needed. In comparison, Espresso, AutoPersist or Go-PMEM only provide FASEs.

Moreover, their low-level interface is hardly usable, if at all. They tend to group `pwb` with `pfence` in composite operations, which is less malleable and actively restricts custom PDT implementations.

8. **(System) Allow recovery-time code to clean inconsistencies in both system and user (meta)data.** Recovery is a central concept in J-NVM. No data can be considered persistent if they can not be re-instantiated in subsequent executions. The reason for which we offer to each persistent object to implement an optional `recover()` callback method. J-NVM orchestrates their invocation with its bootstrap procedure: during the graph traversal of our recovery-time GC (after processing log entries). With this design, persistent objects are responsible for their own NVMM data structure, and are given the opportunity to correct their state early-on in the heap bootstrap process. Moreover, the "reconstructor" of proxies is a good place to re-initialize any soft-state (transient) from durable data. In other terms, the proxies' "reconstructor" easily accommodates for (reconstructible) transient (meta)data in persistent objects, and the `recover()` method, for algorithms whose inconsistencies are fixed at the beginning of next executions. In competitor frameworks, recovery is often, if not always, overlooked. No equivalent handles for recovery are provided to objects. In most cases, that is because they assume all application code is made failure-atomic with FASEs, or similar. In that case, only the log needs to be recovered. This assumption of course restricts the class of algorithm and protocols that can be implemented within these frameworks. J-NVM and its design do not have such limitations, and could hypothetically let users implement their own protocols for failure-atomicity for instance.

To conclude this section, our contributions are not just more efficient than previous art to access NVMM in Java, as we just discussed, but they also perfectly fit our original problem statement. In particular, they *(i)* do not induce significant software overhead when accessing persistent memory locations, *(ii)* are not limiting in respect to NVMM large capacities, *(iii)* have minimal performance impact in crash-free executions, *(iv)* are safe and clear for programmers, *(v)* abstract the complexity of failure-atomicity with object-oriented idioms.

6.3 Limitations

Nonetheless, J-NVM parts from Java's object model a few key points: *(i)* strong persistent types, *(ii)* online garbage collection, *(iii)* extended object life cycle. Although we motivated why we had to diverge from Java's model, these differences can still appear as weaknesses since they limit code re-use. We now remind how we make these choices tolerable to programmers.

Limited code re-use. J-NVM does not support *orthogonal persistence* or *persistence independence* (§2.2.4) for a lot of good reasons we previously mentioned. On the surface, that might appear as limiting to re-employ legacy codes, but recall that in any case, properly turning volatile code for persistence is more involved than simply making object allocations persistent. In particular, persistent data have unique life cycles, and Optane modules exhibit singular performance patterns. With *transitive persistence* (§2.2.4), in AutoPersist, programmers even have to manually partition persistent data by tagging volatile object fields with `@unrecoverable` annotations. This step is necessary to avoid transitively pulling all data on the persistent heap.

Our point is that, neither J-NVM nor *orthogonal persistence* can *effortlessly* port existing code to NVMM. However, we reduce the coding burden with JNVM-Transformer, that lets programmers write regular Java objects and automatically converts them to persistent objects. Our off-the-self persistent types (J-PDT) also contribute to easing software development, with drop-in replacements for common JDK collections and types. The fact that we could under 200 lines of code implement a J-PDT backend for Infinispan, is a good testimony to the effectiveness of these programming abstractions.

We could perhaps go as far as stating that volatile code re-use for persistence on NVMM is a pipe dream. Consequently, a well-thought programming model for persistence prevails over *orthogonal* approaches whose only *forte* is code portability.

Explicit free. Compromising on online garbage collection is a decision which may seriously bug Java users. We remind them that garbage collection is not a silver bullet to avoiding memory leaks: they may still happen when developers forget to *nullify* references. Specifically, garbage collectors approximate object liveness by reachability, and leaks might still result of no longer accessed objects that are yet still linked in the graph [68]. In that regard, *nullify*-ing a reference or calling a *free* method sounds fairly similar. At the exception of the latter requiring users to manually keep track of incoming references to avoid double frees or dangling references. All considered, we emphasize that explicit deletion has been the only common way of disposing of durable data in mainstream persistence facilities – file systems or databases alike.

We now relate more substantial limitations. We must mark though, these are not specific to J-NVM but broadly affect NVMM persistent heaps. The first one is tied to verifying crash-consistency of applications, the second, to interoperability of durable data.

Testing tools. As with file systems [234, 290], or key-value stores [69] verifying crash-consistency of NVMM programs is a topic of uttermost importance. In contrast with other forms of storage, persistence bugs on NVMM are simultaneously more endangering for the data, and harder to catch. In cause, the interleaving of persistence instructions is immensely harder to get right than file calls. Moreover, NVMM is accessed through one giant `mmap` and managed with custom user-level allocator; leaving powerless OS mechanisms for memory protection. Only rare NVMM libraries prevent (some of) these bugs. For instance, **Corundum** [166] extends static checking from the type system and memory management idioms of the Rust language to persistent memory. **Carbide** [165, 167] then works as a bridge between Rust and C++ while preserving Corundum's safety guarantees.

A slew of papers proposed, as a generic solution, tools to proactively detect crash-consistency issues or memory safety violations. Techniques vary: test case generation [89, 220], fault injection [150, 219], injection of online sanity checks [71], inferring of correctness criteria and output validation [141], symbolic execution [244], dynamic instrumentation [122], or model checking [148, 149] (persistency models seen in §2.6.1.2)[1].
Nearly all of them (worryingly) found new persistency bugs in the PMDK, showing that even industry-standard NVMM libraries developed by experts are not exempt of programming mistakes.

[1]Through x86 persistence semantics simulators [148], based on the $Px86_{sim}$ model from Raad et al. [266] and a later refinement by Khyzha et al. [194].

Currently, no such tool focuses on Java. Adapting them for managed languages is not straightforward. They extensively rely on binary instrumentation, or plug on native language compilation toolchains. As we mentioned in §2.8.1, conventional Java virtual machines embed their own JIT compiler and produce binaries on-the-fly. We are confident those ideas in the literature may be usable with Java as well, yet, existing tools are not directly applicable.

Data sharing. As seen in §2.1, durable media were not only used for data persistence, but also portability and interoperability. In essence, files and databases have been used for the very purpose of exchanging or sharing data between programs. However, common NVMM libraries persist data on a recoverable heap, without providing any mean of interacting directly with other programs' heap and data. Protocols for safe and efficient sharing of persistent heap data between multiple applications have yet to be presented. One work comes very close though. **UniHeap** [213] is a shared heap for multiple programming languages. It is externally managed (plugs in different language runtimes) and proposes an unified persistent object model that enables object sharing across the supported languages. If NVMM is to be employed on a large scale, outside of specialized programs, this is an extremely important issue to address.

6.4 Future work

Subsequent research directions do not limit to solving the last two significant shortcomings of persistent heap we just mentioned. Rather, the only logical outcome of having produced a persistent programming library would be to use it to build a "killer app" for NVMM and Java.

Recall the most significant use cases of NVMM we detailed in §2.5, and that initially, our interests were of enhancing big data storage or processing applications with NVMM. Since then, we understood in §2.5.3 that NVMM in databases would only unrestrict storage accesses, but that the overall performance would remain capped by network I/Os. In that section, we shed light on two distinct possible directions: *(i)* wait on faster networks to be available, and perhaps, prototype using Infiniband and RDMA, or *(ii)* leverage NVMM as embedded storage only. We project a contribution on that second idea, but still detail opportunities and prospects for both.

Rack-scale distribution and sharing. CXL (*Compute Express Link*), the new industry-pushed open standard for high-speed and low-latency device interconnects we named in §2.4.2, could be the fast network we lack today to further accelerate storage in distributed settings. CXL is more than just faster networking interfaces, and will definitely raise new interesting challenges. In detail, from specifications, its memory interface will propose a cache-coherent and hardware-mediated disaggregated shared memory. The CXL controller device would apparently directly be connected to the memory controller, such that it can transparently mediate CPU load/store instructions to remote DIMMs exposed by other machines in the rack.

Stateful serverless computations. Function-as-a-Service (FaaS) platforms and "serverless" infrastructures are becoming increasingly popular as cloud computing paradigms for the simplicity they offer. Users only have to provide their code artifacts as *cloud functions*, and not to worry about the executing servers, virtual machines, containers, or

their configuration. Instead, cloud functions are executed by the platform on a black-box FaaS runtime. The simplicity of this programming model is even meliorated by the humongous amount of physical resources offered by cloud providers, allowing near unbounded elasticity.

Currently though, FaaS offerings only support stateless functions with limited I/Os. This is ample for short-lived jobs, but restricting to use FaaS as a more generic computing platform. For instance, consider that functions dealing with some external state need to: first retrieve it from a remote database, then instantiate it locally and perform their operations, to finally push it back to the database. For data-intensive applications, these extra transfers and round-trips lead to reduced performance compared to serverful deployments. For these reasons, cloud functions are less appealing when dealing with external states. On one hand, short-lived jobs suffer from increased response times due to lengthy retrieval of external data. On the other hand, heavy data processing programs are more excessive on network bandwidth, which is usually highly priced by public cloud providers.

Some research papers explore stateful serverless. **Cloudburst** [300] is a prototype FaaS runtime, with a distributed in-memory database cache attached to computing nodes. The cache itself relies on Anna KVS [326] to be causally-consistent, but remains volatile (DRAM-only). **Beldi** [347] is library for AWS Lambdas [3] that intermediates accesses to Amazon's DynamoDB [118]. It essentially provides transactional semantics to cope with data races across functions, or potential inconsistencies left by faulty executions. **Palette** [26] proposes to bring computations closer to data – the opposite of currently applied patterns – with load balancing techniques aware of data locality.

After having developed J-NVM, NVMM almost seems like a perfect fit for serverless ephemeral functions with stateful computations. The low recovery latencies of persistent heaps on NVMM would allow quick resumption of hot or cold functions alike. Our insights were that with J-NVM, persistent data and states could be embedded in cloud functions with no penalty. Put together with a FaaS runtime that has data locality awareness – as in Palette [26] – functions could be scheduled and invoked on appropriate nodes. In doing so, we see that we could almost entirely avoid data transfers. In instances where data might be scattered across multiple nodes, we would then split computations into multiple smaller functions, until no inter-node data dependencies exist. In all, J-NVM and NVMM both seem to have potential to embed local data in ephemeral cloud functions. Assembled with a locality-aware load balancer, network bandwidth usage should be largely reduced in stateful FaaS workflows.

6.5 Closing thoughts

This thesis has shown a system design and programming model for efficient support and use of NVMM in managed languages. Our thought process was guided by practical observations rather than theoretical modeling. As a result, the programming interface to NVMM in J-NVM is almost a no-model approach. That is, we did not question the best possible integration for the notion of persistence in object-oriented languages. Rather, we extended the Java object model for seamless recovery of specifically "tagged" data types (`PObject`).

Explicit persistence. This approach was also motivated by the fact that anterior schemes with more convoluted ways of weaving persistence into existing software never

piqued interest of the industry. Namely, the transparent *persistent processes* in single-level stores (§2.2.1, §2.3.2.1) or *orthogonally persistent* object systems (§2.2.4, §2.3.2.2). Data might just be too precious for applications to forfeit control of their persistence to obscure system software. Alternatively, when whole-states are persisted, or the line between transient and permanent data is blurred, programs also have a difficult time defining which data must be cautiously recovered. Rebooting have been the ancestral way of healing transient bugs [79], but persistent inconsistencies always had to be manually fixed. In conclusion, we believe data models for easier (or transparent) persistence will remain impractical so long they do not help identify persistent data and ease their recovery as well.

Programming flexibility. We were genuinely surprised by the mixing of J-NVM programming model with Java's object-oriented programming. The use of strong types for persistence never restricted us, and Java's idioms provide pleasant flexibility that led us onto some interesting design patterns. Perhaps the most telltale examples are: *(i)* lookups with volatile objects in persistent hash maps (§4.5.2), and *(ii)* the implementation of AutoPersist's object graph migration technique in a specific PDT (§4.5.2).

The first example actually leverage the fact that Java's object equality works irrespective of types. Therefore, a `PString` is not prevented from implementing its `equals()` method to check equality with `java.util.String` objects.

For the second one, we created a `Convertible` interface that we use to bind a volatile and a persistent type together. Both objects have to declare a conversion method from volatile to persistent and vice-versa. The PDT can then automagically persist a volatile type that implements `Convertible` (and its whole sub-graph) when it crosses the persistence frontier. That is, persist it when first inserted in the PDT, or convert it back to its volatile representation when it is removed from it.

In all, we have not found J-NVM to have limited programability in any way due to its explicit persistent types, but were instead pleasantly surprised by possible design patterns.

Non-intrusive solution. Perhaps one of the most interesting aspect of J-NVM is in its nature, as a self-contained user Java library. J-NVM implementation is not intrusive to Java virtual machines, which we think, is an essential factor for adoption of the system. We stress that this is not the case for any *integrated designs* presented in §2.8.4. As we understand, no industry in their right mind would replace components as crucial as OS or JDK with research forks or prototypes. Likewise, such projects, as sizable code base forks, are hardly ever merged back into the upstream. Patch sets are just too big, and submitted by anonymous contributors. For instance, the NOVA file system [332] never made it into mainstream Linux. They could not implement everything as a kernel module and had to patch kernel sources as well. In any case, implementing J-NVM as a set of JVM extensions would have been foolish and unnecessary. A user library in the end is far more versatile and portable. More importantly, it does not compete with legitimate software and organizations that industry spent years investing in and building trust on. In conclusion, J-NVM is not intrusive to language runtimes, as opposed to *integrated designs*, which is strongly superior for ease of deployment and integration in real world applications.

Appendix A

French Summary of the Thesis

Synthèse du rapport de thèse en français

As requested by the rules of procedure of IP Paris doctoral school (section 4.2: "Language of the thesis"), this appendix consists of the compulsory short summary of the thesis written in french.[1] *Conformément au réglement de l'école doctorale IP Paris, cette annexe contient la synthèse obligatoire en français du rapport de thèse.*

Résumé

L'arrivée de la mémoire non-volatile (NVMM) sur le marché propose une alternative rapide et durable pour le stockage de données, avec des performances considérablement accrues par rapport aux supports traditionnels, à savoir SSD et disques durs. La NVMM est adressable à la granularité de l'octet, une caractéristique unique qui permet de maintenir des structures de données complexes par le biais d'instructions mémoires standards, tout en étant résistante aux pannes système et logiciels. Néanmoins, gérer correctement la persistance des données est bien plus compliquée que de simples manipulations mémoire. De plus, chaque bug en NVMM peut désormais compromettre l'intégrité des données ainsi que leur récupération, et il faut donc prendre grand soin quant à sa programmation.

Ainsi, de nouvelles abstractions de programmation pour la persistance et l'intégration dans les langages et compilateurs sont nécessaires afin de faciliter l'usage de la mémoire non-volatile. Cette thèse se penche sur ce problème général. Nous expliquons comment intégrer la mémoire persistante dans les langages de programmation managés, et présentons J-NVM, un framework pour accéder efficacement à la NVMM en Java.

Avec J-NVM, nous montrons comment concevoir une interface d'accès simple, complète et efficace qui lie les spécificités de la persistance sur NVMM avec la programmation orientée objet. En particulier, J-NVM offre des fonctionnalités pour rendre durable des objets Java avec des sections de code atomiques en cas de panne. J-NVM est construit sans apporter de modifications à l'environnement d'exécution de Java, ce qui favorise sa portabilité aux divers environnements d'exécution de Java.

En interne, J-NVM s'appuie sur des objets mandataires qui réalisent des accès directs

[1] This is an idiotic requirement and a real struggle for non-native french speakers; which is why I decided not to rely on my french skills, but rather chose to complete it as our fellow non-french speaking foreign student would. As such, this summary was automatically translated from English to French, without further edits, using the DeepL web translator (https://www.deepl.com).

à la NVMM, gérée comme une mémoire hors-tas. Ce canevas fournit également une bibliothèque de structures de données optimisées pour la NVMM qui restent cohérentes à la suite de redémarrages ou d'arrêts impromptus.

Au cours de cette thèse, nous évaluons J-NVM en ré-implémentant la couche de stockage d'Infinispan, une base de données open-source de niveau industriel. Les résultats obtenus avec les benchmarks TPC-B et YCSB, montrent que J-NVM est systématiquement plus rapide que les autres approches existant à l'heure actuelle pour accéder à la NVMM en Java.

A.1 Introduction

Contexte général. Les magasins de données constituent l'épine dorsale des infrastructures informatiques modernes. Ils prennent en charge de vastes ensembles de données et permettent aux cadres de traitement d'extraire des informations de ces données. Ils sont conçus pour une réponse rapide et un calcul parallèle à une échelle sans précédent. Parmi les exemples récents de ces systèmes, on peut citer les bases de données en mémoire, les bases de données NoSQL et les magasins de valeurs clés. Le fait que les vitesses et les latences des dispositifs de stockage n'aient jamais rattrapé celles de la mémoire principale, accusant aujourd'hui un retard de plusieurs ordres de grandeur, a longtemps constitué un obstacle majeur à la mise en œuvre de ces systèmes, a longtemps constitué un obstacle majeur à l'architecture de ces systèmes.

Dans ces systèmes, bien que le traitement ait lieu dans la mémoire principale, la version des données qui fait autorité est conservée sur des dispositifs de stockage durables (SSD, disque) dont les temps d'accès sont nettement plus lents. Le maintien de la cohérence mutuelle de ces deux versions des données par des opérations synchrones sur les dispositifs persistants induit généralement de sérieux goulets d'étranglement en termes de performances et d'entrées/sorties. La persistance asynchrone est devenue très populaire pour cette raison, malgré les garanties de durabilité dégradées qu'elle offre et la complexité des algorithmes qu'elle requiert. En outre, en raison de cette dichotomie entre la mémoire et le stockage durable, les magasins de données sont condamnés à passer des tonnes de temps au redémarrage avant de pouvoir reprendre un fonctionnement normal. En cause, la nécessité de reconstruire les états précédents dans la mémoire principale, ou de repeupler un cache de données. Si cette phase de démarrage est si longue, c'est parce qu'elle implique la copie d'une partition des données du disque vers la DRAM, afin d'éviter la lenteur du traitement des requêtes juste après la récupération d'une défaillance du système.

L'avènement de la NVMM. En 2015, Intel a fait une annonce bouleversante pour sa gamme de produits *Optane Persistent Memory*. La première technologie *Non-volatile Main Memory* (NVMM) disponible dans le commerce et offrant un accès direct aux données sur le bus de la mémoire, identique à la DRAM, mais avec des capacités jusqu'à 8 fois plus importantes et, surtout, une non-volatilité des données résidentes, à l'instar des dispositifs de stockage. Une telle technologie, qui combine des accès et des performances de type DRAM avec la durabilité du stockage, pourrait redéfinir complètement le rôle et l'architecture des systèmes de stockage, en offrant des possibilités nouvelles et singulières pour la persistance des données à grain fin dans les applications.

Mémoire principale non volatile (NVMM), par leur avènement récent, ont également

remis en question la pertinence de l'abstraction de fichier traditionnelle et de son interface de persistance des données. Celles-là mêmes que les logiciels système exposaient depuis des décennies et à partir desquelles presque toutes les applications étaient construites. En effet, le simple fait que la technologie NVMM actuelle présente des latences d'accès brutes seulement 2 à 3 fois supérieures à celles de la DRAM, ce qui signifie une plage d'opérations comprise entre 100 et 300ns. Une plage que les systèmes de fichiers sont tout simplement incapables de fournir, souffrant de décennies d'accumulation d'optimisations encombrantes, en gardant à l'esprit les lecteurs mécaniques rotatifs, basés sur les secteurs, qui fonctionnent à l'échelle de la milliseconde, loin de l'échelle de la sub-microseconde. On peut encore constater une amélioration des performances par rapport aux disques SSD lorsqu'on utilise la NVMM comme support de stockage par blocs pour les opérations de fichiers, Cependant, le véritable potentiel des NVMM réside dans le mode d'accès direct.

En mode d'accès direct, les applications peuvent utiliser des instructions de mémoire pour accéder aux données résidant dans la NVMM et les gérer à des vitesses inégalées, en contournant complètement le sous-système d'E/S du système d'exploitation. En effet, les données résidant dans la NVMM sont directement adressables par le processeur, ce qui signifie que les produits Optane sont effectivement les premiers à pouvoir effacer la dichotomie mémoire/stockage dans les appareils de base. On peut imaginer que l'introduction de la NVMM dans les systèmes de stockage pourrait supprimer la double représentation des données et conduire à des temps de récupération extrêmement réduits. et conduire à des temps de récupération extrêmement réduits, ainsi qu'à une réduction et une simplification significatives de leurs bases de code, et enfin, à un meilleur débit global ou à des temps de réponse plus courts.

Inversement, l'accès direct impose également aux applications de nouvelles responsabilités en matière de persistance des données. La cohérence et l'intégrité des données persistantes, par rapport aux erreurs potentielles du système ou du logiciel, doivent maintenant être entièrement assumées par le code au niveau de l'utilisateur. En cas de panne, les piles logicielles du système ne seraient plus d'aucune aide pour garantir la récupération d'une version cohérente des données. Cela n'est pas facile à réaliser : comme on peut s'en souvenir, les architectures actuelles des machines se caractérisent par des caches volatiles de l'unité centrale et des modèles de mémoire détendus. Ce qui implique que, *(i)* rien ne peut empêcher que des données stockées dans des emplacements de mémoire soient expulsées des caches et atteignent prématurément le NVMM (*implicit flushes*), ou que *(ii)* le matériel dicte l'ordre dans lequel les mémoires sont expulsées des caches et atteignent la NVMM. Globalement, il s'agit d'une situation dans laquelle n'importe quelle mémoire peut atteindre la NVMM à presque n'importe quel moment et quel que soit l'ordre du programme.

En réponse, Intel a étendu son ISA pour la mémoire persistante et a inclus une sémantique d'instruction révisée ou de nouvelles instructions pour la persistance. Leur nouveau modèle de programmation (au niveau de l'architecture) pour la mémoire persistante présente des instructions (*flush*) pour demander la réécriture asynchrone d'un emplacement de mémoire spécifique et une autre (*fence*) pour établir des relations "happens-before" entre différentes réécritures.

Le fait est que les algorithmes en mémoire existants, tels que les structures de données, sont absolument incapables de fonctionner sur NVMM et de récupérer un état cohérent à la suite d'un crash, sans avoir été modifiés manuellement avec des flushes et des fences. Même si certains programmeurs experts peuvent accomplir cette tâche, elle reste une épreuve pour leur patience et leurs compétences. En outre, ce modèle de programmation de bas niveau conduit à une persistance fragile des programmes. Toute instruction flush

ou fence mal placée peut provoquer de nouveaux bogues, qui mettent silencieusement en péril l'ensemble des données. Il suffit de considérer que la persistance exacerbe les bogues de mémoire, puisque les fuites de mémoire ou la corruption du tas sont désormais permanentes.

À propos de cette thèse. Dans cette thèse, nous surfons sur l'engouement pour l'Optane et constatons avec dépit que l'essentiel des efforts visant à réduire la complexité de la programmation des NVMM se concentre sur les langages natifs (C, C++, etc.), et délaisse presque complètement les langages gérés, comme Java. Une observation singulière, étant donné que Java et son écosystème sont des acteurs majeurs dans le monde des big data - de nombreux magasins de données modernes, des cadres d'analyse ou de traitement des données sont en effet écrits en Java. Tous pourraient immensément bénéficier de la NVMM, mais il n'existe actuellement aucun moyen efficace et facilement applicable d'aborder la NVMM en Java.

Nous pensons qu'en apportant le plein potentiel de la NVMM à Java, nous pourrions offrir à une pléthore d'applications un nouveau type de persistance à grain fin qui changerait la donne. En même temps, les langages de haut niveau, et en particulier les idiomes orientés objet, semblent être des candidats solides pour entrelacer intuitivement la notion de données persistantes ou récupérables dans les programmes, en supprimant effectivement le fardeau de la programmation de la NVMM.

A.2 Motivation et Problématique

A.2.1 Intégration dans les langages de la persistance sur NVMM

Jusqu'à récemment, les supports volatils étaient beaucoup plus rapides que les supports persistants. Cette différence fondamentale a eu un impact considérable sur la manière dont les systèmes sont architecturés.

Les progrès récents de la technologie des mémoires persistantes promettent de redistribuer les cartes. En particulier, la mémoire principale non volatile (NVMM) est une mémoire adressable par octet qui préserve son contenu après une coupure de courant. Elle offre une durabilité et des performances de mémoire similaires à celles de la DRAM, ce qui promet une augmentation spectaculaire des performances de stockage.

Pour tirer parti des avantages de la NVMM, il est essentiel de l'intégrer aux langages de programmation. Cela est particulièrement important pour les langages utilisés dans la conception des systèmes de stockage distribués qui sont au cœur des infrastructures informatiques actuelles. Une telle intégration est cependant difficile car les langages orientés objet gérés sont des logiciels complexes qui héritent de décennies de raffinements et d'optimisations. Cette thèse aborde le problème de l'intégration de NVMM avec le langage Java.

A.2.2 Limitations de l'État de l'Art

À ce jour, les approches qui intègrent le NVMM à Java l'utilisent comme un support de stockage de masse accessible via une interface de système de fichiers [2, 184, 332], l'adressent via l'interface native Java (JNI) [16, 249], ou rendent de manière transparente une partie du tas Java persistante [288, 329]. Comme nous le verrons plus loin, ces approches sont génériques et insatisfaisantes pour plusieurs raisons.

Système de fichier et JNI. Le système de fichiers et les approches JNI maintiennent deux représentations des données, l'une en mémoire et l'autre dans la NVMM. Cela nécessite un transfert continu d'objets entre la mémoire persistante et la mémoire volatile. En particulier, des mécanismes logiciels complexes sont nécessaires pour maintenir les deux représentations mutuellement cohérentes. Nous démontrons dans notre évaluation (§5.3) que ces coûts logiciels (marshaling + consistency), bien qu'insignifiants avec les supports de stockage antérieurs, sont devenus de sérieux goulets d'étranglement avec l'avènement des dispositifs NVMM.

Intégration Totale. La conception intégrée résout le problème de la double représentation : les objets Java ordinaires sont directement et durablement stockés dans la NVMM. Grâce à l'adressabilité par octet de la NVMM, les objets persistants sont directement accessibles avec des instructions de lecture et d'écriture de la mémoire. Les données durables n'ont donc plus besoin d'être copiées par l'application du support de stockage dans la mémoire pour être manipulées. Cependant, l'intégration totale nécessite des modifications importantes et peu pratiques de la machine virtuelle Java (JVM), et s'accompagne de plusieurs limitations de performance et de problèmes de fiabilité, dont voici quelques exemples :

- ***Ramasse-miettes (GC).*** L'intégration d'objets persistants dans le tas Java signifie qu'ils doivent être ramassés. Figure A.1 montre que le ramassage des ordures de seulement 80 GB peut diviser par 3 le temps d'exécution, alors que NVMM devrait héberger des centaines de Go à des To de données. En outre, après avoir étudié plusieurs magasins de données prêts pour NVMM, nous avons constaté que les objets persistants sont souvent supprimés à un nombre très limité d'endroits. Dans l'ensemble, l'utilisation du ramassage des ordures pour les objets persistants ne semble pas nécessaire.

- ***Persistance Orthogonale.*** La conception intégrée manque de types persistants statiques et s'appuie sur l'instrumentation des bytecodes Java pour vérifier de manière transparente si un objet est alloué sur une mémoire volatile ou persistante au moment de l'exécution. En détail, Shull et al. [288] enregistre un ralentissement de 51% alors qu'il n'utilise même pas la NVMM. (9% avec une optimisation ultérieure du compilateur [289]) En outre, lorsque les états persistants de l'application ne sont pas mis en évidence par les types, ni le développeur ni le compilateur ne peuvent facilement identifier les bogues puisqu'ils se produisent au moment de l'exécution [92, 220]. Le fait de confondre un objet volatil avec un objet persistant entraîne une perte de données, à l'inverse d'une fuite de mémoire non volatile. Au lieu de perdre silencieusement des données ou de la mémoire, le moteur d'exécution devrait fournir une aide pour éviter ces situations.

A.2.3 Problématique

Ainsi, à l'heure actuelle, il existe un réel besoin d'une solution Java native appropriée pour accéder à la NVMM. En attendant, aucune charge de travail de traitement de données lourdes dans un environnement géré ne peut exploiter le plein potentiel de la NVMM et en tirer profit.

En particulier, une solution appropriée serait une solution qui : (i) *n'induit aucune surcharge logicielle lors de l'accès aux emplacements de mémoire persistante.* (ii) *ne*

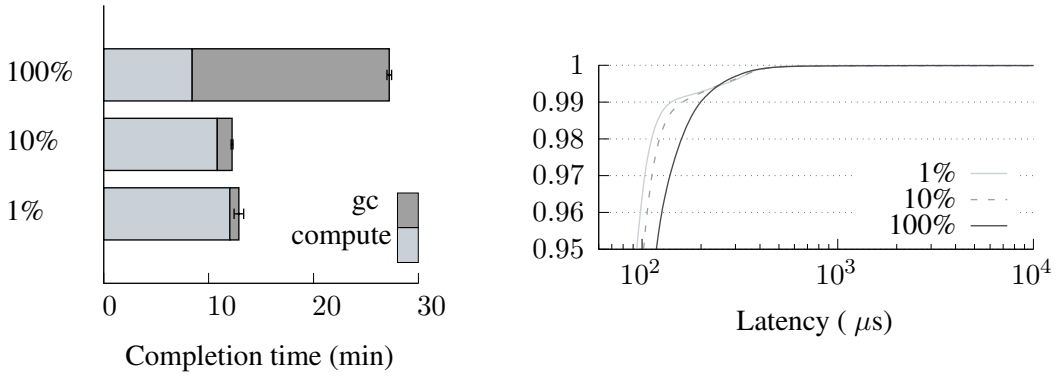


Figure A.1: YCSB-F en Java pour différentes proportions de données en cache mémoire (RAM).

connaît aucune limitation liée aux grandes capacités de la NVMM et à la gestion du tas (iii) *a un impact minimal sur les performances lors d'exécutions sans crash.* (iv) *est sûr et habilitant pour les programmeurs, qui leur laisse le contrôle, tout en faisant abstraction de la complexité de l'atomicité des défaillances grâce à des idiomes orientés objet.*

En revanche, la conception intégrée offre un accès direct à la NVMM, mais elle sacrifie la simplicité du code à une pénalité de performance (GC + instrumentation du code GC) et à des problèmes de fiabilité potentiels.

A.3 Contributions

A.3.1 Idée centrale

Dans cette thèse, nous proposons de remédier à ces lacunes en maintenant la NVMM en dehors du tas Java afin d'éviter une collecte de déchets coûteuse tout en conservant un accès direct à la NVMM comme dans la conception intégrée. À cette fin, nous introduisons un principe de découplage entre la structure de données d'un objet persistant et sa représentation dans la JVM. Plus précisément, les objets persistants sont séparés en une structure de données qui est stockée en dehors du tas sur la NVMM et un objet Java proxy qui reste en dehors du tas dans la mémoire volatile. La structure de données contient les champs de l'objet persistant, tandis que le proxy volatil agit comme une passerelle vers la structure de données durable hors tas et met en œuvre les méthodes de l'objet persistant. Grâce à cette conception, les données durables restent en dehors du tas Java (en utilisant une disposition de mémoire dédiée) et ne peuvent donc pas être collectées par le système d'exécution Java. La double représentation des données est également évitée grâce à une interface JVM qui intègre les instructions de bas niveau qui accèdent à la NVMM directement à partir des méthodes Java.

A.3.2 J-NVM, J-PDT, P-PFA et JNVM-Transformer

Ces idées clés sont mises en œuvre dans le cadre J-NVM [210], une bibliothèque légère pure-Java qui fonctionne sur la JVM Hotspot 8 avec l'ajout minimal de trois instructions spécifiques aux NVMM (`pwb`, `pfence` et `psync` [174]).

J-NVM est une interface de bas niveau qui se concentre sur la manipulation efficace du proxy et de la mémoire. À savoir, la logique nue pour instancier et détruire des objets

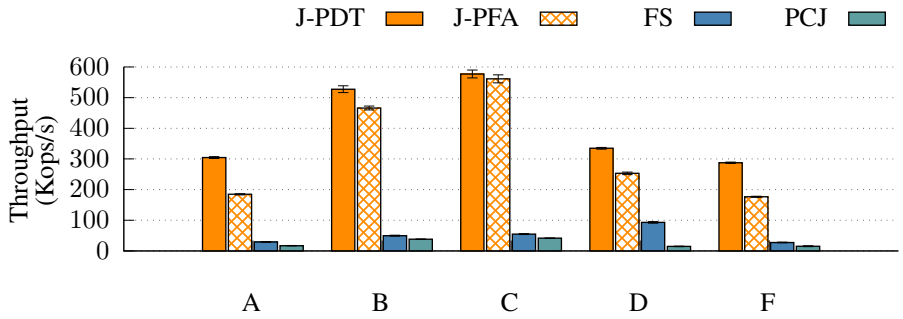


Figure A.2: Le benchmark YCSB.

persistants et accéder efficacement à leurs champs. Afin d'assurer la récupérabilité et la cohérence des données durables par le biais d'abstractions de programmation simples, nous construisons à partir de J-NVM deux interfaces de plus haut niveau : J-PFA et J-PDT.

- J-PFA fournit des blocs de code atomiques de défaillance, c'est-à-dire une manière générique de rendre tout code cohérent en cas de défaillance.

- J-PDT est une collection de structures de données cohérentes en cas de crash pour NVMM (par exemple, des tableaux, des cartes, des arbres), qui ne dépendent pas de J-PFA pour la performance.

De plus, comme J-NVM repose sur des types persistants explicites, nous fournissons un moyen automatisé de rendre les objets Java persistants avec l'ajout d'une seule annotation de classe. En effet, nous incluons un transformateur de code Java pour améliorer et découpler automatiquement les classes Java existantes en une structure de données persistante et un objet proxy volatile. Il est mis en œuvre sous la forme d'un plugin de transformation post-compilation de bytecode à bytecode Java hors ligne, intégré dans le système de construction de l'application. Il recherche les classes annotées et applique la transformation pour chacune d'entre elles, tout en tenant compte et en préservant les fonctionnalités définies par l'utilisateur dans n'importe quelle hiérarchie de classes.

A.4 Évaluation

A.4.1 Résultats

Nous évaluons J-NVM en mettant en œuvre plusieurs backends persistants pour Infinispan [229] - un magasin de données de qualité industrielle - et nous les testons sur une charge de travail de type TPC-B [9] ainsi que sur le benchmark YCSB [99]. Ces implémentations sont disponibles sur [209]. Figure A.2 décrit les performances sur les charges de travail YCSB pour les backends basés sur J-PFA et J-PDT, l'approche originale du système de fichiers FS assis sur DAX-ext4, ainsi qu'un backend basé sur PCJ qui utilise en interne le PMDK d'Intel [19] via l'interface native Java. J-NVM est nettement plus efficace que les approches précédentes, au moins un ordre de grandeur plus rapide.
Tout au long de notre campagne d'évaluation, nous montrons que :

- J-PDT et J-PFA sont systématiquement plus performants que la conception externe. Dans YCSB, J-PDT est au moins 10,5× plus rapide que FS ou PCJ, sauf dans un seul cas où il n'est que 3,6× plus rapide.
- Alors que les blocs atomiques de J-PFA offrent une solution globale, J-PDT, avec ses types de données persistants conçus à la main, s'exécute jusqu'à 65% plus rapidement. Comparé à l'implémentation Volatile, J-PDT est seulement 45-50% plus lent.
- L'intégration de la NVMM dans l'exécution du langage nuit aux performances en raison du coût de la collecte des objets persistants. Pour une application de type Redis écrite avec go-pmem [143], l'augmentation de l'ensemble de données persistantes de 0,3 GB à 151 GB multiplie le temps d'exécution de YCSB-F par 3,4.

A.4.2 Constatations

L'analyse des performances d'J-NVM a permis de dégager d'autres éléments pertinents.

Marshalling. Les faibles performances de FS proviennent des opérations de (dé)marshalling pour déplacer les objets persistants entre leur fichier et leur représentation Java. PCJ est fortement impacté par le coût des appels JNI pour échapper au monde Java. Ces opérations étaient couramment utilisées et n'avaient pas d'impact significatif avec des supports de stockage plus lents, mais elles peuvent désormais constituer des goulets d'étranglement avec NVMM et doivent être évitées dans la mesure du possible.

Mise en cache. Nous observons dans le benchmark YCSB que J-PDT ne bénéficie pas de la mise en cache. En effet, comme les données sont accédées directement et que seuls les mandataires sont conservés dans le cache, l'augmentation du ratio de cache n'a pratiquement aucun impact sur les temps de latence de lecture ou de mise à jour.

Récupération. La performance de la procédure de récupération est évaluée avec une charge de travail de type TPC-B (transactionnelle). J-PFA récupère environ 4.7× plus rapidement que FS et jusqu'à 8.6× plus rapidement avec une optimisation de récupération possible pour les charges de travail purement transactionnelles. Inversement, FS doit repeupler le cache en mémoire de 10% avec empressement lors de la récupération, alors qu'J-PFA ne peut recréer les proxies que paresseusement, avec une utilisation beaucoup moins importante de la bande passante NVMM.

List of Figures

List of Tables

Bibliography

[1] Apache Arrow, . URL https://arrow.apache.org.

[2] Direct Access for files, . URL https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[3] AWS Lambda, . URL https://aws.amazon.com/lambda/.

[4] Apache Lucene, . URL https://lucene.apache.org/.

[5] Project Panama, . URL https://openjdk.org/projects/panama/.

[6] Pmem-Redis, . URL https://github.com/pmem/pmem-redis.

[7] Pmem-rocksdb, . URL https://github.com/pmem/pmem-rocksdb.

[8] *IBM System/38, Technical Developments*. IBM, General Systems Division, Atlanta, Ga., 2nd ed edition, 1980. ISBN 978-0-933186-03-3.

[9] TPC-B, 1990. URL http://www.tpc.org/tpcb.

[10] ISO/IEC 9075-1:1999. Technical report, 1999.

[11] Hibernate ORM, May 2001. URL https://hibernate.org/.

[12] DataNucleus, 2008. URL https://www.datanucleus.org/.

[13] Redis, 2009. URL https://redis.io/.

[14] Memkind, 2016. URL https://pmem.io/memkind/.

[15] The Managed Data Structures Library: Java API. Technical report, 2017. URL https://github.com/HewlettPackard/mds/blob/master/doc/MDS%20Java%20API.pdf.

[16] Persistent Collection for Java, 2017. URL https://github.com/pmem/pcj.

[17] Cassandra-pmem, 2018. URL https://github.com/intel/cassandra-pmem/tree/13981.

[18] Persistent Memory Storage Engine for MongoDB, 2018. URL https://github.com/pmem/pmse.

[19] PMDK, 2018. URL https://pmem.io/pmdk.

[20] Low-Level Persistence Library, 2019. URL https://github.com/pmem/llpl.

[21] *NVM Express Specification 1.4*. 2019. URL https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[22] Pmemkv, 2019. URL https://github.com/pmem/pmemkv.

[23] Comparison between different memories(PCM, STT, RAM, SRAM, DRAM, Flash NAND, HDD), 2019. URL https://www.elinfor.com/knowledge/comparison-between-different-memoriespcm-stt-ram-sram-dram-flash-nand-hdd-p-1090

[24] Performance Report of Intel® Optane™ Persistent Memory on PRIMEFLEX® for SAP HANA®, 2020. URL https://sp.ts.fujitsu.com/dmsp/Publications/public/wp-performance-report-primergy-inteloptane-saphane.pdf.

[25] eADR: New Opportunities for Persistent Memory Applications, January 2021. URL https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[26] Mania Abdi, Sam Ginzburg, Charles Lin, Jose M Faleiro, Íñigo Goiri, Gohar Irfan Chaudhry, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM, May 2023. URL https://www.microsoft.com/en-us/research/publication/palette-load-balancing-locality-hints-for-serverless-functions/.

[27] Michael Ablassmeier. NullFSVFS: A black hole file system that behaves like /dev/null, 2020. URL https://github.com/abbbi/nullfsvfs.

[28] D. A. Abramson and J. Rosenberg. The microarchitecture of a capability-based computer. *ACM SIGMICRO Newsletter*, 17(4):138–145, December 1986. ISSN 1050-916X. doi: 10.1145/19530.19546.

[29] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, Providence RI USA, April 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304061.

[30] Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. ASAP: Architecture support for asynchronous persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 306–319, New York New York, June 2022. ACM. ISBN 978-1-4503-8610-4. doi: 10.1145/3470496.3527399.

[31] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, Xi'an China, April 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037706.

[32] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

[33] Hiroyuki Akinaga and Hisashi Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proceedings of the IEEE*, 98(12):2237–2251, December 2010. ISSN 0018-9219, 1558-2256. doi: 10.1109/JPROC.2010.2070830.

[34] Shoaib Akram. Performance Evaluation of Intel Optane Memory for Managed Workloads. *ACM Transactions on Architecture and Code Optimization*, 18(3):1–26, September 2021. ISSN 1544-3566, 1544-3973. doi: 10.1145/3451342.

[35] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Notices*, 53(4):62–77, December 2018. ISSN 0362-1340, 1558-1160. doi: 10.1145/3296979.3192392.

[36] Shoaib Akram, Jennifer Sartor, Kathryn McKinley, and Lieven Eeckhout. Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3 (1):1–27, March 2019. ISSN 2476-1249. doi: 10.1145/3322205.3311080.

[37] Mohammed Al-Mansari, Stefan Hanenberg, and Rainer Unland. Orthogonal persistence and AOP: A balancing act. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software - ACP4IS '07*, pages 2–es, Vancouver, British Columbia, Canada, 2007. ACM Press. doi: 10.1145/1233901.1233903.

[38] Mohammed Ali Nagi Al-Mansari. *Abstraction over Non-Local Object Information in Aspect-Oriented Programming Using Path Expression Pointcuts: A Case of Object Persistence*. PhD thesis, University of Duisburg-Essen, Germany, 2008. URL http://duepublico.uni-duisburg-essen.de/servlets/DerivateServlet/Derivate-20003/Thesis.pdf.

[39] Paul Alcom. Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off. *Tom's Hardware*, August 2022. URL https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good.

[40] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124, Seoul, Korea (South), February 2021. IEEE. ISBN 978-1-66542-235-2. doi: 10.1109/HPCA51647.2021.00019.

[41] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/anderson.

[42] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner

Thesing, Mehul Wagle, and Thomas Willhalm. SAP HANA Adoption of Non-Volatile Memory. *Proc. VLDB Endow.*, 10(12):1754–1765, 2017. doi: 10.14778/3137765.3137780.

[43] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems*, 9(2):1–35, May 2013. ISSN 1550-4832, 1550-4840. doi: 10.1145/2463585.2463589.

[44] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, January 2018. ISSN 21508097. doi: 10.1145/3187009.3164147.

[45] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976. ISSN 0362-5915, 1557-4644. doi: 10.1145/320455.320457.

[46] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, November 1983. ISSN 0010-4620, 1460-2067. doi: 10.1093/comjnl/26.4.360.

[47] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, December 1996. ISSN 0163-5808. doi: 10.1145/245882.245905.

[48] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–401, July 1995. ISSN 1066-8888, 0949-877X. doi: 10.1007/BF01231642.

[49] Malcolm Atkinson, Mick Jordan, Malcolm Atkinson, and Mick Jordan. A review of the rationale and architectures of PJama: A durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow, 2000.

[50] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 7–16, Egham United Kingdom, July 2018. ACM. ISBN 978-1-4503-5795-1. doi: 10.1145/3212734.3212753.

[51] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in Order to Recover - Detectable Recovery of Lock-Free Data Structures. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 503–505, Virtual Event USA, July 2020. ACM. ISBN 978-1-4503-6935-0. doi: 10.1145/3350755.3400257.

[52] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–277, Seoul Republic of Korea, April 2022. ACM. ISBN 978-1-4503-9204-4. doi: 10.1145/3503221.3508444.

[53] Mary Baker, Satoshi Asami, Etienne Deprit, John K. Ousterhout, and Margo I. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In Barry Flahive and Richard L. Wexelblat, editors, *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, October 12-15, 1992*, pages 10–22. ACM Press, 1992. doi: 10.1145/143365.143380.

[54] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent Memory: A Survey of Programming Support and Implementations. *ACM Computing Surveys*, 54(7):1–37, September 2022. ISSN 0360-0300, 1557-7341. doi: 10.1145/3465402.

[55] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017. ISSN 0001-0782, 1557-7317. doi: 10.1145/3015146.

[56] Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and Lower Bounds on the Space Complexity of Detectable Objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 11–20, Virtual Event Italy, July 2020. ACM. ISBN 978-1-4503-7582-5. doi: 10.1145/3382734.3405725.

[57] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-Free Concurrency on Faulty Persistent Memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, Phoenix AZ USA, June 2019. ACM. ISBN 978-1-4503-6184-2. doi: 10.1145/3323165.3323187.

[58] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1972. ISSN 0001-0782, 1557-7317. doi: 10.1145/355602.361306.

[59] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. ISBN 0-201-10715-5.

[60] Eduardo Berrocal. The Apache Cassandra Transformation for Persistent Memory | Intel Software, 2018. URL https://www.youtube.com/watch?v=oMeOyyiQcP0.

[61] Eduardo Berrocal and Garcia De Carellan. Making NoSQL Databases Persistent-Memory-Aware: The Apache Cassandra* Example, 2018. URL https://www.intel.com/content/www/us/en/developer/articles/technical/making-nosql-databases-persistent-memory-aware-the-apache-cassandra-example.html.

[62] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. page 17 pages, 2016. doi: 10.4230/LIPICS.OPODIS.2015.20.

[63] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, Part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 677–694. ACM, 2016. doi: 10.1145/2983990.2984019.

[64] Shamim Bhuiyan, Michael Zheludkov, and Timur Isachenko. *High Performance In-Memory Computing with Apache Ignite*. Lulu.com, 2017. ISBN 1-365-73235-5.

[65] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. Twizzler: A Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 65–80. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/bittman.

[66] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. The Necessary Death of the Block Device Interface. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. URL https://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper89.pdf.

[67] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/bjorling.

[68] Michael D. Bond and Kathryn S. McKinley. Leak pruning. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '09*, page 277, Washington, DC, USA, 2009. ACM Press. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508277.

[69] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 836–850, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483540.

[70] Dhruba Borthakur, Samuel Rash, Rodrigo Schmidt, Amitanand Aiyer, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, and Aravind Menon. Apache hadoop goes realtime at Facebook. In *Proceedings of the 2011 International Conference on Management of Data - SIGMOD '11*, page 1071, Athens, Greece, 2011. ACM Press. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989438.

[71] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. SafePM: A sanitizer for persistent memory. In *Proceedings of the Seventeenth*

European Conference on Computer Systems, pages 506–524, Rennes France, March 2022. ACM. ISBN 978-1-4503-9162-7. doi: 10.1145/3492321.3519574.

[72] AL Brown and WP Cockshott. The CPOMS persistent object management system. *Universities of Glasgow and St Andrews, PPRR-13*, 1985.

[73] Alfred Leonard Brown. *Persistent Object Stores*. PhD thesis, University of St. Andrews (United Kingdom), 1989.

[74] Eric Bruneton. ASM, 2002. URL https://asm.ow2.io.

[75] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of the Adaptable and Extensible Component Systems*, 2002.

[76] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 60–73, London UK, June 2020. ACM. ISBN 978-1-4503-7566-5. doi: 10.1145/3381898.3397212.

[77] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. Fast Nonblocking Persistence for Concurrent Data Structures. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-210-5. doi: 10.4230/LIPIcs.DISC.2021.14.

[78] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. *ACM SIGPLAN Notices*, 52(4):207–221, May 2017. ISSN 0362-1340, 1558-1160. doi: 10.1145/3093336.3037721.

[79] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - A Technique for Cheap Recovery. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 31–44. USENIX Association, 2004. URL http://www.usenix.org/events/osdi04/tech/candea.html.

[80] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and File Management in the EXODUS Extensible Database System. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 91–100. Morgan Kaufmann, 1986. URL http://www.vldb.org/conf/1986/P091.PDF.

[81] R. G. G. Cattell, Douglas K. Barry, and Mark Berler, editors. *The Object Data Standard: ODMG 3.0*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, 2000. URL http://www.odbms.org/odmg-standard/.

[82] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, LA, USA, November 2010. IEEE. ISBN 978-1-4244-7557-5. doi: 10.1109/SC.2010.56.

[83] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 433–452, Portland Oregon USA, October 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660224.

[84] Michael F. Challis. Database Consistency and Integrity in a Multi-User Environment. In Ben Shneiderman, editor, *Proceedings of the International Conference on Databases: Improving Usability and Responiveness, August 2-3, 1978, Technion, Haifa, Israel*, page 245. Academic Press, 1978.

[85] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994. ISSN 0734-2071, 1557-7333. doi: 10.1145/195792.195795.

[86] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS-VII*, pages 74–83, Cambridge, Massachusetts, United States, 1996. ACM Press. ISBN 978-0-89791-767-4. doi: 10.1145/237090.237154.

[87] Qichen Chen and Heonyoung Yeom. Design of Skiplist Based Key-Value Store on Non-Volatile Memory. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 44–50, Trento, September 2018. IEEE. ISBN 978-1-5386-5175-9. doi: 10.1109/FAS-W.2018.00024.

[88] Shimin Chen and Qin Jin. Persistent B $^{+}$ -trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, February 2015. ISSN 2150-8097. doi: 10.14778/2752939.2752947.

[89] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. Efficiently detecting concurrency bugs in persistent memory programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 873–887, Lausanne Switzerland, February 2022. ACM. ISBN 978-1-4503-9205-1. doi: 10.1145/3503222.3507755.

[90] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 441–457, Online Event United Kingdom, April 2021. ACM. ISBN 978-1-4503-8334-9. doi: 10.1145/3447786.3456252.

[91] Sakib Chowdhury and Wojciech Golab. A Scalable Recoverable Skip List for Persistent Memory. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 426–428, Virtual Event USA, July 2021. ACM. ISBN 978-1-4503-8070-6. doi: 10.1145/3409964.3461819.

[92] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '11*, page 105, Newport Beach, California, USA, 2011. ACM Press. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950380.

[93] Gaetano C. Coccimiglio, Trevor A. Brown, and Srivatsan Ravi. PREP-UC: A Practical Replicated Persistent Universal Construction. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 217–229, Philadelphia PA USA, July 2022. ACM. ISBN 978-1-4503-9146-7. doi: 10.1145/3490148.3538568.

[94] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent object management system. *Software: Practice and Experience*, 14(1): 49–71, January 1984. ISSN 00380644, 1097024X. doi: 10.1002/spe.4380140106.

[95] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, January 1983. ISSN 0001-0782, 1557-7317. doi: 10.1145/357980.358007.

[96] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 259–269, Vienna Austria, July 2018. ACM. ISBN 978-1-4503-5799-9. doi: 10.1145/3210377.3210400.

[97] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 441–454, Providence RI USA, April 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304046.

[98] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*, page 133, Big Sky, Montana, USA, 2009. ACM Press. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629589.

[99] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing - SoCC '10*, page 143, Indianapolis, Indiana, USA, 2010. ACM Press. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152.

[100] George Copeland and David Maier. Making smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84*, page 316, Boston, Massachusetts, 1984. ACM Press. ISBN 978-0-89791-128-3. doi: 10.1145/602259.602300.

[101] Jonathan Corbet. NUMA nodes for persistent-memory management, 2019. URL https://lwn.net/Articles/787418/.

[102] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, Vienna Austria, July 2018. ACM. ISBN 978-1-4503-5799-9. doi: 10.1145/3210377.3210392.

[103] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, Heraklion Greece, April 2020. ACM. ISBN 978-1-4503-6882-7. doi: 10.1145/3342195.3387515.

[104] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–116, San Diego California, February 2020. ACM. ISBN 978-1-4503-6818-6. doi: 10.1145/3332466.3374523.

[105] Tom Coughlin. Gifts From Intel's Optane Memory. *Forbes*, August 2022. URL https://www.forbes.com/sites/tomcoughlin/2022/08/08/gifts-from-intels-optane-memory/?sh=1273e5f94e3d.

[106] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 339–351, Virtual Event China, June 2021. ACM. ISBN 978-1-4503-8343-1. doi: 10.1145/3448016.3457292.

[107] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in Multics. In *Proceedings of the ACM Symposium on Operating System Principles - SOSP '67*, pages 12.1–12.8, Not Known, 1967. ACM Press. doi: 10.1145/800001.811668.

[108] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. NVAlloc: Rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–127, Lausanne Switzerland, February 2022. ACM. ISBN 978-1-4503-9205-1. doi: 10.1145/3503222.3507743.

[109] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *Computer*, 24(11):34–44, November 1991. ISSN 0018-9162. doi: 10.1109/2.116849.

[110] Partha Dasgupta, Raymond C. Chen, Sathis Menon, Mark P. Pearson, R. Ananthanarayanan, Umakishore Ramachandran, Mustaque Ahamad, Richard J. LeBlanc, William F. Appelbe, José M. Bernabéu-Aubán, Phillip W. Hutto, M. Yousef Amin Khalidi, and Christopher J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. *Comput. Syst.*, 3(1):11–46, 1989. URL http://www.usenix.org/publications/compsystems/1990/win_dasgupta.pdf.

[111] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/david.

[112] May Dawn. IBM I Single Level Store ... In Lieu of a Crystal Ball. *IBM Systems magazine*, 2013. URL https://dawnmayi.com/2013/04/16/ibm-i-single-level-store-in-lieu-of-a-crystal-ball/.

[113] Alan Dearie, Rex di Bona, James Farrow, Frans Henskens, David Hulse, Anders Lindström, Stephen Norris, John Rosenberg, and Francis Vaughan. Protection in Grasshopper: A Persistent Operating System. In C. J. van Rijsbergen, Malcolm Atkinson, David Maier, and Véronique Benzaken, editors, *Persistent Object Systems*, pages 60–78. Springer London, London, 1995. ISBN 978-3-540-19912-0 978-1-4471-2122-0. doi: 10.1007/978-1-4471-2122-0_6.

[114] A. Dearle and D. Hulse. On page-based optimistic process checkpointing. In *Proceedings of International Workshop on Object Orientation in Operating Systems*, pages 24–32, Lund, Sweden, 1995. IEEE Comput. Soc. Press. ISBN 978-0-8186-7115-9. doi: 10.1109/IWOOS.1995.470583.

[115] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan, and K. Maciunas. An examination of operating system support for persistent object systems. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 779–789 vol.1, Kauai, HI, USA, 1992. IEEE. ISBN 978-0-8186-2420-9. doi: 10.1109/HICSS.1992.183232.

[116] Alan Dearle and David Hulse. Operating system support for persistent systems: Past, present and future. *Software: Practice and Experience*, 30(4):295–324, April 2000. doi: 10.1002/(sici)1097-024x(20000410)30:4<295::aid-spe301>3.0.co;2-p.

[117] Alan Dearle, Rex di Bona, James Farrow, Frans A. Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Comput. Syst.*, 7(3):289–312, 1994. URL http://www.usenix.org/publications/compsystems/1994/sum_dearle.pdf.

[118] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles - SOSP '07*, page 205, Stevenson, Washington, USA, 2007. ACM Press. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281.

[119] Carole Delporte-Gallet, Panagiota Fatourou, Hugues Fauconnier, and Eric Ruppert. When is Recoverable Consensus Harder Than Consensus? In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 198–208, Salerno Italy, July 2022. ACM. ISBN 978-1-4503-9262-4. doi: 10.1145/3519270.3538418.

[120] Linda DeMichiel and Michael Keith. JSR 220: Java Persistence API, May 2006. URL https://jcp.org/en/jsr/detail?id=220.

[121] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management - ISMM '04*, page 37, Vancouver, BC, Canada, 2004. ACM Press. ISBN 978-1-58113-945-7. doi: 10.1145/1029873.1029879.

[122] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, Virtual USA, April 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446744.

[123] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978. ISSN 0001-0782, 1557-7317. doi: 10.1145/359642.359655.

[124] Andrew Dinn. JEP 352: Non-Volatile Mapped Byte Buffers, 2018. URL https://openjdk.java.net/jeps/352.

[125] Dormando. The Volatile Benefit of Persistent Memory, 2019. URL https://memcached.org/blog/persistent-memory/.

[126] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing - STOC '86*, pages 109–121, Berkeley, California, United States, 1986. ACM Press. ISBN 978-0-89791-193-1. doi: 10.1145/12130.12142.

[127] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, pages 1–15, Amsterdam, The Netherlands, 2014. ACM Press. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592814.

[128] Remi Dulong, Rafael Pires, Andreia Correia, Valerio Schiavoni, Pedro Ramalhete, Pascal Felber, and Gael Thomas. NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 186–198, Taipei, Taiwan, June 2021. IEEE. ISBN 978-1-66543-572-7. doi: 10.1109/DSN48987.2021.00033.

[129] Ted Dziuba. Russian rides Phantom to OS immortality, 2009. URL https://www.theregister.com/2009/02/03/phantom_russian_os/.

[130] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 337–352, Seoul Republic of Korea, April 2022. ACM. ISBN 978-1-4503-9204-4. doi: 10.1145/3503221.3508426.

[131] Alexandra Fedorova. We Replaced an SSD with Storage Class Memory; Here is What We Learned, 2020. URL https://www.mongodb.com/blog/post/we-replaced-ssd-storage-class-memory-here-what-we-learned.

[132] Alexandra Fedorova, Keith A. Smith, Keith Bostic, Susan LoVerso, Michael Cahill, and Alex Gorrod. Writes hurt: Lessons in cache design for optane NVRAM. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 110–125, San Francisco California, November 2022. ACM. ISBN 978-1-4503-9414-7. doi: 10.1145/3542929.3563461.

[133] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '08*, page 237, Salt Lake City, UT, USA, 2008. ACM Press. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345241.

[134] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, December 2010. ISSN 1045-9219. doi: 10.1109/TPDS.2010.49.

[135] Paulo Ferreira and Marc Shapiro. Garbage Collection of Persistent Objects in Distributed Shared Memory. In C. J. van Rijsbergen, Malcolm Atkinson, David Maier, and Véronique Benzaken, editors, *Persistent Object Systems*, pages 184–199. Springer London, London, 1995. ISBN 978-3-540-19912-0 978-1-4471-2122-0. doi: 10.1007/978-1-4471-2122-0_16.

[136] John Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, October 1961. ISSN 0001-0782, 1557-7317. doi: 10.1145/366786.366800.

[137] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE '09*, page 81, Washington, DC, USA, 2009. ACM Press. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508305.

[138] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–40, Vienna Austria, February 2018. ACM. ISBN 978-1-4503-4982-6. doi: 10.1145/3178487.3178490.

[139] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 377–392, London UK, June 2020. ACM. ISBN 978-1-4503-7613-6. doi: 10.1145/3385412.3386031.

[140] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1218–1232, Virtual Canada, June 2021. ACM. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454105.

[141] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 100–115, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483556.

[142] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–74, London UK, June 2020. ACM. ISBN 978-1-4503-7613-6. doi: 10.1145/3385412.3385991.

[143] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. Go-pmem: Native Support for Programming Persistent Memory in Go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/george.

[144] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–61, Philadelphia PA USA, June 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192367.

[145] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665, Valencia, Spain, May 2020. IEEE. ISBN 978-1-72814-661-4. doi: 10.1109/ISCA45697.2020.00060.

[146] Wojciech Golab. The Recoverable Consensus Hierarchy. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–291, Virtual Event USA, July 2020. ACM. ISBN 978-1-4503-6935-0. doi: 10.1145/3350755.3400212.

[147] Wojciech Golab and Aditya Ramaraju. Recoverable Mutual Exclusion: [Extended Abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, Chicago Illinois USA, July 2016. ACM. ISBN 978-1-4503-3964-3. doi: 10.1145/2933057.2933087.

[148] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428, Virtual USA, April 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446735.

[149] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. Checking robustness to weak persistency models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 490–505, San Diego CA USA, June 2022. ACM. ISBN 978-1-4503-9265-5. doi: 10.1145/3519939.3523723.

[150] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: Detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 830–845, Lausanne Switzerland, February 2022. ACM. ISBN 978-1-4503-9205-1. doi: 10.1145/3503222.3507766.

[151] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. White-Box Atomic Multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 176–187, Portland, OR, USA, June 2019. IEEE. ISBN 978-1-72810-057-9. doi: 10.1109/DSN.2019.00030.

[152] Paul Green. Multics Virtual Memory - Tutorial and Reflections, 1993. URL https://multicians.org/pg/mvm.html.

[153] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software Persistent Memory. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 319–331, Boston, MA, June 2012. USENIX Association. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/guerra.

[154] R. Guerraoui and R.R. Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407, Tokyo, Japan, 2004. IEEE. ISBN 978-0-7695-2086-5. doi: 10.1109/ICDCS.2004.1281605.

[155] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. ISSN 0360-0300, 1557-7341. doi: 10.1145/289.291.

[156] Nassim A. Halli, Henri-Pierre Charles, and Jean-François Mehaut. Performance comparison between Java and JNI for optimal implementation of computational micro-kernels. 2014. doi: 10.48550/ARXIV.1412.6765.

[157] Jonathan Halliday. Mashona, 2021. URL https://github.com/jhalliday/mashona.

[158] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, October 1985. ISSN 0163-5980. doi: 10.1145/858336.858337.

[159] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 775–788, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378472.

[160] G. D. Held, M. R. Stonebraker, and E. Wong. INGRES: A relational data base system. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition on - AFIPS '75*, page 409, Anaheim, California, 1975. ACM Press. doi: 10.1145/1499949.1500029.

[161] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM SIGPLAN Notices*, 25(3):197–206, March 1990. ISSN 0362-1340, 1558-1160. doi: 10.1145/99164.99185.

[162] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, May 1993. ISSN 0163-5964. doi: 10.1145/173682.165164.

[163] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. ISSN 0164-0925, 1558-4593. doi: 10.1145/78969.78972.

[164] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 17–33. USENIX Association, 2018. URL https://www.usenix.org/conference/nsdi18/presentation/honda.

[165] Morteza Hoseinzadeh. *Fast, Durable, and Safe Data Management Support for Persistent Memory*. PhD thesis, 2021. URL https://escholarship.org/uc/item/5w12f260#main.

[166] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442, Virtual USA, April 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446710.

[167] Morteza Hoseinzadeh and Steven Swanson. Carbide: A Safe Persistent Memory Multilingual Programming Framework. San Diego CA USA, 2022. URL http://nvmw.ucsd.edu/program/.

[168] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482, Belgrade Serbia, April 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064204.

[169] 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*, pages 1049–1063, Philadelphia PA USA, June 2022. ACM. ISBN 978-1-4503-9249-5. doi: 10.1145/3514221.3517884.

[170] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL https://www.cidrdb.org/cidr2022/papers/p64-huang.pdf.

[171] David Hulse and Alan Dearle. RT1R1/2: Report on the efficacy of Persistent Operating Systems in supporting Persistent Application Systems. page 25, 2000.

[172] Taeho Hwang, Jaemin Jung, and Youjip Won. HEAPO: Heap-Based Persistent Object Store. *ACM Transactions on Storage*, 11(1):1–21, February 2015. ISSN 1553-3077, 1553-3093. doi: 10.1145/2629619.

[173] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–442, Atlanta Georgia USA, March 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872410.

[174] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing*, volume 9888, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53425-0 978-3-662-53426-7. doi: 10.1007/978-3-662-53426-7_23.

[175] 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. 2019. doi: 10.48550/ARXIV.1903.05714.

[176] Goslingn James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java™\ language specification. chapter 17.4. Java se 14 edition, 2020.

[177] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 330–335, India, January 2014. IEEE. ISBN 978-1-4799-2513-1. doi: 10.1109/VLSID.2014.63.

[178] Jungi Jeong and Changhee Jung. PMEM-spec: Persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–529, Virtual USA, April 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446698.

[179] Jungi Jeong, Jianping Zeng, and Changhee Jung. Capri: Compiler and Architecture Support for Whole-System Persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 71–83, Minneapolis MN USA, June 2022. ACM. ISBN 978-1-4503-9199-3. doi: 10.1145/3502181.3531474.

[180] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, Shanghai China, October 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132764.

[181] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[182] David Jordan and Craig L. Russell. *Java Data Objects*. O'Reilly, Beijing : Sebastopol, Calif, 2003. ISBN 978-0-596-00276-3.

[183] Mick J. Jordan and Malcolm P. Atkinson. Orthogonal Persistence for Java? - A Mid-Term Report. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*, pages 335–352, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-585-1.

[184] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, Huntsville Ontario Canada, October 2019. ACM. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359631.

[185] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 804–818, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483567.

[186] Kevin C. Kahn, William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, and Fred J. Pollack. iMAX: A multiprocessor operating system for an object-based computer. *ACM SIGOPS Operating Systems Review*, 15(5):127–136, December 1981. ISSN 0163-5980. doi: 10.1145/1067627.806601.

[187] Sheetal V. Kakkad and Paul R. Wilson. Address Translation Strategies in the Texas Persistent Store. In *5th Conference on Object-Oriented Technologies and Systems (COOTS 99)*, San Diego, CA, May 1999. USENIX Association. URL https://www.usenix.org/conference/coots-99/address-translation-strategies-texas-persistent-store.

[188] J. L. Keedy. Persistent Virtual Memory, 1990. URL https://www.monads-security.org/persistent-virtual-memory.html.

[189] Terence Kelly. Persistent Memory Programming on Conventional Hardware: The persistent memory style of programming can dramatically simplify application software. *Queue*, 17(4):1–20, August 2019. ISSN 1542-7730, 1542-7749. doi: 10.1145/3358955.3358957.

[190] Terence Kelly. Persistent Memory Allocation: Leverage to move a world of software. *Queue*, 20(2):16–30, April 2022. ISSN 1542-7730, 1542-7749. doi: 10.1145/3534855.

[191] Terence Kelly. Persistent-Memory gawk User Manual, August 2022. URL https://www.gnu.org/software/gawk/manual/pm-gawk/pm-gawk.pdf.

[192] Kishor Kharbas. JEP 316: Heap Allocation on Alternative Memory Devices, 2016. URL https://openjdk.java.net/jeps/316.

[193] Ana Khorguani, Thomas Ropars, and Noel De Palma. ResPCT: Fast checkpointing in non-volatile memory for multi-threaded applications. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 525–540, Rennes France, March 2022. ACM. ISBN 978-1-4503-9162-7. doi: 10.1145/3492321.3519590.

[194] Artem Khyzha and Ori Lahav. Taming x86-TSO persistency. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, January 2021. ISSN 2475-1421. doi: 10.1145/3434328.

[195] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[196] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-Level Storage System. *IEEE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962. ISSN 0367-7508. doi: 10.1109/TEC.1962.5219356.

[197] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. *ACM SIGPLAN Notices*, 51(4):385–398, June 2016. ISSN 0362-1340, 1558-1160. doi: 10.1145/2954679.2872392.

[198] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Transactions on Storage*, 14(1):1–17, April 2018. ISSN 1553-3077, 1553-3093. doi: 10.1145/3129263.

[199] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 424–439, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483589.

[200] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Taipei, Taiwan, October 2016. IEEE. ISBN 978-1-5090-3508-3. doi: 10.1109/MICRO.2016.7783761.

[201] Elliot K. Kolodner and William E. Weihl. Atomic incremental garbage collection and recovery for a large stable heap. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data - SIGMOD '93*, pages 177–186, Washington, D.C., United States, 1993. ACM Press. ISBN 978-0-89791-592-2. doi: 10.1145/170035.170068.

[202] Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, and Gustavo Alonso. How to use Persistent Memory in your Database, 2021.

[203] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, Shanghai China, October 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132770.

[204] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[205] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991. ISSN 0001-0782, 1557-7317. doi: 10.1145/125223.125244.

[206] C.R. Landau. The checkpoint mechanism in KeyKOS. In *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86–91, Dourdan, France, 1992. IEEE Comput. Soc. Press. ISBN 978-0-8186-3015-6. doi: 10.1109/IWOOOS.1992.252995.

[207] Sangwon Lee, Miryeong Kwon, Gyuyoung Park, and Myoungsoo Jung. LightPC: Hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 289–305, New York New York, June 2022. ACM. ISBN 978-1-4503-8610-4. doi: 10.1145/3470496.3527397.

[208] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, Huntsville Ontario Canada, October 2019. ACM. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359635.

[209] Anatole Lefort. J-NVM code base, 2021. URL `https://github.com/jnvm-project/jnvm`.

[210] Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, and Gaël Thomas. J-NVM: Off-heap Persistent Objects in Java. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 408–423, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483579.

[211] Alberto Lerner and Philippe Bonnet. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2852–2858, Virtual Event China, June 2021. ACM. ISBN 978-1-4503-8343-1. doi: 10.1145/3448016.3457540.

[212] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, December 2019. ISSN 2150-8097. doi: 10.14778/3372716.3372728.

[213] Daixuan Li, Benjamin Reidys, Jinghan Sun, Thomas Shull, Josep Torrellas, and Jian Huang. UniHeap: Managing persistent objects across managed runtimes for non-volatile memory. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, Haifa Israel, June 2021. ACM. ISBN 978-1-4503-8398-1. doi: 10.1145/3456727.3463775.

[214] Nan Li and Wojciech Golab. Detectable Sequential Specifications for Recoverable Shared Objects. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:19, Dagstuhl, Germany, 2021.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-210-5. doi: 10.4230/LIPIcs.DISC.2021.29.

[215] Zhe Li and Mingyu Wu. Transparent and lightweight object placement for managed workloads atop hybrid memories. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 72–80, Virtual Switzerland, February 2022. ACM. ISBN 978-1-4503-9251-8. doi: 10.1145/3516807.3516822.

[216] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. Crash Consistent Non-Volatile Memory Express. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 132–146. ACM, 2021. doi: 10.1145/3477132.3483592.

[217] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data - SIGMOD '96*, pages 318–329, Montreal, Quebec, Canada, 1996. ACM Press. ISBN 978-0-89791-794-0. doi: 10.1145/233269.233346.

[218] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988. ISSN 0001-0782, 1557-7317. doi: 10.1145/42392.42399.

[219] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378452.

[220] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, Virtual USA, April 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446691.

[221] Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, March 1977. ISSN 0362-5915, 1557-4644. doi: 10.1145/320521.320540.

[222] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles - SOSP '97*, pages 92–101, Saint Malo, France, 1997. ACM Press. ISBN 978-0-89791-916-6. doi: 10.1145/268998.266665.

[223] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *Proceedings of the VLDB Endowment*, 13(8):1147–1161, April 2020. ISSN 2150-8097. doi: 10.14778/3389133.3389134.

[224] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures

on Asymmetric NVM Architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378511.

[225] Kiwan Maeng and Brandon Lucia. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/maeng.

[226] Tobias Mann. Why Intel killed its Optane memory business. *The Register*, July 2022. URL https://www.theregister.com/2022/07/29/intel_optane_memory_dead/.

[227] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Sysposium on Principles of Programming Languages - POPL '05*, pages 378–391, Long Beach, California, USA, 2005. ACM Press. ISBN 978-1-58113-830-6. doi: 10.1145/1040305.1040336.

[228] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent Memory Transactions, March 2018. URL http://arxiv.org/abs/1804.00701.

[229] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[230] Alonso Marquez, John N. Zigman, and Stephen M. Blackburn. Fast portable orthogonally persistent JavaTM. *Software: Practice and Experience*, 30(4):449–479, April 2000. ISSN 0038-0644, 1097-024X. doi: 10.1002/(SICI)1097-024X(20000410)30:4<449::AID-SPE306>3.0.CO;2-Y.

[231] Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, and Yoav Zuriel. Oak: A scalable off-heap allocated key-value map. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 17–31, San Diego California, February 2020. ACM. ISBN 978-1-4503-6818-6. doi: 10.1145/3332466.3374526.

[232] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 789–806, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378456.

[233] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992. ISSN 0362-5915, 1557-4644. doi: 10.1145/128765.128770.

[234] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/mohan.

[235] Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech Golab. A Closer Look at Detectable Objects for Persistent Memory. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*, pages 56–64, Salerno Italy, July 2022. ACM. ISBN 978-1-4503-9280-8. doi: 10.1145/3524053.3542749.

[236] Ron Morrison, Richard Connor, Graham Kirby, David Munro, Malcolm P. Atkinson, Quintin Cutts, Fred Brown, and Alan Dearie. The Napier88 Persistent Programming Language and Environment. In Malcolm P. Atkinson and Ray Welland, editors, *Fully Integrated Data Environments*, pages 98–154. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-642-64055-1 978-3-642-59623-0. doi: 10.1007/978-3-642-59623-0_6.

[237] J.E.B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, Aug./1992. ISSN 00985589. doi: 10.1109/32.153378.

[238] Brown Munro and Moss Morrison. Incremental Garbage Collection of a Persistent Object Store using PMOS. page 13, 1998.

[239] Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeff Markham. *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 1st edition, 2014. ISBN 0-321-93450-4.

[240] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–148, Xi'an China, April 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037730.

[241] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL https://www.usenix.org/conference/fast19/presentation/nam.

[242] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, page 401, London, England, UK, 2012. ACM Press. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151018.

[243] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In

Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-053-8. doi: 10.4230/LIPIcs.DISC.2017.37.

[244] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How Persistent is your Persistent Memory Application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/neal.

[245] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 349–365, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen.

[246] Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler. Reliable object storage to support atomic actions. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles - SOSP '85*, pages 147–159, Orcas Island, Washington, United States, 1985. ACM Press. ISBN 978-0-89791-174-0. doi: 10.1145/323647.323642.

[247] Carl Olofson. Unleashing the Power of In-Memory Computing: Intel Optane DC Persistent Memory for SAP HANA, 2019. URL https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sap-hana-and-persistent-memory.pdf.

[248] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, page 29, Cascais, Portugal, 2011. ACM Press. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043560.

[249] Oracle. *Java Native Interface Specification*. Java se 14 edition, 2020.

[250] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972. ISBN 0-262-15012-3.

[251] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles - SOSP '93*, pages 161–174, Asheville, North Carolina, United States, 1993. ACM Press. ISBN 978-0-89791-632-5. doi: 10.1145/168619.168632.

[252] 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 *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, San Francisco California USA, June 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2915251.

[253] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. GPM: Leveraging persistent memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 142–156, Lausanne Switzerland, February 2022. ACM. ISBN 978-1-4503-9205-1. doi: 10.1145/3503222.3507758.

[254] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, page 225, Prague, Czech Republic, 2013. ACM Press. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465374.

[255] Matej Pavlovic, Alex Kogan, Virendra J. Marathe, and Tim Harris. Brief Announcement: Persistent Multi-Word Compare-and-Swap. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 37–39, Egham United Kingdom, July 2018. ACM. ISBN 978-1-4503-5795-1. doi: 10.1145/3212734.3212783.

[256] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, Minneapolis, MN, USA, June 2014. IEEE. ISBN 978-1-4799-4394-4 978-1-4799-4396-8. doi: 10.1109/ISCA.2014.6853222.

[257] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the Intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, Washington District of Columbia USA, September 2019. ACM. ISBN 978-1-4503-7206-0. doi: 10.1145/3357526.3357568.

[258] Rui Humberto R. Pereira and José Baltasar García Pérez-Schofield. An aspect-oriented framework for orthogonal persistence. *5th Iberian Conference on Information Systems and Technologies*, pages 1–6, 2010.

[259] Taciano D. Perez, Marcelo V. Neves, Diego Medaglia, Pedro H. G. Monteiro, and César A. F. De Rose. Orthogonal persistence in nonvolatile memory architectures: A persistent heap design and its implementation for a Java Virtual Machine. *Software: Practice and Experience*, 50(4):368–387, April 2020. ISSN 0038-0644, 1097-024X. doi: 10.1002/spe.2781.

[260] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '10*, page 146, Toronto, Ontario, Canada, 2010. ACM Press. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806615.

[261] Fred J. Pollack, Kevin C. Kahn, and Roy M. Wilkinson. The iMAX-432 object filing system. In *Proceedings of the Eighth Symposium on Operating Systems Principles - SOSP '81*, pages 137–147, Pacific Grove, California, United States, 1981. ACM Press. ISBN 978-0-89791-062-0. doi: 10.1145/800216.806602.

[262] Shashank Priya and Daniel J. Inman, editors. *Energy Harvesting Technologies*. Springer US, Boston, MA, 2009. ISBN 978-0-387-76463-4 978-0-387-76464-1. doi: 10.1007/978-0-387-76464-1.

[263] Liam Proven. Why the end of Optane is bad news for all IT. *The Register*, August 2022. URL https://www.theregister.com/2022/08/01/optane_intel_cancellation/.

[264] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, October 2018. ISSN 2475-1421. doi: 10.1145/3276507.

[265] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA): 1–27, October 2019. ISSN 2475-1421. doi: 10.1145/3360561.

[266] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, January 2020. ISSN 2475-1421. doi: 10.1145/3371079.

[267] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, Portland, OR, USA, June 2019. IEEE. ISBN 978-1-72810-057-9. doi: 10.1109/DSN.2019.00028.

[268] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 392–407, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483550.

[269] Erik Rieger. SAP HANA with Intel Optane Persistent Memory on VMware vSphere, 2020. URL https://blogs.vmware.com/apps/2020/06/sap-hana-with-intel-optane-dc-persistent-memory-on-vmware-vsphere.html.

[270] J. Rosenberg and David Abramson. MONADS-PC: A capability based workstation to support software engineering. *Proc. 18th Hawaii International Conference on System Sciences*, January 1985.

[271] John Rosenberg, Frans Henskens, Fred Brown, Ron Morrison, and David Munro. Stability in a Persistent Store Based on a Large Virtual Memory. In C. J. van Rijsbergen, John Rosenberg, and J. Leslie Keedy, editors, *Security and Persistence*, pages 229–245. Springer London, London, 1990. ISBN 978-3-540-19646-4 978-1-4471-3178-6. doi: 10.1007/978-1-4471-3178-6_16.

[272] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, 39(9):62–69, September 1996. ISSN 0001-0782, 1557-7317. doi: 10.1145/234215.234472.

[273] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on*

Operating Systems Principles - SOSP '91, pages 1–15, Pacific Grove, California, United States, 1991. ACM Press. ISBN 978-0-89791-447-5. doi: 10.1145/121132.121137.

[274] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42 (2):34–40, 2017.

[275] Haytham Salhi, Feras Odeh, Rabee Nasser, and Adel Taweel. Open Source In-Memory Data Grid Systems: Benchmarking Hazelcast and Infinispan. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 163–164, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4404-3. doi: 10.1145/3030207.3053671.

[276] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles - SOSP '93*, pages 146–160, Asheville, North Carolina, United States, 1993. ACM Press. ISBN 978-0-89791-632-5. doi: 10.1145/168619.168631.

[277] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, Berkeley, CA, 2020. ISBN 978-1-4842-4931-4 978-1-4842-4932-1. doi: 10.1007/978-1-4842-4932-1.

[278] Steve Scargall. *Operating System Support for Persistent Memory*, pages 31–54. Apress, Berkeley, CA, 2020. ISBN 978-1-4842-4931-4 978-1-4842-4932-1. doi: 10.1007/978-1-4842-4932-1_3.

[279] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 1–8, Kohala Coast HI USA, August 2015. ACM. ISBN 978-1-4503-3713-7. doi: 10.1145/2803140.2803144.

[280] Russell Sears and Eric Brewer. Stasis: Flexible Transactional Storage. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association. URL https://www.usenix.org/conference/osdi-06/stasis-flexible-transactional-storage.

[281] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. PMNet: In-Network Data Persistence. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817, Valencia, Spain, June 2021. IEEE. ISBN 978-1-66543-333-4. doi: 10.1109/ISCA52012.2021.00068.

[282] Gal Sela and Erez Petrank. Durable Queues: The Second Amendment. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 385–397, Virtual Event USA, July 2021. ACM. ISBN 978-1-4503-8070-6. doi: 10.1145/3409964.3461791.

[283] Margo Seltzer, Virendra Marathe, and Steve Byan. An NVM Carol: Visions of NVM Past, Present, and Future. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 15–23, Paris, April 2018. IEEE. ISBN 978-1-5386-5520-7. doi: 10.1109/ICDE.2018.00011.

[284] Jonathan S. Shapiro and Jonathan Adams. Design Evolution of the EROS Single-Level Store. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 59–72, USA, 2002. USENIX Association. ISBN 1-880446-00-6.

[285] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. *ACM SIGOPS Operating Systems Review*, 33(5):170–185, December 1999. ISSN 0163-5980. doi: 10.1145/319344.319163.

[286] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing - PODC '95*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press. ISBN 978-0-89791-710-0. doi: 10.1145/224964.224987.

[287] Yang Shi. Another Approach to Use PMEM as NUMA Node, 2019. URL https://lwn.net/Articles/783811/.

[288] Thomas Shull, Jian Huang, and Josep Torrellas. AutoPersist: An easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 316–332, Phoenix AZ USA, June 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314608.

[289] Thomas Shull, Jian Huang, and Josep Torrellas. QuickCheck: Using speculation to reduce the overhead of checks in NVM frameworks. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE 2019*, pages 137–151, Providence, RI, USA, 2019. ACM Press. ISBN 978-1-4503-6020-3. doi: 10.1145/3313808.3313822.

[290] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 1–16, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

[291] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '13*, page 485, Houston, Texas, USA, 2013. ACM Press. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451169.

[292] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In C. J. van Rijsbergen, Antonio Albano, and Ron Morrison, editors, *Persistent Object Systems*, pages 11–33. Springer London, London, 1993. ISBN 978-3-540-19800-0 978-1-4471-3209-7. doi: 10.1007/978-1-4471-3209-7_2.

[293] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: Good, fast, cheap persistence for C++. *ACM SIGPLAN OOPS Messenger*, 4(2):145–147, April 1993. ISSN 1055-6400. doi: 10.1145/157710.157737.

[294] Ryan Smith. Intel to Wind Down Optane Memory Business - 3D XPoint Storage Tech Reaches Its End. *AnandTech.com*,

July 2022. URL https://www.anandtech.com/show/17515/intel-to-wind-down-optane-memory-business.

[295] SNIA. *NVM Programming Model (NPM)*. 2017. URL https://www.snia.org/tech_activities/standards/curr_standards/npm.

[296] Frank G. Soltis. *Inside the as/400*. Twenty Ninth Street Press, 1996. ISBN 1-882419-13-8.

[297] Frank G. Soltis. *Fortress Rochester: The inside Story of the IBM iSeries*. NEWS/400 Books, Loveland, CO, 2001. ISBN 978-1-58304-083-6.

[298] Xinyang (Kevin) Song, Sihang Liu, and Gennady Pekhimenko. Persistent Memory – A New Hope. *ACM SIGARCH Blog*, September 2022. URL https://www.sigarch.org/persistent-memory-a-new-hope/.

[299] Susan Spence. Managed Data Structures, 2016. URL https://github.com/HewlettPackard/mds.

[300] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, August 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407836.

[301] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981. ISSN 0001-0782, 1557-7317. doi: 10.1145/358699.358703.

[302] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *ACM SIGMOD Record*, 15(2):340–355, June 1986. ISSN 0163-5808. doi: 10.1145/16856.16888.

[303] Zi Fan Tan, Jianan Li, Haris Volos, and Terence Kelly. Persistent Scripting. In *NVMW '22: 13th Annual Non-Volatile Memories Workshop*, San Diego, California, USA, 2022. URL http://nvmw.ucsd.edu/program/#paper-35.

[304] Satish M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems*, OODS '86, pages 148–159, Washington, DC, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-0734-3.

[305] Adam Thompson and CJ Newburn. GPUDirect Storage: A Direct Path Between Storage and GPU Memory, August 2019. URL https://developer.nvidia.com/blog/gpudirect-storage/.

[306] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The Aurora Single Level Store Operating System. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 788–803, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483563.

[307] Preetika Tyagi. Enable Cassandra for Persistent Memory, 2017. URL https://issues.apache.org/jira/browse/CASSANDRA-13981.

[308] Kenton Varda. Protocol Buffers: Google's Data Interchange Format. Technical report, Google, June 2008. URL http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[309] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9.

[310] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '11*, page 91, Newport Beach, California, USA, 2011. ACM Press. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950379.

[311] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–362, Phoenix AZ USA, June 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314650.

[312] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/wang-qing.

[313] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. 2022. doi: 10.48550/ARXIV.2209.09459.

[314] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. The case for distributed shared-memory databases with RDMA-enabled memory disaggregation. *Proceedings of the VLDB Endowment*, 16(1):15–22, September 2022. ISSN 2150-8097. doi: 10.14778/3561261.3561263.

[315] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, Paris, April 2018. IEEE. ISBN 978-1-5386-5520-7. doi: 10.1109/ICDE.2018.00049.

[316] William Wang and Stephan Diestelhorst. Persistent Atomics for Implementing Durable Lock-Free Data Structures for Non-Volatile Memory (Brief Announcement). In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 309–311. ACM, 2019. doi: 10.1145/3323165.3323166.

[317] Scott Watanabe. *Solaris 10 ZFS Essentials*. Prentice Hall Press, USA, 1st edition, 2010. ISBN 0-13-700010-3.

[318] Jim Webber. A Programmatic Introduction to Neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 217–218, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 978-1-4503-1563-0. doi: 10.1145/2384716.2384777.

[319] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A Fast, General System for Buffered Persistent Data Structures. In *50th International Conference on Parallel Processing*, pages 1–11, Lemont IL USA, August 2021. ACM. ISBN 978-1-4503-9068-2. doi: 10.1145/3472456.3472458.

[320] Seth J. White and David J. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data - SIGMOD '94*, pages 395–406, Minneapolis, Minnesota, United States, 1994. ACM Press. ISBN 978-0-89791-639-4. doi: 10.1145/191839.191919.

[321] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 19(4):6–13, July 1991. ISSN 0163-5964. doi: 10.1145/122576.122577.

[322] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Transactions on Architecture and Code Optimization*, 9(4): 1–24, January 2013. ISSN 1544-3566, 1544-3973. doi: 10.1145/2400682.2400689.

[323] Rafael Winterhalter. ByteBuddy, 2014. URL https://bytebuddy.net/#/.

[324] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase Change Memory. *Proceedings of the IEEE*, 98(12):2201–2227, December 2010. ISSN 0018-9219, 1558-2256. doi: 10.1109/JPROC.2010.2070050.

[325] Joel Van Der Woude and Matthew Hicks. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude.

[326] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A KVS For Any Scale. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2019. ISSN 1041-4347, 1558-2191, 2326-3865. doi: 10.1109/TKDE.2019.2898401.

[327] Kai Wu and Dong Li. Unimem: Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Main Memory for High Performance Computing. *Journal of Computer Science and Technology*, 36(1):90–109, January 2021. ISSN 1000-9000, 1860-4749. doi: 10.1007/s11390-020-0942-z.

[328] Michael Wu and Willy Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In Forest Baskett and Douglas W. Clark, editors, *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, pages 86–97. ACM Press, 1994. doi: 10.1145/195473.195506.

[329] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 70–83, Williamsburg VA USA, March 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173201.

[330] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. GCPersist: An efficient GC-assisted lazy persistency framework for resilient Java applications on NVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–14, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7554-2. doi: 10.1145/3381052.3381318.

[331] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-DIMM buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 488–505, Rennes France, March 2022. ACM. ISBN 978-1-4503-9162-7. doi: 10.1145/3492321.3519556.

[332] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 323–338. USENIX Association, 2016. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[333] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, Shanghai China, October 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132761.

[334] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–439, Providence RI USA, April 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304077.

[335] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES+ISSS '11*, page 325, Taipei, Taiwan, 2011. ACM Press. ISBN 978-1-4503-0715-4. doi: 10.1145/2039370.2039420.

[336] Albert Mingkun Yang and Tobias Wrigstad. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Transactions on Programming Languages and Systems*, 44(4):1–34, December 2022. ISSN 0164-0925, 1558-4593. doi: 10.1145/3538532.

[337] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and*

Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019, pages 221–234. USENIX Association, 2019. URL https://www.usenix.org/conference/fast19/presentation/yang.

[338] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/yang.

[339] 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 *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, February 2015. USENIX Association. ISBN 978-1-931971-20-1. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang.

[340] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 343–358, Online Event United Kingdom, April 2021. ACM. ISBN 978-1-4503-8334-9. doi: 10.1145/3447786.3456246.

[341] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, Hong Kong, December 2017. IEEE. ISBN 978-1-5386-0692-6. doi: 10.1109/CloudCom.2017.14.

[342] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[343] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, October 2016. ISSN 0001-0782, 1557-7317. doi: 10.1145/2934664.

[344] Dimitry Zavalishin. PhantomOS. URL https://phantomos.org.

[345] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. ReplayCache: Enabling Volatile Caches for Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM Inter-*

national Symposium on Microarchitecture, pages 170–182, Virtual Event Greece, October 2021. ACM. ISBN 978-1-4503-8557-2. doi: 10.1145/3466752.3480102.

[346] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. NBTree: A lock-free PM-friendly persistent B $^{+}$ -tree for eADR-enabled PM systems. *Proceedings of the VLDB Endowment*, 15(6):1187–1200, February 2022. ISSN 2150-8097. doi: 10.14778/3514061.3514066.

[347] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

[348] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, July 2015. ISSN 2150-8097. doi: 10.14778/2809974.2809984.

[349] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1029–1046. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhang-wen.

[350] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Santa Clara, CA, USA, May 2015. IEEE. ISBN 978-1-4673-7619-8. doi: 10.1109/MSST.2015.7208275.

[351] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 207–219. USENIX Association, 2019. URL https://www.usenix.org/conference/fast19/presentation/zheng.

[352] Shengan Zheng, Jingyu Wang, Dongliang Xue, Jiwu Shu, and Linpeng Huang. Hydra: A Decentralized File System for Persistent Memory and RDMA Networks. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4192–4206, December 2022. ISSN 1045-9219, 1558-2183, 2161-9883. doi: 10.1109/TPDS.2022.3180369.

[353] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/zuo.

[354] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, October 2019. ISSN 2475-1421. doi: 10.1145/3360554.

ECOLE
DOCTORALE

Titre : Intégration de la mémoire persistante en Java

Mots clés : Mémoire non-volatile, JAVA, mémoire persistante

Résumé : L'arrivée de la mémoire non-volatile (NVMM) sur le marché propose une alternative rapide et durable pour le stockage de données, avec des performances considérablement accrues par rapport aux supports traditionnels, à savoir SSD et disques durs. La NVMM est adressable à la granularité de l'octet, une caractéristique unique qui permet de maintenir des structures de données complexes par le biais d'instructions mémoires standards, tout en étant résistante aux pannes système et logiciels. Néanmoins, gérer correctement la persistance des données est bien plus compliquée que de simples manipulations mémoire. De plus, chaque bug en NVMM peut désormais compromettre l'intégrité des données ainsi que leur récupération, et il faut donc prendre grand soin quant à sa programmation.
Ainsi, de nouvelles abstractions de programmation pour la persistance et l'intégration dans les langages et compilateurs sont nécessaires afin de faciliter l'usage de la mémoire non-volatile. Cette thèse se penche sur ce problème général. Nous expliquons comment intégrer la mémoire persistante dans les langages de programmation managés, et présentons J-NVM, un framework pour accéder efficacement à la NVMM en Java.
Avec J-NVM, nous montrons comment concevoir une interface d'accès simple, complète et efficace qui lie les spécificités de la persistance sur NVMM avec la programmation orientée objet. En particulier, J-NVM offre des fonctionnalités pour rendre durable des objets Java avec des sections de code atomiques en cas de panne. J-NVM est construit sans apporter de modifications à l'environnement d'exécution de Java, ce qui favorise sa portabilité aux divers environnements d'exécution de Java.
En interne, J-NVM s'appuie sur des objets mandataires qui réalisent des accès directs à la NVMM, gérée comme une mémoire hors-tas. Ce canevas fournit également une bibliothèque de structures de données optimisées pour la NVMM qui restent cohérentes à la suite de redémarrages ou d'arrêts impromptus.
Au cours de cette thèse, nous évaluons J-NVM en réimplémentant la couche de stockage d'Infinispan, une base de données open-source de niveau industriel. Les résultats obtenus avec les benchmarks TPC-B et YCSB, montrent que J-NVM est systématiquement plus rapide que les autres approches existant à l'heure actuelle pour accéder à la NVMM en Java.

Title : A support for Persistent Memory in Java

Keywords : non-volatile main memory, JAVA, persistent memory

Abstract : Recently released non-volatile main memory (NVMM), as fast and durable memory, dramatically increases storage performance over traditional media (SSD, hard disk). A substantial and unique property of NVMM is byte-addressability – complex memory data structures, maintained with regular load/store instructions, can now resist machine power-cycles, software faults or system crashes. However, correctly managing persistence with the fine grain of memory instructions is laborious, with increased risk of compromising data integrity and recovery at any misstep. Programming abstractions from software libraries and support from language runtime and compilers are necessary to avoid memory bugs that are exacerbated with persistence.
In this thesis, we address the challenges of supporting persistent memory in managed language environments by introducing J-NVM, a framework to efficiently access NVMM in Java. With J-NVM, we demonstrate how to design an efficient, simple and complete interface to weave NVMM-devised persistence into object-oriented programming, while remaining unobtrusive to the language runtime itself. In detail, J-NVM offers a fully-fledged interface to persist plain Java objects using failure-atomic sections. This interface relies internally on proxy objects that intermediate direct off-heap access to NVMM. The framework also provides a library of highly-optimized persistent data types that resist reboots and power failures. We evaluate J-NVM by implementing a persistent backend for Infinispan, an industrial-grade data store. Our experimental results, obtained with a TPC-B like benchmark and YCSB, show that J-NVM is consistently faster than other approaches at accessing NVMM in Java.

Institut Polytechnique de Paris
91120 Palaiseau, France