Persistent Memory: A Survey of Programming Support and Implementations

ALEXANDRO BALDASSIN, São Paulo State University (Unesp), Institute of Geosciences and Exact Sciences, Brazil
JOÃO BARRETO, DANIEL CASTRO, and PAOLO ROMANO, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

The recent rise of byte-addressable non-volatile memory technologies is blurring the dichotomy between memory and storage. In particular, they allow programmers to have direct access to persistent data instead of relying on traditional interfaces such as file and database systems. However, they also bring new challenges, as a failure may render the program in an unrecoverable and inconsistent state. Consequently, a lot of effort has been put by both industry and academia into making the task of programming with such memories easier while, at the same time, efficient from the runtime perspective. This survey summarizes such body of research, from the abstractions to the implementation level. As persistent memory is starting to appear commercially, the state-of-the-art research condensed here will help investigators to quickly stay up to date while also motivating others to pursue research in the field.

CCS Concepts: • **Information systems** → *Information storage systems*; • **Hardware** → *Memory and dense storage*; • **General and reference** → **Surveys and overviews**.

Additional Key Words and Phrases: persistent memory, failure-atomic sections, persistent heap

ACM Reference Format:
Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. 2021. Persistent Memory: A Survey of Programming Support and Implementations. *ACM Comput. Surv.* 1, 1, Article 1 (January 2021), 37 pages. https://doi.org/10.1145/3465402

1 INTRODUCTION

Traditionally, computer systems have relied on non-volatile mass storage devices whose performance is vastly inferior to volatile main memory (DRAM). Because of the slow performance, these block devices are not attached to the processor's memory bus and thus applications do not have direct access to the data contents. Instead, programmers rely on standard Application Programming Interfaces (API) provided by Operating Systems (OS) and Database Management Systems (DBMS).

There is, however, a mismatch between most programming languages and the standard APIs for persistence storage. For instance, if some data structure is required to be durable, programmers have to translate its volatile representation into a format that can be stored in the non-volatile device, a process known as *serialization*. If the same data structure needs to be restored later, the reverse action needs to be performed via a *deserialization* mechanism. Therefore, persistency adds

Authors' addresses: Alexandro Baldassin, alexandro.baldassin@unesp.br, São Paulo State University (Unesp), Institute of Geosciences and Exact Sciences, Avenida 24A, 1515, 13506-900, Rio Claro, São Paulo, Brazil; João Barreto; Daniel Castro; Paolo Romano, INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Rua Alves Redol, 9, 1000-029, Lisbon, Portugal, joao.barreto@tecnico.ulisboa.pt,daniel.castro@tecnico.ulisboa.pt,romano@inesc-id.pt.

0360-0300/2021/1-ART1 $15.00
https://doi.org/10.1145/3465402

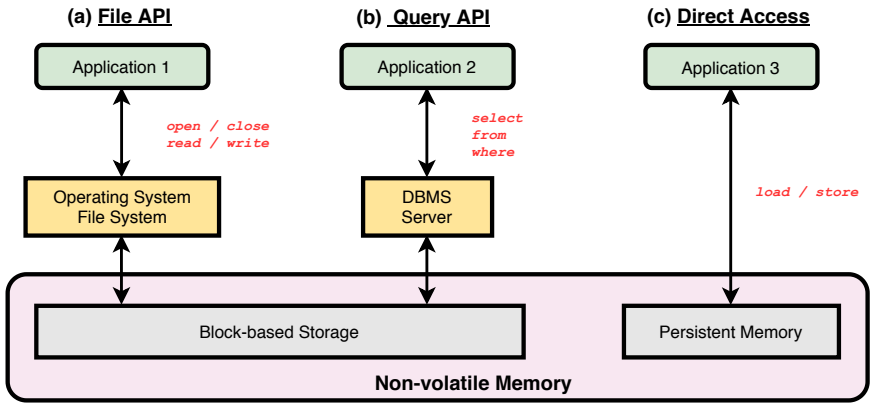


Fig. 1. Interfaces to non-volatile memory: (a) Operating System File API; (b) Database Query API; and (c) Load/Store instructions. While (a) and (b) have been standardized and are well-known, (c) is considered a disruptive model because it allows direct access to persistence data and requires new programming models.

programming effort, is more error-prone and, at the same time, degrades performance due to the translation and interaction between volatile and non-volatile domains.

Recent advances in non-volatile memory technology promise to completely change the traditional way of dealing with persistent data [9, 46]. These new devices: (i) have a performance close to DRAM and hence are connected directly to the processor's memory bus; (ii) provide byte-addressability, allowing applications to directly access them by issuing load and store instructions; (iii) are denser and thus can provide higher capacity; (iv) offer low cost per bit; and, (v) dissipate less energy compared to DRAM. The main representatives of such new technology are Phase Change Memory (PCM) [44, 87, 88, 150], Resistive RAM (ReRAM) [1], and Spin Torque Transfer Magnetic RAM (STT-MRAM) [2, 45]. In early April 2019, Intel Optane DC Persistent Memory became the first byte-addressable persistent memory module commercially available [142]. Based on the Intel/Micron 3D XPoint technology which, although not officially disclosed, displays characteristics similar to PCM [134], Optane DC has been gathering many research endeavors. Recent studies [120, 155] disclosed performance features showing slower access latency (2x-3x) and more limited bandwidth (2x-6x for reads, 6x-18x for writes) compared to DRAM. Moreover, it is expected that all of these NVM technologies will suffer from finite write endurance.

1.1 Terminology and Scope

Like with all new technology, the terminology used in different publications to refer to this new byte-addressable non-volatile memory diverges. Some examples include: *Storage Class Memory* (SCM), *Non-Volatile Memory* (NVM), *Non-Volatile RAM* (NVRAM), *Byte-addressable Persistent RAM* (BPRAM), *Non-Volatile Main Memory* (NVMM), *Persistent MEMory* (PMEM), *Non-Volatile DIMM* (NVDIMM), and *Non-Volatile Byte-addressable Memory* (NVBM). We will adopt the SNIA[1] (Storage Networking Industry Association) terminology, as specified in the NVM Programming Model (NPM) [139], and use **Persistent Memory**, or **PM**, when referring to byte-addressable persistent memory. **Non-Volatile Memory**, or **NVM**, is used in a more general context to refer to any persistent media that is based on memory, such as flash memory.

[1]SNIA is an organization that is leading the development of specifications for NVM and is supported by over 150 industry partners including Intel, Microsoft, Red Hat, HP, and VMware (https://www.snia.org/forums/cmsi/nvmp).

Fig. 1 illustrates three different interfaces to non-volatile memory. Applications 1 and 2 access the block-based storage through an API, respectively, the OS file system and a DBMS server. Despite being consolidated and robust interfaces, they lack flexibility and do not integrate seamlessly with conventional programming languages [79]. In contrast, Application 3 has direct access to the persistent memory by using load/store instructions, which allows durable data to be addressed just like volatile memory. Whereas scenarios (a) and (b) depict well-established and time-tested approaches, scenario (c) is becoming feasible only now thanks to the advent of the new persistent memory and research on programming with this new technology.

Contrary to traditional APIs, direct access to durable data exposes the full power of persistent memory to programmers. As the old saying goes, with great power comes great responsibility, and it is no different here. In particular, data consistency is a major concern because a system crash could leave the program in an unrecoverable state after reboot. The fact that CPU caches are volatile in current computing systems only aggravates the problem, as the order in which updates get persisted may be altered by the hardware. Without the appropriate programming support, coping with this problem involves direct interaction with low-level hardware mechanisms. Unfortunately, it is challenging to write correct and efficient code using such low-level mechanisms [93, 94, 113]. If a programmer misses any necessary persist barrier, the program becomes prone to data inconsistencies. In contrast, a conservative programmer that overuses persist barriers will observe drastic performance losses due to the high cost of such instructions [13].

Given the recent availability of persistent memory and the lack of programming support, a number of papers have been published in the last few years discussing programming models and efficient runtime systems. We believe that now is the opportune moment to summarize the main aspects of the proposed systems and provide practitioners with a concrete overview of the area, as well as identify open questions that can serve as drivers for future research in this field.

Previous surveys in the context of PM focus on other specific subjects. Suzuki and Swanson [141] and Xia [154] present a survey of NVM technologies and their respective challenges. Architectural integration issues arising from these different technologies are further surveyed by Boukhobza [18]. The first survey to address software aspects of NVM is due to Mittal and Vetter [103]. Their focus, however, is not on programming models (only a page of the entire survey is devoted to this topic) but rather on aspects such as energy efficiency and the design of hybrid memory systems. Seltzer et al. [134] provide a brief historical perspective of PM, focusing on the fundamental technologies and techniques developed up to that point (circa 2018). More recently, Puglia et al. [122] offer a systematic mapping study of storage and file systems for NVM. These works are complementary to ours, as the goal of this survey is to provide a state-of-the-art summary of programming models and their respective implementation techniques.

1.2 Structure of the Survey

This survey is built around two main topics concerning the programmability of persistent systems: (i) **support for failure-atomic updates**, which encompasses the main building blocks for consistently updating persistent data; and (ii) **access and heap management**, comprising the interface for accessing the PM device, identifying and managing persistent data. The survey first discusses these topics from the PM programmer's perspective, describing which programming constructs have been proposed to correctly update persistent data, as well as the different methods to access and manage it. After the programmer's view is presented, the various implementation techniques developed to efficiently provide failure-atomic updates and access the PM heap are discussed.

Table 1. Number of papers collected for this survey starting from 2011 that considered (i) failure-atomic updates, (ii) access and heap management, and (iii) respective implementation aspects.

	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	Total
F.A. Updates	2	2	1	1	1	3	1	4	1	2	18
Access/Manag.	2	2	1	1	3	3	2	4	3	1	22
Implementation	2	2	4	2	9	11	12	14	10	14	80

For conducting this survey, we considered papers published in the main conferences in the field (ASPLOS, PLDI, PPoPP, EuroSys, ISCA, MICRO, to name a few) starting from 2011[2]. Table 1 shows the number of collected papers split into the main topics covered by this survey, according to the year they were published. Some works, such as Mnemosyne [143] and NV-Heaps [30], cover all the main topics and thus are computed in all of them. Therefore, it is possible for a paper to be counted in multiple categories. The criterion we adopted to classify a paper is based on the most important contribution presented. The table shows an increasing research interest in programming models and runtimes for PM, particularly from 2015 onwards. Most papers deal with implementation strategies, focusing on optimization aspects (including hardware support). This is to be expected, as interfaces (e.g., transaction) tend to change less frequently than the implementations.

This survey first presents some background concepts (§2), followed by the sections on failure-atomic updates (§3) and persistent heap management (§4). The implementation aspects for these main subjects are discussed at the end of each respective section. Finally, we summarize and conclude the survey (§5).

2 BACKGROUND

Since the very beginning, the memory systems of electronic computers employed a two-level organization motivated by contrasting technological characteristics (see Fig. 2a). Primary storage, more commonly known as **main memory**, has fast access time but is relatively small and expensive. On the other hand, **secondary memory** (or external storage) is known for its large capacity and lower cost per bit, but its access time tends to be much slower. As a consequence, only main memory is directly connected to the processor (via a memory bus), whereas secondary storage is usually connected to a slower I/O bus. An important difference between main and secondary memories is that, while the former is volatile (information is lost when not powered), the latter can retain the information. In this section we start by describing the role of PM in the memory hierarchy (§2.1), the concept of persistence domains (§2.2), failure models (§2.3) and conclude with a discussion of memory persistency models (§2.4).

2.1 PM and Memory Hierarchy

The introduction of a new type of memory that has characteristics of both main memory (relatively fast, byte-addressable) and secondary memory (relatively large, non-volatile) allows different organizations of the memory hierarchy, as shown in Fig. 2. The first scenario that can be considered is to use PM as a simple replacement for a block-based device, such as the Hard Disk (HD) in Fig. 2a. Although only minimal changes to the software stack are required (a PM-aware block-based driver), this configuration does not exploit the full potential of PM due to the associated OS overheads [24]. Therefore, only options from (b) through (d) are worth considering in this survey. Notice that these configurations may still use some kind of secondary memory (represented in light gray).

[2]This is the year that marked the publication of the two seminal papers addressing programmability of PM systems, both presented at ASPLOS: Mnemosyne [143] and NV-Heaps [30].

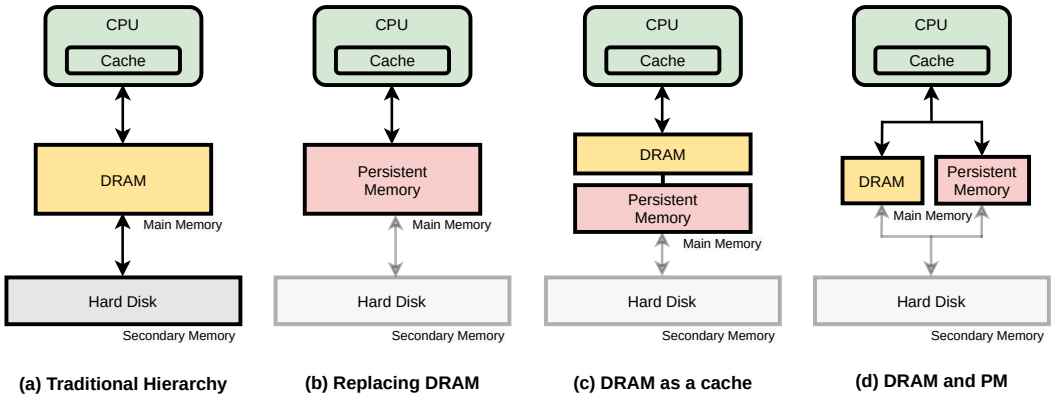


Fig. 2. Different memory hierarchy organizations: (a) a traditional two-level memory system, with both main (volatile) and secondary (block-based, non-volatile) memories; (b) PM replacing DRAM as main memory; (c) vertical integration: DRAM working as a cache to PM; and (d) horizontal integration: data can be placed in either DRAM or PM. Configurations (b), (c), and (d) could potentially provide a block-based backstore.

One option is for PM to replace DRAM entirely, as shown in Fig. 2b. This configuration might work as a cost-effective, albeit slower, alternative for capacity-only occasions. Although possible, it is very unlikely that DRAM will be completely replaced in the near future, as the performance gap between DRAM and PM is still noticeable as recent studies indicate [120, 155]. For instance, current machines that support Intel Optane DC PM require at least one DRAM DIMM [149]. As a result, systems will most likely adopt a hybrid approach with both DRAM and PM integrated in a vertical or horizontal fashion [18]. In the former case, depicted in Fig. 2c, DRAM effectively works as a cache to PM, potentially hiding the higher PM write latency. This configuration is transparent to software (no changes are required) and makes use of the large capacity provided by PM, but cannot exploit the persistent property of PM. Intel Optane DC PM supports this model when configured in *Memory Mode* [120, 155].

The last organization places DRAM and PM at the same level in the hierarchy, as shown in Fig. 2d. This is, by far, the most common configuration assumed by the research papers covered in this survey. This organization raises new questions regarding consistent updates and data placement, which need to be addressed by the software stack. Intel Optane DC PM supports this model when configured in *App Direct mode*. Although most of the state-of-the-art works covered in this survey follow this organization, it is worth noticing that a few proposals [35, 72] further assume that some levels of the cache hierarchy are also persistent.

2.2 Persistence Domains

Computer platforms with PM devices support the concept of **Persistence Domain** (PD) (also known as **power-fail protected domain**) [131, 139], a region of a computer system that preserves the content of any data reaching such region. Therefore, once data reaches the PD, it can be recovered after a system restart. Notice that a PD is usually not limited to a PM device, that is, volatile devices that can hold the state long enough so that it can reach a persistent device are also included. In fact, current platforms require at least a PD known as **Asynchronous DRAM Refresh** (ADR), which also includes the Writing Pending Queue (WPQ) of the memory controller [128, 129]. If a power failure occurs while data is still in the WPQ, the platform is required to have enough stored energy to flush it to the persistent media. Therefore, it is enough for system software to

make sure that data has reached the memory controller in order to provide persistence under ADR. Current research assumes at least ADR and it is the PD model that will be implicitly considered throughout the rest of this survey.

The processor caches are also included in the PD known as **enhanced Asynchronous DRAM Refresh** (eADR) [131]. Such functionality requires additional stored energy, such as an external power supply or batteries. One clear advantage of platforms supporting eADR is that applications are not required to perform explicit flush operations which, in turn, might simplify programming effort as well as improve performance. As of the time of writing, eADR was still not available in mainstream commercial systems but Intel was about to release a platform featuring it with its third generation of Xeon processors and Optane 200 series [107].

The work on Whole-System Persistence [110] (WSP) places even the processor registers in a PD, creating an entire persistent system. WSP assumes only PM (no DRAM), as in Fig. 2b, and also relies on the residual energy from the system power supply to flush all data to PM. On recovery, the state is restored instantaneously and transparently. Although an interesting idea, WSP does not solve all the problems as even with WSP there are non-trivial issues related to recovering peripheral device state and how to deal with software crashes.

Finally, Zardoshti et al. [157] investigated new PDs in which DRAM is also taken into account. In the first proposal, PDRAM, the entire DRAM is persistent. PDRAM has the same issue as WSP in which it is somewhat idealistic and might also necessitate some form of external battery. PDRAM-lite, the second proposal, considers only a small fraction of DRAM as persistent. Zardoshti et al. showed that PDRAM-lite can improve the performance of some PM systems since DRAM can be used for storing logging information and still guarantee persistence.

2.3 Failure Models

A **failure** is a behavior that deviates from the specification. Failures might cause corruption or even permanent damage to application data. In more general terms, they can be classified into two main groups: 1) *software failures*, such as bugs that cause pointer corruption and memory allocation error; and 2) *hardware failures*, such as bit flips and power outages. An important aspect of PM systems is whether a given failure is *tolerated* or not [72, 111]. A tolerated failure does not cause data inconsistency and application code still has access to it after recovery.

One may also classify failures according to *locality*: it might range from affecting only a single thread to affecting a subgroup of them, which can then recover and continue execution independently; or affect the whole system, in which case we have a **full system failure**. Although more restricted, real-world systems usually assume full system failure. Failures can also be categorized according to whether they abruptly halt thread/core execution (usually referred to as **fail-stop failures**), or first cause data corruption [111]. For instance, an illegal instruction failure immediately halts execution, whereas memory corruption failures may corrupt critical application data.

A **failure model** specifies which failures are tolerated and whether they affect the whole system or not. Persistence systems provide different failure models. For instance, some systems do not tolerate (or at least provide only partial support for) software crashes [72, 92, 110]. Yet others tolerate even media and memory corruption [159].

2.4 Memory Persistency Models

At the Instruction Set Architecture (ISA) level, a store instruction is responsible for writing a given value V to a memory location L. From the previous discussion, we know that V is durable when it reaches a PD. However, V may need to pass through several intermediate (and volatile) storage layers before it reaches a PD, such as the processor store buffer, multiple levels of cache memory, and the memory controller buffer. Therefore, one cannot in general assume that V will be durable

immediately after the store instruction has retired. As such, it is convenient to differentiate a **store** from a **persist**: *a persist operation implies that the corresponding store is durable*, that is, V has reached a PD.

In multiprocessor systems, **memory consistency** [62] restricts the visible order of memory operations (loads and stores) among the cores. Analogously, Pelley et al. [119] introduced the term **memory persistency** to account for restrictions on the *order of persists*. Just like a memory consistency model must provide programmers with barriers to enforce the visibility of memory operation ordering with respect to other processors, memory persistency barriers constrain the visible order of persists with respect to failures. Memory persistency is therefore a natural extension of memory consistency for systems with PM. It is important to highlight that a persistency model specifies an interface, not an implementation. Therefore, a given model may allow completely different implementations at the microarchitecture level.

Pelley et al. [119] split persistency models into two main classes: **strict** and **relaxed** persistency. With **strict persistency**, the consistency model itself specifies persist ordering: the persist order *matches* the order in which stores to volatile memory become visible. This model is attractive from the programmers' perspective since they only need to reason about a single model (for both consistency and persistency), but implementing strict persistency is usually inefficient as it introduces frequent stalls due to ordering constraints. For instance, under sequential consistency, a store would only become visible after the previous one was persisted. **Buffered strict persistency** is an optimization that can alleviate the issue by allowing the persistent state to lag behind the volatile state, while still guaranteeing strict ordering of persists and visible side effects. Finally, **relaxed persistency** allows for larger performance improvements by *decoupling* the persistency model from the consistency model. In other words, the order of persists may be different from the visible order of stores. However, it complicates the programming model, since a new set of persistent barriers are introduced to control persist ordering.

A relaxed persistency model worth discussing is **epoch persistency** [35, 73, 123]. An epoch is a group of instructions separated by a new *persist barrier*. Persists are ordered according to their corresponding epoch order: stores in epoch N can only be persisted after all stores from epoch $N-1$. Within an epoch, persists are allowed to reorder (relatively to store order). In this strict version of epoch consistency, an epoch is only allowed to start when all stores belonging to the previous one have persisted. **Buffered epoch persistency** allows an epoch to start without waiting for the persists from the previous epoch (persistent state may lag behind).

The important work of Pelley et al. on memory persistency considered Sequential Consistency (SC) as the underlying consistency model, from which persistency models were devised. Several other works proposed persistency model implementations that complement more realistic relaxed consistency models, such as Release Consistency (RC) [73, 84] and Total Store Order (TSO) [35, 75, 108, 123]. Persistency models for existing ISAs, such as x86 or ARM, have also been discussed in the literature [42, 73, 83, 124, 125, 135, 136]. Consider the Intel x86 ISA extensions for PM [70], for instance. The instruction `CLWB` (Cache Line Write Back) initiates a write-back of the specified cache line but does not wait until the corresponding data reaches the PD. A `CLWB` is ordered with respect to older writes to the same cache line but is *not ordered* with respect to other `CLWB`. Intel x86 requires an `SFENCE` barrier to force ordering between two or more `CLWB`. `SFENCE` also makes sure the respective cache line reaches the PD, therefore playing a double role (ordering and persistence).

Fig. 3 shows code examples under strict and epoch persistency models for idealized machines and the equivalent for Intel x86. In Fig. 3a, `FOO` is implicitly persisted before `BAR` in a machine implementing strict persistency (ideal). Although Intel x86 machines would not reorder the stores themselves, the persists could be reordered. As a result, each store has to be followed by `CLWB` and `SFENCE` if strict persistency is desired. Fig. 3b presents a similar example, where a machine

Ideal	Intel x86
`STORE FOO, 1`	`STORE FOO, 1`
`STORE BAR, 1`	**`CLWB`** `FOO`
	`SFENCE`
	`STORE BAR, 1`
	`CLWB` `BAR`
	`SFENCE`

(a) Strict Persistency

Ideal	Intel x86
`STORE FOO, 1`	`STORE FOO, 1`
`STORE BAR, 1`	`STORE BAR, 1`
`EPOCH_BARRIER`	**`CLWB`** `FOO`
`STORE FLAG, 1`	**`CLWB`** `BAR`
	`SFENCE`
	`STORE FLAG, 1`

(b) Epoch Persistency

Fig. 3. Code snippets illustrating two memory persistency models: (a) strict persistency and (b) epoch persistency. For each model, the left side shows the code expected in an ideal machine (the one implementing that model), and the right side the equivalent using Intel x86 instructions.

ideally implementing epoch persistency enforces the value of `FLAG` to be persisted only after the values for `FOO` and `BAR` (the first epoch). Such a machine would provide a persist barrier (e.g., EPOCH_BARRIER) to convey the intended behavior. In Intel x86 machines, a `CLWB` must be issued for each store, followed by a single `SFENCE`. Notice that it does not matter whether the persists of `FOO` or `BAR` are reordered, as long as they are persisted before `FLAG`, which is accomplished by an `SFENCE`. There are two important differences between the way epoch consistency is attained with Intel x86 compared to a persist barrier as in the ideal system's persist barrier. First, an epoch persist barrier implicitly orders the persists of *every* store preceding it. In the case of Intel x86, an explicit `CLWB` is required for each store. Second, a persist barrier may not affect the global visibility of subsequent stores, whereas `SFENCE` does (the following stores are only visible after the stores preceding the fence are).

3 FAILURE-ATOMIC UPDATES

Persistent memory allows applications to modify durable state without the use of an intermediate layer such as the file system or a DBMS, but programmers must ensure that updates are consistently applied to PM in the presence of failures. Persistency models provide means to reason on the ordering of single persists to PM (usually with the granularity of 8-byte words), but fall short in addressing another key problem that often arises at higher abstraction levels: how to ensure the **failure-atomicity** of multiple update operations, i.e., in the presence of a crash either all the updates are persisted or none is.

The term *atomicity* is sometimes used ambiguously to convey both the idea of ensuring that a set of updates appear as indivisible from the perspective of other concurrent threads (as used by the programming language and transactional memory communities [15]), or with regard to the PM state upon recovery after a crash (following the classic ACID properties of database transactions [56]). To avoid ambiguity, we explicitly use *failure-atomicity* to refer to the latter and *thread-atomicity* when referring to the former. A failure-atomic operation prevents partial updates to PM: either all updates are applied or the PM state is left unchanged, even in the presence of failures. Thread-atomicity is instead related to the notion of *isolation*, the key mechanism to avoid atomic sections (such as transactions, in the case of ACID) interfering with each other (and thus observing partial updates). This distinction is important because, as we shall see, not all proposals in the PM literature provide thread-atomicity.

To illustrate the issues that will be discussed in this section, consider the bank account transfer example illustrated in Fig. 4a, where a quantity is withdrawn from `FOO` (line 1) and credited to `BAR` (line 2). Using the Intel x86 persistency model as an example, lines 3-5 guarantee that data reaches

```
1  FOO = FOO-100
2  BAR = BAR+100
3  CLWB  FOO
4  CLWB  BAR
5  SFENCE
```

(a) Explicit persists

```
pSync()
  ...
  FOO = FOO-100
  BAR = BAR+100
pSync()
```

(b) FA-Epoch

```
lock(l1)
  FOO = FOO-100
  BAR = BAR+100
unlock(l1)
```

(c) Lock

```
begin_tx()
  FOO = FOO-100
  BAR = BAR+100
end_tx()
```

(d) Transaction

Fig. 4. Code snippet for a bank account transfer using: (a) explicit persists (**not** failure-atomic); (b) failure-atomic epochs defined by synchronization points; (c) lock-based critical sections; and (d) transactions.

the PD and is thus durable. However, if there is a failure at any point before line 5 completes, the transfer may be corrupted. For instance, it might happen that the value was debited from FOO but not credited to BAR, or the other way around, as the order of the persists is not enforced. Therefore, the code block in Fig. 4a is **not** failure-atomic.

Providing failure-atomicity for the example of Fig. 4a would require programmers to explicitly implement some logging mechanism (see §3.5), which is not practical. Furthermore, writing PM programs at the ISA-level is clearly inappropriate, since not only it is hard to reason about, but also makes the code dependant on a particular architecture, and hence hinders portability. Current PM systems provide high-level interfaces to ensure *failure-atomic updates of a group of persists*. We adopt and expand the terminology used by Chakrabarti et al. [25] and refer to such a group as a *Failure-Atomic SEction* (FASE)[3]. FASEs in the PM literature can be grouped into three main categories, according to the key abstraction they employ for ensuring failure-atomicity: **FA-epoch** [4], **lock**, and **transaction**. Fig. 4b,c,d illustrate the three approaches for the bank account transfer mentioned previously.

The notion of *immediate persistency*[5] is also important in the context of FASEs. A FASE provides immediate persistency if all of its updates are persisted before any code after the FASE is executed. Otherwise, we say that a FASE guarantees *buffered persistency*. Systems satisfying buffered persistency usually provide some method that applications can invoke to force all completed but pending FASEs (those whose updates have not been persisted yet) to be persisted.

In the rest of this section we first describe each type of FASE: FA-epoch (§3.1), lock (§3.2), and transaction (§3.3). We then present approaches for failure-atomicity that focus on data structures (§3.4) and discuss the main implementation techniques (§3.5).

3.1 FA-Epoch

A FA-epoch is a group of instructions separated by synchronization points (in the example of Fig. 4b, pSync). When a synchronization point is reached, all persists since the last point are guaranteed to have reached PM (assuming immediate persistency). After the synchronization point in line 5 of Fig. 4b, the bank account transfer is persisted. If there is a failure anywhere inside the FA-epoch (lines 2–4), none of the stores issued in that FA-epoch are durable. In other words, persistent data reflects the state of the system as seen from the last successful FA-epoch. Furthermore, PM systems that implement FA-epoch usually also provide an API call to allow applications to undo all the modifications since the last synchronization point.

[3]Originally, the term referred specifically to lock-based critical sections, but has been used in a more general context recently [12, 37, 72, 100].

[4]We use FA-epoch (Failure-Atomic epoch) to differentiate from the persistency model called **epoch persistency** (§2.4).

[5]In the PM literature, this concept has received different names such as *durable linearizability* [73] and *immediate durability* [85].

The epoch-based programming model was first considered by PS-Algol in the 1980's [5] as a persistent programming approach tailored to database systems. More recently, it has been adopted by SoftPM [59] and its successor LibPM [98], as well as by Failure-Atomic `msync()` (FAMS) [79, 118]. In particular, FAMS is an extension of the POSIX `mmap()` and `msync()` interfaces to PM programming. Since it relies on a consolidated interface, it could potentially be more easily integrated into legacy applications [79]. The main difference of FA-epoch compared to the other FASE abstractions described in the next sections is that they do not require an explicit block (such as lock/unlock and begin/end transaction), since a block is implicitly defined by the intervals between consecutive synchronization points. This factor can be seen as an advantage of such model, as it avoids a class of bugs resulting from failing to properly match begin/end calls [79].

SoftPM [59], LibPM [98], and FAMS [79, 118] provide FA-epochs at the library level, without any extension to the underlying programming language model. None of them provide thread-atomicity, requiring some other concurrency mechanism for synchronization in a multi-threaded environment (such as locks). Therefore, the semantics of concurrent FA-epochs are not formally defined in those libraries. A recent approach, Acquire-Release Persistency (ARP) [81], proposes extensions to the C++ memory model to incorporate epoch-ordered persists. ARP can be seen as a low-level interface from which higher-level libraries such as SoftPM, LibPM, and FAMS could be built on top. This would allow such libraries to formally specify thread-atomicity semantics.

3.2 Lock

FASEs inferred by *locks* (lock FASEs) are the most well studied and formalized in the PM literature currently [17, 25, 53, 81]. Using the example of Fig. 4c, a critical section is the region of code between the `lock` (line 1) and `unlock` (line 4) operations of the same lock object (l1). As a consequence, the bank account transfer performed in lines 2 and 3 constitutes a FASE and the transfer is thus failure-atomic (and also thread-atomic).

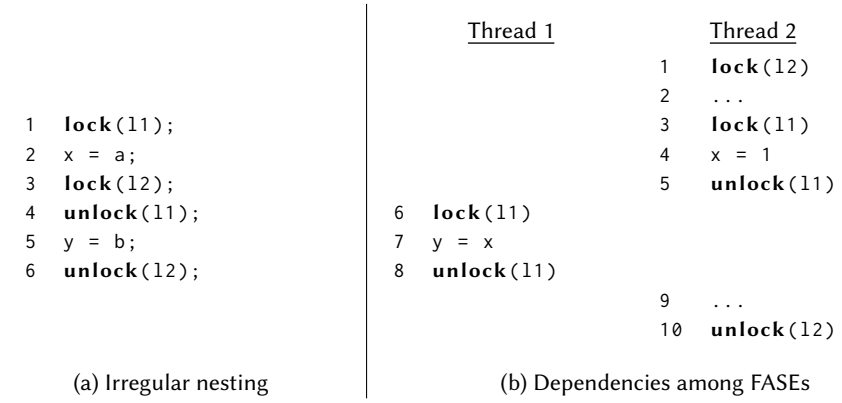
The great appeal of this approach is that it is based on a common programming idiom, facilitating the adoption of PM and also helping to bring durability to legacy code. Furthermore, FASEs in this category also provide thread-atomicity by default, as they are constructed based on synchronization operations. The majority of existing works extends the C++11 memory model to provide inter-thread persistency and assumes *Data-Race-Free* (DRF)[6] programs. The existing approaches differ on the granularity used to characterize a FASE and the semantics provided. The rest of this section discusses the main published proposals.

3.2.1 Atlas. The insight of using lock-based code to infer persistency was pioneered by the Atlas PM system [25]. One of the main goals of Atlas is to provide persistency with as few changes as possible to existing programming models. The Atlas team realized that locking primitives already conveyed enough information to infer durability and thus suggested extending such primitives with failure-atomicity semantics. For simplicity, all synchronization operations are described in terms of *lock* and *unlock* primitives.

Atlas assumes that persistent data might become inconsistent only in critical sections (a region of code between `lock` and `unlock` operations), and therefore locking primitives can be used as indicators of consistent program points. If no locks were acquired by any thread then all persistent data should be in a consistent state[7]. The semantics of FASEs constrained by critical sections can

[6] C++11, similarly to Java, guarantees sequential consistency for correctly synchronized programs, i.e., *Data-Race-Free* (DRF) programs [16]. In the case of C++11, the semantics for programs with data races is *undefined*. Two memory operations are said to be in a data race (or said to conflict) if they access the same memory location and at least one of them is a store.

[7] This assumption obviously does not apply to single-threaded applications, which will require the introduction of critical sections in the code.



(a) Irregular nesting (b) Dependencies among FASEs

Fig. 5. Subtleties of lock FASEs [17]. In (a), the FASE is determined by the outermost critical section (lines 1–6), delimited by lock and unlock operations on different lock objects (l1 and l2). In (b), numbers represent points in time for two concurrent threads. Although Thread 1's FASE is completed after instant 8, if there is a failure before Thread 2 completes its FASE (instant 10), the changes performed by Thread 1 must not be observable by the application after recovery.

be much more subtle than the bank account transfer example (Fig. 4c) might suggest. Most of the issues are related to the way critical sections can be nested. We will use the examples of Boehm and Chakrabarti [17], as presented in Fig. 5, to illustrate the sort of subtle issues that can arise. In the first case, depicted in Fig. 5a, the critical sections do not nest perfectly: the one delimited by lock l1 (lines 1–4) overlaps with the other delimited by lock l2 (lines 3–6). Since persistent data might be inconsistent during the whole region (lines 1–6), the FASE must be defined according to the *outermost* critical section.

The other scenario, exemplified in Fig. 5b, shows `Thread 1` and `Thread 2` executing concurrently (the numbers represent time instants). Notice that the FASE executed by `Thread 2` is completed only after l2 is released at instant 10. After instant 5, the value written to variable `x` is already visible externally and can be read by concurrent threads, as done by `Thread 2` in its FASE (instant 7). If a failure happens before `Thread 2` is able to finish its FASE then the updates performed by `Thread 1` must not be persisted. Hence, after recovery, the value of variable `x` must be the one before the write at instant 4. Any other threads that happened to have consumed `x` (like `Thread 1` did) must also undo any modifications that resulted from reading `x`. These dependencies among FASEs must be appropriately tracked by the PM system to avoid inconsistencies.

In principle, persistent updates outside of critical sections have no failure-atomicity guarantees as they may appear to recovery code as partially completed. Instead of disallowing such case, Atlas does provide some failure-atomicity guarantees for updates that are ordered before a persisted FASE: if such updates appear before a FASE A, and FASE A is known to be persisted, then the updates must also be persisted.

The great advantage of Atlas with respect to other lock-based approaches, such as SFR (discussed next), is that the programming model is the most natural to programmers, as it guarantees failure-atomicity at the granularity of the outermost critical section. Thus, the recovery code observes the PM state as it existed when no locks were held. However, implementing the PM programming model as originally proposed by Atlas revealed itself very challenging performance-wise [53, 64, 72, 82, 92, 153], which motivated other works discussed in the following.

3.2.2 SFRs. The approach proposed by Gogte et al. [53] relaxes somewhat the semantics proposed by Atlas by considering *Synchronization-Free Regions* (SFR) as the granularity of a FASE. An SFR is characterized by the code region delimited by two synchronization operations, such as lock and unlock. For DRF programs, SFRs are serializable. Using SFRs as the unit of failure-atomicity brings this guarantee to the recovery code, ensuring that the state after recovery reflects a consistent PM state. The difference with respect to Atlas is that Atlas, upon recovery, reverts to a state where no locks are held, whereas failure-atomic SFRs reflect the program state at the *frontier of the last synchronization operation* (assuming immediate persistency).

For programs without overlapping critical sections, Atlas' FASEs and SFR's FASEs are the same. An example is Fig. 4c, where the SFR comprises the bank transfer in lines 2 and 3. But otherwise, it is possible to observe partially completed critical sections with SFRs. For instance, consider the code for `Thread 2` in Fig. 5b. There are at least three SFR FASEs: the first delimited by instants 1 and 3, the second by 3 and 5, and the third by 5 and 10. If there is a crash anywhere between instants 6 and 9, the state after recovery will still reflect the updates before line 5 (x will be 1). If failure-atomicity of the outermost critical section is desired, programmers must implement some form of logging to roll back the state appropriately. Compared to Atlas, failure-atomic SFRs trade off ease of programming (requiring the programmer to reason on the correctness of a larger number of possible recovery states) in order to improve performance. Although FASEs in Atlas are easier to reason about, Atlas does not support synchronization primitives such as condition variables and lacks semantics for persistent lock-free programs.

3.2.3 Other Approaches. There are other published approaches that are mostly based on the Atlas model but add other restrictions [64, 72, 92, 153]. In general, they do not tolerate persistent updates outside FASEs and apply optimized logging mechanisms to improve performance (see §3.5). JUSTDO [72] and iDO [92] utilize a different solution to recover after failures, in which the code of a critical section is run to completion instead of having the persistent state rolled back, a method known as *recovery via resumption*. However, this method might impose further restrictions on the programming model. For instance, JUSTDO requires persistent caches for maximum efficiency, not allowing volatile data and caching of values in registers (which might require disabling some compiler optimizations such as register promotion) within a FASE. It also requires that locks are persistent. iDO relaxes some of these limitations and, for instance, does not require persistent locks or persistent caches. It also allows the use of volatile data inside FASEs. Both JUSTDO and iDO do not tolerate software failures, since re-executing the buggy code upon recovery will not restore consistency.

3.3 Transaction

Most PM systems provide FASEs in the form of transactions, including PMDK [131], Mnemosyne [143], NV-Heaps [30], NVL-C [40], Breeze [101], Espresso [152], and AutoPersist [138]. A transaction is essentially a block of code that satisfies the classical ACID properties: Atomicity, Consistency, Isolation, and Durability [56]. Not all PM systems supporting transactions provide thread-atomicity (isolation) though (see Table 2). In the bank account transfer example of Fig. 4d, `begin_tx()` (line 1) and `end_tx()` (line 4) delimit the transaction boundaries. Compared to the lock FASE of Fig. 4c, notice that transactions do not require a name (such as the object `l1`). They are also considered more elegant and robust compared to the other alternatives [30, 92]. For instance, transactions cannot overlap as in the example of Fig. 5a.

Semantically, a transaction can be treated as an acquisition and release of a single global lock [102]. In this regard, transactions can be seen as a more restricted case of lock FASEs [17]. Although most PM systems disallow updating persistent data outside transactions, a similar approach taken by

```
1  TX_BEGIN(pool) {
2      TX_ADD_DIRECT(&D_RW(FOO));
3      TX_ADD_DIRECT(&D_RW(BAR));
4      D_RW(FOO) = D_RO(FOO)-100;
5      D_RW(BAR) = D_RO(BAR)+100;
6  } TX_END
```

Fig. 6. Bank account transfer example using PMDK [131] transactional support. Library-level transactions require programmers to identify the memory region the transaction will affect (line 1), as well as marking the memory positions to be updated (lines 2–3).

Atlas would also make such updates possible [25]. Arguably, the main obstacle to the wide adoption of the transactional model concerns legacy code [17, 25, 92]. Converting lock-based code to its transactional equivalent is not always trivial, as some coding idioms such as condition variables and hand-over-hand locking are not immediately compatible with transactions [145]. Moreover, transactions are not yet widely available as a standard language feature, although a proposal for C++ does exist [71].

Even though some PM systems provide compiler extensions to support transactions [30, 143], in general this feature is offered at the library level, which usually adds some extra complexities to programmers. The code snippet in Fig. 6 shows the bank account example written with PMDK [131]. Programmers need to specify the persistent region the transaction will operate on (line 1, `pool`) and the persistent memory locations that will be affected (lines 2–3). Since PMDK uses fat pointers (see §4.3.2), the `D_RW` and `D_RO` macros are necessary when accessing persistent data.

3.4 General-purpose Constructions for Persistent Data Structures

So far, we have addressed approaches that aim at supporting FASEs in general-purpose blocks of code. In an alternative path, several works have considered the particular case where FASEs correspond to the operations of a specific data structure. Along this path, a large body of proposals has devised failure-atomic variants of popular data structures (e.g., B-trees [29, 90], hash-based indexing structures [109, 112]), designed with PM in mind. By focusing on a particular data structure, these proposals are not only able to redesign the underlying data structure to make it more amenable to PM, but also to employ custom failure-atomic mechanisms that exploit the intrinsic properties of such data structure. Consequently, this single-purpose approach has the potential to outperform general-purpose FASEs.

However, custom persistent data structures have a considerably restrictive applicability. Since designing a well-tuned failure-atomic data structure from scratch is far from trivial, the average programmer can only use custom persistent data structures (directly off the shelf) when they match the needs of his/her program. Furthermore, failure-atomic data structures typically cannot be composed inside complex operations (i.e., transactions) that atomically manipulate two or more data structures.

Recent works have also explored a middle ground between general-purpose FASEs and single-purpose custom persistent data structures, by proposing general-purpose frameworks which aid the transition between a wide class of volatile data structures to their corresponding PM-ready, failure-atomic variants. Among such proposals, we can distinguish two main categories. The first category presents objective easy-to-use recipes that a programmer can follow to manually transform a (volatile) data structure implementation into a persistent variant. In most cases, the resulting

data structure is guaranteed to have desirable efficiency, safety and, in some scenarios, liveness properties, while relieving the programmer from non-trivial decisions.

Izraelevitz et al. [73] were the first to propose a transformation recipe, defined by 5 main rules, that takes a non-blocking, volatile data structure, and creates a corresponding persistent version. Their general approach required, in a nutshell, issuing flush and fence instructions for every store (with release semantics) or compare-and-swap (CAS) operations in the volatile algorithm. Thus, the approach generates an excessive number of flushes and fences, which impair performance. As a result, new proposals have been considered to address the issue. David et al. [39] propose a methodology for transforming lock-free data structures to persistent variants without resorting to logging in the main data structure methods. RECIPE [89] provides a principled approach for making certain index data structures persistent. RECIPE provides *conversion actions* that need to be applied (manually) in case the code satisfies any of the three conditions presented. These conditions all require non-blocking reads and thus the approach is more applicable to non-blocking data structures. MOD [61] also proposes a recipe with the same goal, but specifically focused on purely functional data structures, i.e., whose functions preserve previous versions of the data structure when modified. MOD's recipe employs *functional shadowing*, which combines shadow paging (see §3.5) to minimize persist ordering overheads with optimizations from functional programming to reduce the overheads of shadow paging.

A second category takes a volatile data structure implementation, asks some lightweight code additions from the programmer, and then automatically creates a persistent version. The proposals in this category are commonly called *universal constructions* for persistent data structures. Such constructions require the programmer to augment the original (volatile) implementation of the data structure with high-level elements, e.g., annotating which data structure methods may update data, deriving the original class from a specific superclass, among others. The original methods are then automatically encapsulated into wrapper code that transparently injects the code that handles logging and recovery. When used with their simplest features, universal constructions require less manual instrumentation to the original implementation than, for instance, transaction FASEs (recall Fig. 6). Fig. 7 shows an example of turning the volatile vector data structure from C++ STL into its persistent counterpart using Pronto [100]. Two main steps are required: 1) create a wrapper class for the object (i.e., vector), extending Pronto's `PersistentObject` (line 2), and respective wrapper methods (i.e., `size`, `push_back`, and `pop_back`); 2) surround the update operations with the `op_begin` and `op_commit` commands (lines 14–16 and 20–22). These commands function as a FASE, as they guarantee the failure-atomicity of the operation. Read-only methods (i.e., `size`) do not require any annotation.

Also in this category, NVTraverse [47] provides automatic translation of a large class of lock-free data structures, which they named *traversal data structures*. Such structures are characterized by operations that first traverse a tree-like data structure and then perform modifications on nodes. For the so-defined traversal data structures, no flushes and fences are required in the traverse phase. Although the method itself is automatic, the authors first had to convert some common lock-free data structures (such as linked lists and hash tables) into their traversal counterparts in order to apply the transformation. ONLL [34] produces a lock-free persistent version, in which at most one fence instruction is required per operation. ONLL requires that the original data structure has been instrumented so that ONLL routines intercept calls to read-only and update methods of the data structure. CX-PUC [38] is the first persistent universal construction with bounded wait-free progress, requiring two fences per operation. Since the CX-PUC construction does not track which memory locations are modified (store interposing is not performed), a flush of the entire memory region in which the object was allocated is necessary. The method is therefore restricted to small objects. To mitigate the above limitation, the CX-PTM variant [38] combines

```
1  template <class T>
2  class PVect : PersistentObject {
3    vector<T, Alloc<T>> *vVect;
4  public :
5    PVect(string name):
6     PersistentObject(name) {
7      vVect = new vector<T, Alloc<T>>(alloc);
8    }
9
10   size_t size() const {
11     return vVect->size();
12   }
13   void push_back(T value) {
14     op_begin(value);
15     vVect->push_back(value);
16     op_commit();
17   }
18
19   void pop_back() {
20     op_begin();
21     vVect->pop_back();
22     op_commit();
23   }
24 };
```

Fig. 7. Using Pronto [100] to add durability to a vector data structure. The volatile vector is wrapped in a new class (PVect) and inherits from PersistentObject (line 2), which is part of Pronto's interface. Pronto's memory allocator is passed as reference (line 7), so that persistent memory can be appropriately managed. Update methods must be surrounded by the op_begin (lines 14 and 20) and op_commit commands (lines 16 and 22). This example is based on Fig. 4 of the original paper [100].

CX-PUC with transactions in order to achieve higher throughput at the expense of additional annotations for each data structure load and store.

While, in principle, the two above-mentioned categories for general-purpose persistent data structures are envisioned to incur only trivial programming effort, in practice the existing proposals impose additional constraints that partially contradict this initial goal. Firstly, some solutions impose further restrictions on the original implementation of data structure, which may entail non-trivial code adaptations to match such constraints. For instance, a common denominator is that externally-visible effects of any mutative operation should only depend on the current (persistent) state of the data structure and the arguments to the operation. Secondly, these approaches are typically less transparent when applied to concurrent data structures. For example, Pronto [100] requires the programmer to extend the internal code of each mutative operation by calling an auxiliary function to entangle the thread-atomicity order with the failure-atomicity order. Thirdly, most proposals in both categories do not generate composable persistent data structures, while those that do, require additional programming burden [38, 61].

3.5 Implementation Aspects

In this section we survey the techniques that have been proposed to implement the different types of FASEs presented in the previous section. Table 2 shows the main published works along with the type of FASE adopted and other features we discuss next. The table shows that a large amount of papers supports the transaction abstraction, despite the fact that this model has not been yet widely adopted for software development [25]. The proposed implementations can be characterized by four main design dimensions: the scheme they use to log persistent updates (§3.5.1); at which granularity they track and log updates (§3.5.2); whether they support thread-atomicity or not, and by what means (§3.5.3); and which persistency guarantees they offer (§3.5.4). Some of these features, such as logging scheme and tracking granularity, are completely hidden from programmers. Others, like thread-atomicity and persistency guarantees, affect both the programming model and implementation.

3.5.1 Logging scheme. As can be seen in Table 2, the majority of the existing PM systems implement failure-atomic updates using variations of well-known techniques largely employed in database

Table 2. Main published PM systems with: (i) type of FASE used; (ii) logging scheme; (iii) tracking granularity; (iv) thread-atomicity support; and (v) persistency guarantee.

System	FASE	logging scheme	t-gran.	thr-atom.	persistency [1]
FAMS [79, 118]	FA-epoch	redo	page	no	immediate
LibPM [98]	FA-epoch	undo	page	no	immediate
Atlas [25]	lock	undo	word	yes	buffered
SFR [53]	lock	undo	word	yes	both
JUSTDO [72]	lock	redo (resumption)	word	yes	immediate
iDO [92]	lock	redo (resumption)	word	yes	immediate
NVThreads [64]	lock	CoW + redo	page	yes	buffered
PMThreads [153]	lock	CoW + dual-copy	page	yes	buffered
PMDK [131]	transaction	undo	object	no	immediate
Breeze [101]	transaction	undo	object	no	immediate
NV-Heaps [30]	transaction	undo	object	yes	immediate
DCT [83]	transaction	undo	word	yes	immediate
Crafty [48]	transaction	undo	word	yes	buffered
Mnemosyne [143]	transaction	redo	word	yes	both
Pisces [58]	transaction	redo	object	yes	immediate
REWIND [27]	transaction	redo + undo	word	no	both
SoftWrAP [52]	transaction	CoW+redo	range	no	buffered
DudeTM [91]	transaction	CoW + redo	word	yes	buffered
cc-HTM [50]	transaction	CoW + redo	word	yes	both
NV-HTM [23]	transaction	CoW + redo	word	yes	immediate
NV-PhTM [10]	transaction	CoW + redo	word	yes	immediate
SPHT [22]	transaction	CoW + redo	word	yes	immediate
ArchTM [151]	transaction	CoW	object	yes	immediate
Romulus [37]	transaction	dual-copy	word	partial[2]	immediate
Kamino-Tx [99]	transaction	redo + dual-copy	object	yes	buffered
LSNVMM [65]	transaction	log-structured	word	no	immediate
EFLightPM [66]	transaction	redo/SP[3]	word	yes	immediate
TimeStone [85]	transaction	redo + op. logging	object	yes	immediate

[1] Most systems with buffered persistency provides some call to bring persistent state up to date but no details are presented about the implementation/performance costs. In those cases, we consider that only *buffered* persistency is supported.
[2] It provides another API for multi-threaded applications where all modifications are serialized and performed by a single writer thread.
[3] The choice is made according to the amount of data to be modified.

systems with traditional block-based storage: *Write-Ahead Logging* (WAL) and *shadow paging*. The original techniques had to be rethought in the context of PM due to the need to reduce the cost of flushes/fences, to limit *write amplification* (defined as the number of additional bytes written to PM for every byte of application data [108]) and to control write traffic.

Write-Ahead Logging. WAL is the key mechanism used by typical database systems to provide failure-atomicity and durability [104]. WAL requires the updates to PM to be recorded and persisted in a *log* before being written to the actual memory locations in PM. There are two main logging schemes employed by current PM systems: *undo* and *redo logging*. In undo logging, the old version

of the data is first saved in the log before the data is changed directly in PM; if there is a failure before a FASE completes, the undo log is used to roll back the changes. In redo logging, the new version of the data is written to the log; when a FASE is about to complete, it is necessary first to make sure the redo log is persisted and then apply the updates to PM, a procedure usually known as *replay*. If there is a failure before the redo log is persisted, it can be just discarded as the original values in PM have not been changed. If a failure occurs during replay, the redo log is just replayed upon recovery. Besides the old or new data version, some implementations of lock FASEs (e.g., Atlas [25] and SFR [53]) also require logging synchronization operations (acquire and release) for correct recovery.

Fig. 8 shows a simplified version of both undo[8] and redo logging applied to the previous bank transfer example using the x86 persistency model. It assumes a persistent log with an append operation and a status to control whether it must be rolled back (undo logging) or forward (redo logging) upon recovery. Log replay is triggered after recovery if the status of the log is `ACTIVE` (it is initially `INACTIVE`). The main characteristic of undo logging (Fig. 8a) is that the new values are updated directly in PM (lines 8 and 13). Therefore, before each update, the old value must be inserted into the log (lines 5 and 10) and the corresponding entries persisted (lines 6–7 and 11–12). Note that the necessary flushes and barrier do not need to be placed right after the updates and are postponed (lines 15–17) until before changing the log status to `INACTIVE` (lines 19–21). If a failure occurs at any point before that, the changes will be rolled back upon recovery using the log entries.

Redo logging (Fig. 8b) requires the new values to be first inserted into the log (lines 1–2) and the corresponding log entries persisted (lines 4–6). If a failure occurs up to this point, nothing will happen after recovery because the log status is still `INACTIVE`. After its status changes to `ACTIVE` (lines 8–10), the redo log is considered persisted and can be replayed in case there is a failure before the FASE is completed (up until line 21). The replay consists of traversing the log (not shown explicitly) and persisting the updates (lines 12–17). The log is then deactivated (lines 19–21) and can be reclaimed.

Comparing the costs for undo and redo logging, it can be observed that both require two flushes (`CLWB`) for each memory location written (one for the data itself and another for the corresponding log entry) and two more for updating the log status. Therefore, the total cost for flushes is $2W + 2$, where W is the number of writes. The number of `SFENCE` barriers issued by each scheme is different, however. Undo logging requires one for each log entry [9], one for the new values, and two for the log status: a total of $W + 3$. Redo logging requires only one for all log entries, one for the new values, and two for the log status: a total of 4 barriers. The disadvantage of redo logging is that, since newly written values are only recorded in the log, read operations first need to check if the data is already present in the log, which might add overhead.

Table 2 shows that the basic redo and undo schemes have been employed in implementations of all different FASEs, from FA-epochs to transactions. The logging approach was also one of the differences between the two seminal PM systems: while Mnemosyne [143] used redo logging, NV-Heaps [30] opted for undo logging. Both logging approaches have been extensively studied in the PM literature since then [80, 91, 116, 144, 157, 158]. Overall, redo logging tends to perform better when the number of updates is relatively large, whereas undo logging performs best for read-dominated workloads [144]. Moreover, for systems in which persist barriers are expensive (such as Intel Optane DC), redo logging is more likely to present better results because it only requires $O(1)$ barriers compared to undo logging's $O(W)$ [157].

[8]We are assuming in this example that the store locations are not known a priori and therefore a barrier is needed for each log operation (append). Otherwise, one single persist barrier after all log entries are appended would be enough.
[9]Recall that only one barrier is needed if the undo logging approach assumes that the store locations are already known.

```
1  log.status = ACTIVE
2  CLWB  log.status
3  SFENCE
4
5  log.append(&FOO, FOO)
6  CLWB  (last entry)
7  SFENCE
8  FOO = FOO-100
9
10 log.append(&BAR, BAR)
11 CLWB  (last entry)
12 SFENCE
13 BAR = BAR+100
14
15 CLWB  FOO
16 CLWB  BAR
17 SFENCE
18
19 log.status = INACTIVE
20 CLWB  log.status
21 SFENCE
```

(a) Undo logging

```
log.append(&FOO, FOO-100)
log.append(&BAR, BAR+100)

CLWB  (FOO log entry)
CLWB  (BAR log entry)
SFENCE

log.status = ACTIVE
CLWB  log.status
SFENCE

FOO = FOO - 100 // log writeback
BAR = BAR + 100 // log writeback

CLWB  FOO
CLWB  BAR
SFENCE

log.status = INACTIVE
CLWB  log.status
SFENCE
```

(b) Redo logging

Fig. 8. Bank transfer example with Write-Ahead Logging (WAL) for x86: (a) with undo logging, the old values are first inserted into the log (lines 5 and 10) and persisted (lines 6–7 and 11–12) before the data is updated in PM (lines 8 and 13); (b) with redo logging, the new values are first inserted into the log (lines 1 and 2) and persisted (lines 4–6), before being replayed (lines 12–17) at the end. The log *status* controls whether it needs to be rolled back (undo logging) or forward (redo logging) after recovery.

Flushes and fences have been identified as the major source of overhead in current x86 machines [13, 33, 37, 61, 83, 108]. Therefore, optimizing their use is crucial to achieve optimal performance. For instance, one could try to reduce the number of SFENCEs required in the bank transfer example of Fig. 8b by adding a *commit marker* as the last entry in the redo log. The recovery code will check if the commit marker is present in order to decide whether the log should be replayed. Therefore, instead of using an SFENCE after the status of the log is changed to ACTIVE (line 10) and INACTIVE (line 21), a single SFENCE after the commit marker is flushed should suffice, reducing the total cost from 4 to 3. Further optimizations to reduce this number to 2 are possible using torn bits or checksums [83, 143].

Cohen et al. [33] explored modern persistent memory coherency protocols and devised logging algorithms to reduce the number of fences. The algorithms are based on the observation that the order that stores are made to the same cache line corresponds to the order they are persisted, which they named *Persistent Cache Store Order* (PCSO). According to PCSO, if the last store has persisted, then all preceding stores to the same cache line have also been persisted. The same authors later proposed a specific undo logging technique based on PCSO called *In-Cache-Line Logging* [31] (inCLL). It can be used for certain data structures whose modifications fit a single cache line, but cannot be generalized to implement all types of FASEs.

Resumption-based redo logging is a technique that only applies to FASEs that cannot abort (FA-epoch and lock). As such, upon recovery, instead of replaying a log, one could simply resume the FASEs and execute them to completion. In the literature, there are two systems that use this idea

in the context of lock FASEs: JUSTDO [72] and iDO [92]. The main drawback of resumption-based logging is that they cannot tolerate software failures, since re-executing the buggy code will not restore consistency upon recovery.

Regardless of the chosen logging scheme (undo or redo), WAL adds both space and time overhead. Extra space is required to hold the log entries and is proportional to the number of updates performed in a FASE. WAL requires two writes to PM for every update in a FASE: one to insert the old (undo) or new (redo) value into the log, and another to update the data in PM. This *write-twice* behavior originates a number of drawbacks [37, 65, 72, 153]. First, it potentially wears out the PM device faster. Second, it increases the write amplification. In the example of Fig. 8, if at the least the address and the value are written to the log, the write amplification is 200%. Third, it increases write bandwidth, which is critical in current PM technology. Finally, it requires more barriers, increasing the FASE latency.

Shadow Paging and Copy-on-Write. The WAL mechanism described earlier provides *in-place* updates, that is, the new value is written directly on top of the original value in PM. *Shadow Paging* (SP) [55], the other popular mechanism to provide atomicity and durability in database systems, allows out-of-place updates without incurring the write-twice problem of WAL. The general idea of SP is that, instead of performing the updates directly on the original objects, private copies are first created so that persistent updates can be applied to them without disturbing the original objects. Because the objects are local, they can be modified without worrying about the order of persists. When the FASE is about to finish, the original objects are replaced with the updated copies atomically. In the common case, only one persist barrier (e.g., `SFENCE`) is needed for failure-atomic updates using the SP approach [61, 66]. If a failure occurs before the FASE is over, nothing must be done during recovery other than reclaiming the memory space allocated for the copies.

The Copy-on-Write (CoW) technique is sometimes used as a synonym for SP, but here we adopt a more strict meaning, aligned to what is generally understood in the context of OSs [19]. In this context, CoW is an efficient technique that enables objects (pages) to be shared as long as they are not written to. Once an object is modified, a private copy is created to prevent the changes from becoming visible to all objects. When implemented by the OS' virtual memory manager, CoW replaces the original page with its shadow copy in the same position in the process' virtual address space. This transparently ensures that any pointers to objects residing on the original page remain valid (however, now pointing to the equivalent copies on the shadow page). Therefore, most implementations of SP rely on the OS-based CoW technique to implement out-of-place updates.

Since SP performs out-of-place updates, it solves the write-twice problem present in WAL mechanisms. However, since it is based on the CoW technique, it creates a new issue: copying overhead and consequent write amplification [35, 37, 61, 68]. Most of the existing PM systems integrate CoW with either redo logging or the dual-copy technique (described later).

Although the vast majority of proposals that resort to SP employ OS-based CoW, other alternatives are possible. For instance, in MOD [61], shadow copies are explicitly created (i.e., allocated and filled in) by the application code. This approach is typically employed when the original program already implements out-of-place updates via object copying (as in purely functional data structures [61]). Such programs can avoid the OS-based CoW overheads and exploit shadow copying at finer granularity copies (e.g., object-granularity) for improved efficiency. However, it is harder to apply such technique for legacy data structures as the address of an object changes after its shadow copy is created upon new updates.

Operation Logging. Another alternative to WAL is to intercept and log calls to operations (e.g., methods in a data structure), similarly to operation logging in database systems [104]. Hence, only the invocations of mutative operations and the corresponding arguments need to be logged, rather

than the individual updates to PM that each operation performed (as in WAL). This lightweight form of logging can significantly reduce the persistence overheads when compared to update-based logging. Conversely, recovery is possible by re-executing the logged operations whose outcome was not persisted before a failure. Operation logging typically requires that the externally-visible effects of any mutative operation should only depend on the current (persistent) persistent state and the arguments to the operation. This approach is specially tailored to persistent data structures [38, 61, 100] (where the above restriction is largely met), but has also been employed to support transactions [85].

Hybrid strategies. Modern approaches for implementing failure-atomic updates combine different strategies. Fig. 9 shows two of the most recent techniques. The CoW + redo logging approach, displayed in Fig. 9a, first creates a *working copy* of the persistent data ①. Usually, the OS CoW support is used [23, 91] and the working copy is preferably created in DRAM, although that is not required. Then the execution of the FASE is broken into three fully asynchronous steps as initially proposed by DudeTM [91]. In the first step, called *perform* ②, the modifications are performed in-place in the working copy and also kept locally in a volatile redo log. The working copy prevents the changes made by threads from immediately affecting the persistent state. The second step, *persist* ③, can be realized by a helper thread in background and consists of persisting the redo logs produced by each FASE in a separate redo log kept in PM. After this step, the modifications are persisted and will be replayed after recovery in case a failure occurs. Finally, in the third step, the redo logs are replayed in the persistent heap ④ by one or more replayer threads. This step performs log truncation and is important to prevent the persistent redo log from becoming full, which could potentially stall the execution of FASEs due to the lack of free log space.

The big advantage of the CoW + redo logging approach is its flexibility. For instance, if we consider transaction FASEs, the perform step could be built with any out-of-the-box Software Transactional Memory (STM) system, or even Hardware Transactional Memory (HTM), as explored by cc-HTM [50] and NV-HTM [23] in the context of Intel TSX [36]. It also allows removing costly fence instructions from the critical path, as the persist step can be performed asynchronously by another thread. If immediate persistency is required, the persist stage can be merged into the perform stage as done by NV-HTM [23], with the drawback of adding the fence costs in the critical path again. Moreover, decoupling the execution of FASEs also creates further optimization opportunities. For instance, the three decoupled steps can be overlapped (increasing concurrency) and write combination strategies can be developed to reduce the number of writes to PM (improving write endurance) and decrease write amplification [23, 91]. This approach has been used in the PM literature by both transaction (SoftWrap [52], DudeTM [91], cc-HTM [50], NV-HTM [23], NV-PhTM [10], SPHT [22]) and lock (NVThreads [64]) FASEs.

Another popular approach used by some of the most recent works is based on the *dual-copy* technique, as illustrated in Fig. 9b. In this approach, two versions of the persistent heap are kept all the time, namely *working copy* and *consistent copy*. The working copy is where the FASEs execute in-place modifications ①. After a FASE is finished and the working copy state is persisted, the modifications are propagated to the consistent copy ② to bring it up-to-date. Notice that the consistent copy is never accessed directly by a FASE and serves solely as a backup of the previous version of the working copy. A failure might happen at two moments: 1) during modification of the working copy; 2) after the FASE is complete but before all modifications have been copied over to the consistent copy. In the former case, the recovery procedure must copy the contents of the consistent copy back to the working copy in order to undo any incomplete modifications. In the latter case, the contents of the working copy will be propagated to the consistent copy during the recovery procedure. Although the dual-copy technique limits the write amplification to 100%, it

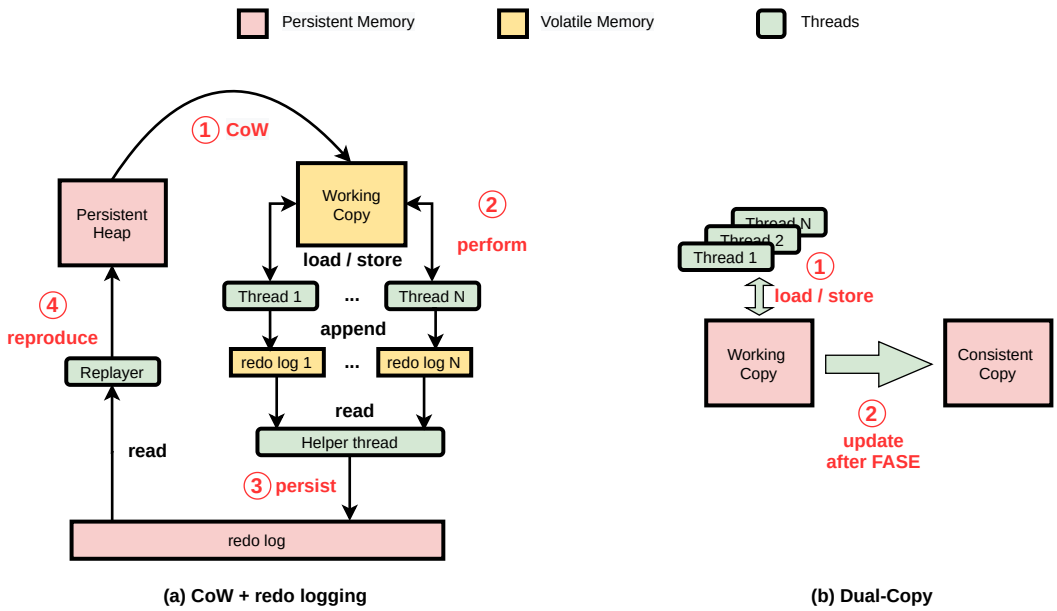


Fig. 9. Two modern approaches to implement failure-atomic updates: (a) CoW is used to create a working copy of the persistent heap in DRAM – updates are first appended to redo logs and then persisted and reproduced in the persistent heap by background threads; (b) a twin copy of the data is kept in PM and only one version is modified – after a FASE is completed, the updates are propagated to the consistent copy which functions as an undo log in case a failure happens when modifying the working copy.

should be noticed that it also reduces the PM capacity in half, thus doubling the memory cost of the system.

The dominant source of overhead of the dual-copy scheme is the copy operation necessary to update the consistent copy. Recent systems based on this scheme, such as Romulus [37], Kamino-Tx [99] and PMThreads [153], adopt different strategies to cope with this problem. Romulus [37] uses a special type of volatile redo log, which maintains the addresses and possibly the size of the modifications. Consequently, only the modified data in the working copy is copied back to the consistent copy. The copy operation, however, is still performed in the critical path (as part of the transaction commit). In order to avoid that cost, Kamino-Tx [99] first constructs a persistent redo log for each FASE during runtime. This redo log is then used to copy the modifications asynchronously after a FASE is finished, effectively removing the cost from the critical path. In case a failure happens during the copy, the redo log is replayed at recovery time to reconstruct the consistent copy. The main drawback of Kamino-Tx is that it increases the write amplification compared to Romulus due to the extra persistent space required for the redo logs.

PMThreads [153] improves on both Romulus and Kamino-TX by coupling CoW with the dual-copy scheme. There are three versions of each persistent heap: the usual working copy and persistent copy, plus a shadow copy in DRAM. FASEs perform in-place updates in the shadow copy, which is then used to update the persistent working copy periodically, at quiescent points. When the working copy is updated successfully, the working and persistent copies atomically switch roles. Therefore, the only copy operation necessary is the one to propagate the updates from the shadow copy in DRAM to the working copy in PM. In order to perform a failure-atomic switch of roles,

PMThreads requires a version number to be attached to persistent data. To reduce the cost of adding such metadata, modifications are tracked at the page granularity instead of individual words. Overall, PMThreads reduces write amplification considerably because each store in the shadow copy results in at most one store in PM. It does require storing a persistent version number for each page however, which slightly increases write amplification.

Other hybrid approaches have been proposed with the goal of reducing logging costs. For instance, EFLightPM [66] can use either redo logging or shadow paging according to the number of modifications performed in a FASE. If the majority of the persistent region is updated, a CoW-based technique is used, whereas a redo logging scheme is employed for minor modifications. However, EFLightPM requires programmers to manually specify the preferred type for each FASE in the program. TimeStone [85] uses a multi-layered hybrid logging scheme. A volatile log is kept at the first layer (`TLog`), where a copy of the object is updated. When a FASE finishes, TimeStone persists the computation performed in a second layer (`OLog`) in PM using operation logging, allowing TimeStone to achieve immediate persistency with low costs. When the utilization of `OLog` reaches a threshold, it triggers the reclamation of `TLog`, which is finally persisted in the third level of the multi-layered log (`CLog`). At this point, `OLog` entries can also be reclaimed. The entries in `CLog` are replayed periodically at quiescent points and reclaimed. During recovery, TimeStone first replays the entries stored in `Clog`, followed by the operations in `OLog`. The multi-layered approach used by TimeStone can reduce write amplification considerably.

Log-structured approach. LSNVMM [65] reduces logging costs in a totally different way: the PM heap itself is organized in the form of logs. Data is updated by appending it to the end of the log, reducing space fragmentation and avoiding redo/undo logging overhead. Application addresses are translated to log offsets by an address mapping, which must be adjusted every time a new data is updated. Thus, the behavior of the mapping mechanism is crucial for performance, since it is invoked by every load or store instruction in the application. Given this requirement, the address mapping employs a highly-optimized tree-based data structure and a node cache to reduce the cost of traversing the tree. The log entries store enough metadata so that the data structure used to perform the translation can be rebuilt after a failure. One of the main features of the log-structured approach compared to other logging techniques is that it can improve bandwidth utilization and reduce write wear, as data is written sequentially (appended) at the end of the log. Despite all optimizations, LSNVMM still introduces overhead due to the additional level of indirection required by the mapping scheme. Furthermore, a garbage collector is required to recycle dead log space by moving sparse live data from several regions to a new one in a compact way.

Hardware approaches. A lot of recent works have investigated how to improve the efficiency of software written to PM through modifications in the processor, caches and/or memory controllers. Although a full discussion of the particularities of these works is out of the scope of this survey, we briefly present here the main approaches and how they could potentially alleviate the problem of the high costs of flushes and fence instructions and help software in implementing efficient failure-atomic update strategies.

Instead of waiting for the fence instruction to complete, Shin et al. [136] proposed *Speculative Persistence* (SP), a technique that allows the execution to continue speculatively after the fence. The speculative stores are buffered and not allowed to propagate to PM until the fence instruction completes, at which point the speculation is over. Other works focused on relaxing the ordering constraints to improve efficiency. Epoch barriers have been proposed in BPFS [35], allowing persists within an epoch to be reordered. Further optimizations for epoch barriers have been investigated in LB++ [75], allowing flushes to be initiated by the hardware as soon as possible in order to remove the long latency of such operations from the critical path. Decoupling persist ordering from

volatile execution have been investigated in DPO [84] and HOPS [108]. Hardware persist buffers are added so that the long latency of persists can be removed from the critical execution path. StrandWeaver [54] relaxes the ordering constraints even further by using the concept of a *strand*, a logically independent sequence of operations within a thread. The main difference between a strand and an epoch is that stores to PM from different strands are unordered (this is not the case with epoch) and can be persisted concurrently, unless a persist barrier is used.

Another optimization that has been extensively studied is the addition of *hardware-assisted logging*. In this approach, logging is done transparently by the hardware, which can efficiently overlap log writing and transaction execution. All proposed systems invariably use transactions as the main building block for failure-atomicity. The approaches differ in the logging strategy and the complexity of the hardware. Hardware redo logging is implemented in WrAP [49], LOC [96], ReDU [74], HOOP [20], and the approach designed by Doshi et al. [41]. Undo logging strategies have been incorporated in ATOM [77] and Proteus [135], whereas FWB [116] and MorLog [148] explores undo+redo strategies. Hardware shadow paging at the cache-line granularity has been investigated in SSP [115]. Some strategies manage to avoid logging and leverage the multiple versions of data already present in the memory hierarchy. Kiln [160] uses a persistent last-level cache to buffer the updates, whereas LAD [60] uses the memory controller's persistent queue.

Hardware-assisted logging proposals usually do not deal with concurrency control, requiring software to devise a pessimist or optimistic strategy. Solutions providing ACID semantics have been proposed in PTM [147], PHTM [8], PHyTM [7], DHTM [76], and by Giles et al. [51]. Most of these approaches extend Intel Reduced Transactional Memory (RTM) [36] with durability guarantees.

3.5.2 Tracking granularity. The granularity in which data is tracked has a big effect on the performance of the underlying system, as increasing the granularity can reduce the logging costs. Coarser-grained tracking granularities can reduce the overhead of maintaining logs but can also increase write amplification if used carelessly. For example, assume a FASE writes to 100 different memory addresses, all within the boundary of a single 4KiB page. A page-level tracking granularity would only require a single log entry but need an extra 4KiB to store the undo/redo data. Some page-based PM systems such as NVThreads [64] only store the difference between the original and the modified version of the page in the log.

FA-epochs typically track modifications at the page granularity [79, 98]. The main technique used by systems tracking at this granularity is to register a handle with the OS to intercept page protection faults. Initially, all pages are write-protected. When the application accesses such pages, a fault will be generated and caught by the underlying system to keep track of modified pages. In the case of FA-epoch, only the modified pages are written back to persistent media when a synchronization point is reached. If a FASE is composed of only small modifications (as with the example of the bank account transfer), and synchronization points are reached frequently, a significant overhead might be introduced since the system is paying the price of a full 4KiB write-back because only a few bytes were changed [79].

3.5.3 Thread-atomicity. Some early systems proposed in the 1990's opted to factor out concurrency control from the design of transactional FASEs. Systems such as RVM [130] and Rio Vista [95] argued that concurrency control is best handled by *"higher levels of software"* and that *"adopting any concurrency control scheme would penalize the majority of applications, which are single-threaded and do not need locking"*. Some recent PM systems also chose not to incorporate any isolation mechanism into the design of FASEs in favor of greater flexibility [52, 65, 98, 118]. The main argument is that integrating isolation into the design would force application developers to adopt a single concurrency control model. However, some systems do acknowledge that not providing any native isolation support might have negative impacts in terms of performance and programmability [118].

FASEs that are based on locks, such as Atlas [25], inherit the thread-atomicity guarantees already provided by such synchronization method. Other approaches, such as cc-HTM [50], NV-HTM [23] and Crafty [48], leverage the isolation features supplied by the transactional memory hardware to implement transaction-based FASEs. Systems based on STM, such as Mnemosyne [143], also leverage the isolation level provided by the STM library. Solutions that delegate isolation to other layers of the software stack usually suggest the use of mutexes [37, 65], but it is not clear how exactly that can be done. While a single global mutex could be easily adopted, it would serialize the entire execution. Using more fine-grained locks tends to get much harder quickly [63].

Systems that support concurrent FASEs need to deliver a high parallelism level while, at the same time, provide correct execution. The vast majority of log-based systems implement per-thread logs in order to increase the concurrency level. In this case, the implementation must be careful so that enough information is stored to allow the logs to be replayed in the correct order in case of failures. For instance, systems that employ redo logs need to store a timestamp for each persisted log record to control the replay order. Such timestamps can be generated by incrementing a shared global counter, but this would create scalability issues as identified by DudeTM [91]. Therefore, systems such as NV-HTM [23] generate timestamps through a physical clock (e.g., using the Intel `rdtsc` instruction) to avoid contention.

Another strategy to increase the parallelism level in systems with transaction FASEs is to relax the isolation level. For instance, Pisces [58] allows more transactions to commit and persist simultaneously by adopting snapshot isolation (SI) instead of the more popular serializability or linearizability isolation levels. Such design tends to favor read-dominated workloads, but does not work for all types of applications, as SI suffers from the write-skew anomaly [43]. Realizing that no single isolation level suits all application types, TimeStone [85] supports multiple levels (linearizability, serializability, and snapshot isolation), allowing programmers to specify which mechanism is more appropriate according to the characteristics of the application.

3.5.4 Persistency guarantees. Some systems distinguish between *immediate* and *buffered* persistency. The former ensures that, when a subsequent FASE executes, all the previous FASEs are durable, i.e., in case of recovery from a crash their changes will be present in PM. Buffered persistency relaxes this guarantee, as it only ensures that previous FASEs will eventually be persisted.

In PM systems like DudeTM [91] and Crafty [48], buffered persistency allows amortizing the cost of memory flushes and fences during the commit of transactions. While they ensure that subsequent transactions observe fresh state, such state may still be either volatile or unstable, i.e., it is still not prepared for crashes and might be rolled back on recovery. Some PM systems discuss the performance impact of implementing immediate persistency, as in most of them a logically sequential step during commit is required. For instance, cc-HTM [50] proposes the externalization of transaction barriers to programmers in order to achieve immediate persistency. These artifacts provide strong optimization alternatives, but also expose programmers to a whole new set of complex problems such as where exactly the barriers should be placed in the code.

On the other hand, immediate persistency systems provide a much more natural semantics to programmers, but their performance is usually worse due to the strong guarantees they provide. For instance, NV-HTM [23] requires a wait phase during commit in which the committing transaction has to wait for all the preceding transaction they may depend upon. Given NV-HTM reliance on HTM, it does not track transaction reads explicitly, hence all the preceding transactions are taken as possible conflicting transaction. As a result, every single transaction must wait for all preceding transactions (both conflicting and non-conflict transactions), i.e., while preceding transactions are not prepared for a crash, the current transaction cannot return to the application.

3.5.5 Intel Optane DC PM. With the introduction of the first commercial PM in the market, Intel Optane DC PM [142], a few papers have conducted a performance characterization and offered some guidelines [120, 155]. These guidelines have been used in recent papers to speed up persistent transactional systems that are built on top of Intel Optane. ArchTM [151] is designed around the guideline of avoiding small writes (less than 256 bytes) and coalescing them in order to make use of the higher bandwidth provided by the sequential access pattern. The commit protocol in SPHT [22] is highly optimized to make the most out of the limited write bandwidth of Intel Optane. Furthermore, SPHT provides NUMA-aware log replay so as to reduce the previously-studied shortcomings of NUMA on Intel Optane DC PM [120, 155].

4 ACCESS AND HEAP MANAGEMENT

Transparent interfaces to non-volatile media have appeared on early systems such as Multics [11] and the experimental 801 Minicomputer Storage system [26]. In those systems, both volatile and persistent data (called *files*) were part of the virtual memory address space and thus uniformly accessed. If persistent data was requested, the OS would automatically take care of performing the required I/O operations (based on demand paging) to move the data between the internal levels of the storage hierarchy.

The easier programmability provided by the single-level storage in Multics was offset by the lack of control required to implement failure-atomic updates and was later considered a major flaw in its design [57]. Although transparent access to volatile and persistent data was still considered for database management systems such as ObjectStore [86], and is even used currently in DB2 for IBM i [69], it has not made its way into the majority of existing OSs. Instead, *mmap*, a POSIX system call, is used to map files into the process' virtual address space. Other early approaches to abstract access to persistent data consisted of user-level libraries, such as the Recoverable Virtual Memory (RMV) [130] and the Stasis [133] systems. Those proposals were still designed with block-based storage devices in mind.

With the introduction of PM, these old approaches needed to be rethought and new alternatives have been suggested. This section presents the programming interface provided by modern PM systems to access (§4.1) and manage (§4.2) persistent data. It also discusses implementation strategies and performance tradeoffs (§4.3).

4.1 Accessing Persistent Data

A persistent programming model has to specify how persistent data is exposed to programmers. The works in the PM literature can be classified into three major categories: **transparent**, **transitive**, and **selective** persistence. Fig. 10 shows a simplified example of adding an element to a persistent linked list using each of the three different schemes. Notice that failure-atomic updates are not addressed in this example.

Transparent persistence is the simplest or most straightforward approach from the programmer's perspective, as no markings in the code must be added to track persistent data (Fig. 10a). Such model is provided by Whole-System Persistence [110] (WSP), Transparent Hybrid NVM [127] (ThyNVM) and PiCL [114], which differ in how they implement transparent persistence. WSP supports only PM devices (no DRAM) and relies on the residual energy provided by the power supply to flush all data to PM on failures, whereas ThyNVM and PiCL use a periodic checkpointing mechanism. All three approaches tolerate power failures, but cannot deal with software crashes (consider a bug in the memory allocator used in Fig. 10a, for instance). Furthermore, because ThyNVM and PiCL rely on checkpointing, they do not provide immediate persistency, as the state after recovery, although consistent, might not reflect the most recent updates performed before the failure.

```
1  /* initialized
2   previously */
3  lst_t *head = ...;
4
5  lst_t *new_node =
6     malloc();
7  new_node->next = head;
8  head = new_node;
```

(a) Transparent Persistence

```
rid = pROpen("mylist");
list_t *head =
   pGetRoot(rid);

list_t *new_node =
   malloc();
new_node->next = head;
head = new_node;
```

(b) Transitive Persistence

```
rid = pROpen("mylist");
list_t *head =
   pGetRoot(rid);

list_t *new_node =
   pMalloc();
new_node->next = head;
head = new_node;
```

(c) Selective Persistence

Fig. 10. Adding an element to a persistent linked list using three different schemes: (a) transparent, (b) transitive, and (c) selective persistence. Transparent persistence does not require any special markings, but transitive and selective persistence relies on programmers to identify root pointers. Selective persistence further demands that programmers make sure the data being added to the list is persistent (using a PM-aware allocator, in this example).

In transitive and selective persistence, persistent data is exposed to applications in the form of *persistent regions* (also called persistent containers or pools). A persistent region is a contiguous range in the virtual address space of the application that is mapped to the PM device. *Persistent heap* is used to refer to the contents of the region. Access to the persistent heap is realized by means of *root pointers*: entry points from which persistent data is accessible. Accessing persistent data thus requires two initial steps: 1) create/open a persistent region (line 1 of Fig. 10b/c); 2) retrieve the root pointer (line 3 of Fig. 10b/c). The main difference between transitive and selective persistence is that a system providing transitive persistence *implicitly* identifies persistent data as all data that is reachable from the root pointer, whereas selective persistence requires programmers to *explicitly* select persistent data.

Transitive persistence is one of the main implementation techniques in which the lifetime of data is determined by its reachability. With transitive persistence, the identification of persistent data is performed automatically by computing the transitive closure of all data reachable from a root pointer. In the example of Fig. 10b, the assignment in line 8 automatically makes the new node persistent. Transitive persistence, also known as persistence by reachability, first appeared in the context of *orthogonally persistent systems* and the integration of programming languages with database systems in the 1980's [3, 4, 6]. The goal of orthogonally persistent systems is to treat data in a uniform manner, independent of its longevity, size, or type.

The availability of new PM devices renewed the interest in transitive persistence and two recent papers proposed its use: AutoPersist [137, 138] and JaphaVM [121]. Both systems use different methods for identifying root pointers. In AutoPersist, such roots are identified by labeling static fields with a new annotation added to the Java language (`@durable_root`). JaphaVM implicitly assumes that static variables of all classes and the stack of all active threads are roots of persistence. SoftPM [59] and LibPM [98] also implement transitive persistence but targets low-level imperative languages, requiring programmers to identify root pointers similarly to the example of Fig. 10b. Although an elegant model, using transitivity may cause more data to be persisted than necessary and may incur high runtime overheads [17, 30].

The large majority of the published systems assume selective persistence, requiring programmers to explicitly specify which data should be persisted. In the example of Fig. 10c, the memory for the new node must be allocated from the persistent heap (line 5) before being attached to the linked list. If the standard allocator is called instead, the pointer to the next element will be invalid

after recovery, as volatile data is lost in case of failures. Notice that this scheme is naturally more error-prone. Indeed, Marathe et al. [97] report their experience in adapting Memcached to PM and concluded that the effort was "surprisingly complex". Among the difficulties, they highlight that deciding what to persist is hard. Ren et al. [126] reached similar conclusions after a field study with 30 programmers.

Several systems allow programmers to limit the scope of pointers and provide some level of safety. For instance, NV-Heaps [30] uses operator overloading to implement pointer types. There are two main types: 1) pointers from volatile to persistent heap (V-P); 2) pointers from persistent to persistent heap (P-P). Pointers from persistent heap to volatile data (P-V) are disallowed. NV-Heaps further differentiates intra-heap from inter-heap P-P pointers. Intra-heap P-P pointers are allowed, but NV-Heaps forbids inter-heap P-P pointers because the pointers become unsafe if the heap that contains the data is not available (it also complicates other internal mechanisms such as garbage collection). Returning to the example of the allocation in line 5 of Fig. 10c, the bug resulting from calling the standard allocator could be avoided by using a V-P pointer, as the assignment would cause a runtime error. NVL-C [40] provides something similar to NV-Heaps but uses type qualifiers, allowing the error to be caught at compile time. Mnemosyne [143] also provides basic support for detecting P-V through the `persistent` type qualifier.

Espresso [152], a Java-based persistent system, takes a different approach and lets programmers choose among three distinct safety levels when dealing with volatile and persistent heaps. The highest safety level, *type-based safety*, is similar to NV-Heaps. The lowest level, *user-guaranteed safety*, allows P-V references and the burden on checking for errors is left to programmers. On the other hand, it is the level with the highest performance. *Zeroing safety*, the middle-ground level, will automatically nullify all out references upon reloading, allowing applications to use null-checks but slowing down the recovery phase since the whole heap needs to be traversed. In a similar fashion, Cohen et al. [32] proposed the *transient* keyword to annotate fields of persistent objects, which are then automatically zeroed by the runtime system during recovery. Systems with low-level interfaces to PM, such as Intel's PMDK [131] and Breeze [101], provides some level of type-safety by requiring the use of special macros.

4.2 Persistent Heap Management

Conventional memory allocators are usually optimized to increase allocation speed and reduce heap fragmentation. The design of persistent allocators is complicated by extra requirements imposed by the very nature of persistent data. Internally, a failure might corrupt the allocator metadata which, in turn, could potentially produce side-effects such as incorrect reallocations. Externally, the interface between the allocator and applications has to be carefully planned in order to avoid *persistent memory leaks*. For instance, consider the case in which the allocator correctly allocates memory but a failure occurs before the application had the chance of storing the returned address in a persistent location.

Whereas internal consistency is more easily achieved, dealing with external consistency is more problematic since it requires a consensus between the allocator and the underlying PM system [12]. There are several different ways in which this consensus can be obtained in the literature. Fig. 11 presents the two most common methods used by current systems for inserting a new element into a persistent linked list. In Fig. 11a, the allocation (line 2), initialization of the element (line 4) and the insertion itself (lines 6 and 7) must be inside a FASE (in this example, a transaction). The approach presented in Fig. 11b only requires the insertion to be wrapped in a FASE (lines 5-8) and more closely resembles the equivalent code in a volatile setting.

The approach illustrated in Fig. 11a is taken by several systems, such as Mnemosyne [143], PMDK [131], HEAPO [67], Breeze [101], and EFLightPM [66]. There are two main problems with

```
1  transaction {
2     lst_t *new_node = pMalloc();
3
4     /* initialize new_node data */
5
6     new_node->next = head;
7     head = new_node;
8  }
```

(a)

```
lst_t *new_node = pMalloc();

/* initialize new_node data */

transaction {
   new_node->next = head;
   head = new_node;
}
```

(b)

Fig. 11. Code snippet for adding a newly allocated element into a persistent linked-list: (a) allocation, initialization, and insertion must be performed inside a FASE (transaction); (b) only the insertion is required to be in a FASE.

this method though. First, the interface is not what most programmers would expect since it deviates from the traditional allocation interfaces [12]. It requires programmers to use FASEs every time persistent memory must be (de)allocated. For instance, if the allocation in line 2 of Fig. 11a is mistakenly placed before the transaction, the code will be susceptible to a persistent memory leak. Second, it unnecessarily increases the size of FASEs, which could also impair performance.

Systems that provide a more conventional interface to persistent memory allocation, as illustrated in Fig. 11b, include NV-Heaps [30], NVL-C [40], Makalu [12], and Ralloc [21]. These systems rely on some form of Garbage Collection (GC) to avoid persistent leakage. NV-Heaps and NVL-C rely on a reference counting GC and provide weak pointers in the language in order to break cycles. A cycle occurs when two objects refer to each other, preventing memory reclamation. Notice that this increases programming complexity, as programmers must decide where to use strong or weak pointers. Makalu and Ralloc use a more conservative tracing GC. The way persistent heaps are structured makes tracing GC a natural fit, as root pointers serve as starting points for the tracing algorithm: any block not reachable from the root pointer is deallocated by the GC algorithm. A particularity of Makalu is that GC only occurs offline, upon recovery. This has the effect of improving online allocation performance but requires explicit deallocations.

There are still other approaches to persistent memory management in the literature. The interface provided by `nvm_malloc` [132] requires a two-step process: *reserve* and *activate*. The reserve stage allocates memory and returns a corresponding reference. The application can then initialize the data and, afterward, must call the activate method to indicate to the allocator that the memory reserved can now be consolidated. This avoids persistent leakage because a failure between reserve and activate will simply discard the reserved (but not activated) region upon recovery. PMDK [131] also supports a similar API where the allocation procedure modifies the destination pointer (passed as parameter) in a failure-atomic way. Besides the destination pointer, the allocation procedure can also accept a pointer to a function responsible to initialize the allocated data.

Some memory management designs focus on optimizing specific aspects of PM, such as its limited write endurance. NVMalloc [105] prevents wear-out by limiting the re-utilization of memory blocks and keeping as much metadata as possible in DRAM (such auxiliary metadata can be rebuilt upon recovery). NVMalloc also implements techniques to avoid metadata corruption based on checksums. WAlloc [156] proposes the Less Allocated First Out (LAFO) algorithm, which avoids reallocating a block as soon as it is freed and helps with wear-leveling.

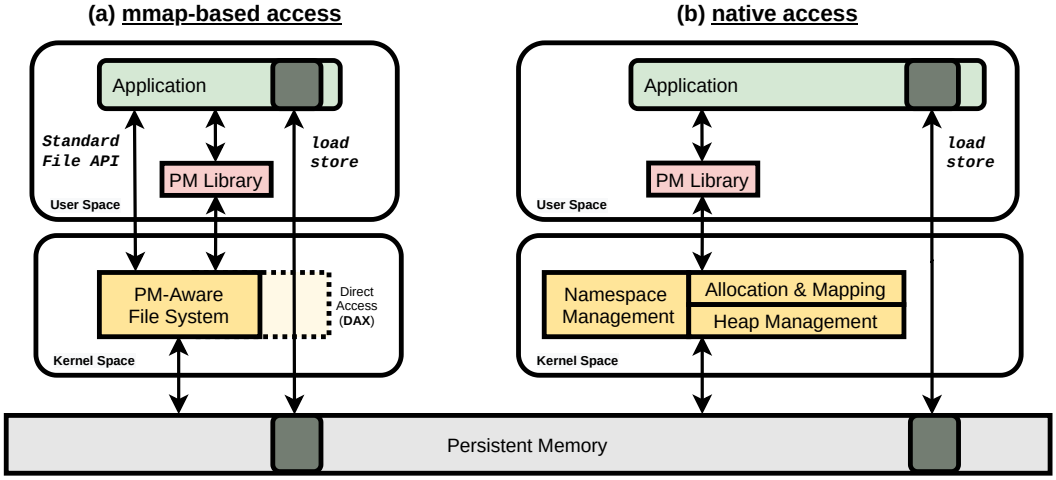


Fig. 12. Two main strategies for accessing persistent data: (a) using the OS file system interface to map the device into the process virtual address space; (b) extending the OS virtual memory subsystem with PM support. Both approaches (assuming a DAX-enabled mmap) can provide direct access to PM through load and store instructions.

4.3 Implementation Aspects

The above sections focus on how persistent heap management is seen from the programmer's point of view. In this section we deal with how such abstractions can be implemented, their robustness, and performance tradeoffs. The discussion is divided into two main parts, the first dealing with low-level PM interfaces (§4.3.1) and the second with persistent pointers handling (§4.3.2).

4.3.1 Low-level PM Interface. Two main strategies exist to interact with persistent data at the lowest level: 1) use the OS file system support; 2) extend the OS virtual memory (VM) system. These strategies are illustrated in Fig. 12.

The first approach (Fig. 12a) relies on the OS file system support to open a file and map (using the `mmap` syscall) its contents into the process address space. After the mapping is performed, the data can be accessed directly by means of load/store instructions in case DAX is enabled [129]. DAX stands for *direct access* and is supported by most OSs nowadays, including Linux and Windows. This method is also the main approach supported by the SNIA programming model [139] and is used by most PM systems, including Mnemosyne [143], NV-Heaps [30] and LibPM [98].

When DAX was still not widely available, accessing PM through `mmap` would generate significant overhead because of the kernel mediation (e.g., page cache) [129]. Slowdowns in throughput of up to 23% have been observed for read-dominated workloads by Huang et al. [66] due to context switches. Some systems, therefore, took another approach and proposed extensions in kernel space (Fig. 12b) so that persistent memory could be incorporated into the VM subsystem. Examples of such systems are SoftPM [59], HEAPO [67], pVM [78], and EFLightPM [66]. Integrating persistent memory into the VM subsystem also allows for seamless memory capacity scaling across DRAM and PM as demonstrated by pVM.

There is not a consensus about which form of accessing PM is preferable. On the plus side, using the file system support simplifies the implementation and can provide some basic protection through the file access control mechanisms. Although initially more complex, a native implementation can

potentially solve some of the issues of mmap-based interfaces. For instance, creating a lot of small persistent regions could cause some overhead as every `mmap` involves other related system calls (e.g., `open`). Another example is if the persistent region needs to be expanded but the file cannot grow linearly in the virtual address space (because the addresses are already taken by other processes), requiring the entire file to be remapped. In general, the main criticism in using a mmap-based approach is that it requires reconciling two systems (VM and file system) that are not necessarily compatible [14, 140].

4.3.2 Persistent Pointers. When persistent regions need to be reopened, either explicitly or because they are recovering from a failure, it is very likely that their contents will be mapped to a different address space. Therefore, references to locations inside the region might need to be updated so that they point to the correct object, creating a demand for *position-independent persistent pointers*. There are three main schemes for dealing with persistent pointers: **static pointers**, **pointer swizzling**, and **fat pointers**.

Using static pointers is the simplest but the most restrictive scheme. It does not require any special pointer representation (i.e., virtual addresses are still used), not adding any overhead, but forcing the persistent region to be mapped to the same address every time. This is particularly problematic if `mmap` is being used, as there is no way to enforce a static mapping. Even when the flag `MAP_FIXED` is used to force the mapping to a specific address, if the memory region overlaps pages of existing mappings, the OS might simply discard the overlapped part of existing mappings, which could further lead to memory corruption [19]. Another disadvantage of static mapping is that it restricts the use of Address Space Layout Randomization (ASLR), increasing the risk of security threats. Systems such as Mnemosyne [143], Atlas [25], and HEAPO [67] assume static pointers.

Pointer swizzling [106] is a method for converting position-independent attributes (most commonly, address offsets) to direct pointer references. Systems that employ some form of pointer swizzling, such as SoftPM [59] and LibPM [98], store enough metadata in the persistent region so that the pointers can be updated when the region is opened or recovered. A similar approach is employed by Cohen et al. [32] for pointer relocation and reconstruction of persistent objects in C++. Notice that this scheme incurs a one-time overhead and accessing the region after it is loaded will not add any extra cost.

The most predominant position-independent scheme among PM systems is based on fat pointers [131]. A fat pointer is composed of a region identifier and a byte offset within the region. Dereferencing a pointer is a costly operation, as it requires finding the base address of the region associated with the identifier (the result can be cached) and adding the offset. Moreover, the size of fat pointers is usually bigger than direct pointers, also incurring a space overhead. For instance, NV-Heaps [30] and PMDK use 128-bit fat pointers (a native pointer is 64-bit).

Chen et al. [28] introduced the concept of *implicit self-contained representation* (ISCR) for pointers. A pointer is in ISCR if the following conditions apply: (i) it is no larger than a normal pointer; (ii) all information needed to dereference the pointer is self-contained; (iii) programmers use the pointer to access data the same way they would with a regular pointer. Notice that condition (i) already rules out fat pointers. The authors present two ISCR designs, one called *off-holder* for intra-region references, and another called *Region ID in Value* (RIV), that also work for inter-region references. Off-holder simply stores the difference between the pointer's target address and its own (similar to the PC-relative addressing mode used by processors). RIV utilizes the unused bits in the address to encode the region ID. Extensions to the type system of C/C++ are proposed to improve the pointer's usability [28].

From the performance's perspective, Chen et al. [28] report a time overhead of 3x for both fat pointers and pointer swizzling. They further report that off-holder and RIV provide performance

close to native pointers. It is important to note that their evaluation was performed on a PM emulator. Wang et al. [146] proposed that region IDs should have their own memory address space and provided hardware support (including new load/store instructions) to accelerate the translation from region IDs to virtual addresses. Finally, Bittman et al. [14] present the design of persistent pointers in the Twizzler OS. They also represent pointers as a region ID and offset pair but, since they require 128-bit region IDs, an indirection level is necessary to avoid storing the large IDs within the pointer. Their approach resembles fat pointers but they target a truly global address space (composed of multiple machines), for which RIV pointers will not suffice.

5 CONCLUSIONS

Exploiting the full potential of new byte-addressable persistent devices will require new programming support and efficient runtime systems. This survey presents the main published approaches in the literature to deal with failure-atomic updates, persistent heap management, and optimizations.

Whereas allowing direct access to persistent data provides clear advantages over traditional approaches employed by file and database systems, it also brings new challenges to programmers. Current systems deal with data consistency issues by supporting high-level FASEs, which may be based on epochs, synchronization primitives (such as locks), or transactions. In particular, as our survey showed, transactions have been widely adopted as the main programming feature. However, our survey also indicated a lack of formalization concerning the semantics of persistent transactions. As envisioned by SNIA [117], most programming support for PM is currently provided as libraries. We, therefore, expect works focusing on the integration with programming languages and compilers to appear, as evidenced by recent papers on the extension of the C++ memory consistency model to support persistence. An alternative approach based on frameworks for constructing persistent data structures is also promising.

In addition to failure-atomic sections, how persistent data is presented and accessed have also been an important research topic. Although more transparent and programmer-friendly approaches exist, our survey indicates that the majority of current state-of-the-art systems assume selective persistence, which requires programmers to explicitly indicate the data that should be persisted. We also expect new research focusing on more attractive approaches such as those based on transitive persistence. On the memory management front, it is important that new algorithms take into account the specificities of the new PM devices, such as limited write endurance.

Finally, there has been a lot of research on efficient implementations of failure-atomic sections. Most implementations employ a variation of WAL, shadow paging, and copy-on-write. Reducing the number of persistent fences is a key factor, as they potentially stall the execution until the data has been persisted and therefore are a costly operation. Elaborated algorithms have been proposed in software to alleviate the problem, some even using HTM support. There have also been a number of papers proposing hardware-assisted logging in order to increase efficiency. We expect the trend on hardware approaches to continue and some of today's ideas to eventually become commercially available in future systems.

ACKNOWLEDGMENTS

This work was supported by FAPESP (2018/15519-5, 2019/10471-7), FCT (UIDB/50021/2020), and EU's H2020 R&I programme (EPEEC project, GA 801051).

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec. 2010), 2237–2251.

[2] 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. 2013. Spin-Transfer Torque Magnetic Random Access Memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (May 2013), 13:1–13:35.

[3] Malcolm Atkinson, Ken Chisholm, Paul Cockshott, and Richard Marshall. 1983. Algorithms for a Persistent Heap. *Software: Practice and Experience* 13, 3 (1983), 259–271.

[4] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *The VLDB Journal* 4, 3 (July 1995), 319–402.

[5] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. 1983. An Approach to Persistent Programming. *Comput. J.* 26, 4 (Nov. 1983), 360–365.

[6] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. 1996. An Orthogonally Persistent Java. *ACM SIGMOD Rec.* 25, 4 (Dec. 1996), 68–75.

[7] Hillel Avni and Trevor Brown. 2016. PHyTM: Persistent Hybrid Transactional Memory for Databases. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 409–420.

[8] Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware Transactions in Nonvolatile Memory. In *DISC'15*. 617–630.

[9] Anirudh Badam. 2013. How Persistent Memory Will Change Software Systems. *Computer* 46, 8 (Aug. 2013), 45–51.

[10] Alexandro Baldassin, Rafael Murari, João P. L. de Carvalho, Guido Araujo, Daniel Castro, João Barreto, and Paolo Romano. 2020. NV-PhTM: An Efficient Phase-Based Transactional System for Non-Volatile Memory. In *Euro-Par'20*. 477–492.

[11] A. Bensoussan, C. T. Clingen, and R. C. Daley. 1972. The Multics Virtual Memory: Concepts and Design. *Commun. ACM* 15, 5 (May 1972), 308–318.

[12] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-Volatile Memory. In *OOPSLA'16*. 677–694.

[13] Kumud Bhandari, Chakrabarti, Dhruva R., and Hans-J. Boehm. 2012. Implications of CPU Caching on Byte-Addressable Non-Volatile Memory Programming. Technical Report HPL-2012-236.

[14] Daniel Bittman, Peter Alvaro, and Ethan L. Miller. 2019. A Persistent Problem: Managing Pointers in NVM. In *PLOS'19*. 30–37.

[15] Colin Blundell, E. Christopher Lewis, and Milo M.K. Martin. 2006. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Comput. Archit. Lett.* 5, 2 (Feb. 2006), 17–17.

[16] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *PLDI'08*. 68–78.

[17] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-Volatile Memory. In *ISMM'16*. 55–67.

[18] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. 2018. Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM Trans. Des. Autom. Electron. Syst.* 23, 2 (Jan. 2018), 1–32.

[19] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel* (3th ed.). O'Reilly Media.

[20] Miao Cai, Chance C. Coats, and Jian Huang. 2020. HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory. In *ISCA'20*. 584–596.

[21] Wentao Cai, Haosen Wen, H. Alan Beadle, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *PPoPP'20*. 421–422.

[22] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable Persistent Hardware Transactions. In *FAST'21*. 155–169.

[23] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *IPDPS'18*. 368–377.

[24] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. 2010. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *SC'10*. 1–11.

[25] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *OOPSLA'14*. 433–452.

[26] Albert Chang and Mark F. Mergen. 1988. 801 Storage: Architecture and Programming. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 28–50.

[27] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508.

[28] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-Volatile Memory. In *MICRO'17*. 191–203.

[29] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797.

[30] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *ASPLOS'11*. 105–118.
[31] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *ASPLOS'19*. 441–454.
[32] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-Oriented Recovery for Non-Volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 153:1–153:22.
[33] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 67:1–67:24.
[34] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *SPAA'18*. 259–269.
[35] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-Addressable, Persistent Memory. In *SOSP'09*. 133–146.
[36] Intel Corporation. 2020. Intel® Architecture Instruction Set Extensions Programming Reference. (2020).
[37] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *SPAA'18*. 271–282.
[38] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent Memory and the Rise of Universal Constructions. In *EuroSys'20*. 1–15.
[39] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *USENIX ATC'18*. 373–386.
[40] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *HPDC'16*. 125–136.
[41] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic Persistence for SCM with a Non-Intrusive Backend Controller. In *HPCA'16*. 77–89.
[42] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. 2021. TSOPER: Efficient Coherence-Based Strict Persistency. In *HPCA'21*.
[43] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (June 2005), 492–528.
[44] Scott W. Fong, Christopher M. Neumann, and H.-S. Philip Wong. 2017. Phase-Change Memory—Towards a Storage-Class Memory. *IEEE Trans. Electron Devices* 64, 11 (Nov. 2017), 4374–4385.
[45] Xuanyao Fong, Yusung Kim, Rangharajan Venkatesan, Sri Harsha Choday, Anand Raghunathan, and Kaushik Roy. 2016. Spin-Transfer Torque Memories: Devices, Circuits, and Systems. *Proc. IEEE* 104, 7 (July 2016), 1449–1488.
[46] R. F. Freitas and W. W. Wilcke. 2008. Storage-Class Memory: The Next Storage System Technology. *IBM J. Res. Dev.* 52, 4.5 (July 2008), 439–447.
[47] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination Is More Important than the Journey. In *PLDI'20*. 377–392.
[48] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *PLDI'20*. 59–74.
[49] Ellis Giles, Kshitij Doshi, and Peter Varman. 2013. Bridging the Programming Gap Between Persistent and Volatile Memory Using WrAP. In *CF'13*. 1–10.
[50] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM. In *ISMM'17*. 70–81.
[51] Ellis Giles, Kshitij Doshi, and Peter Varman. 2018. Hardware Transactional Persistent Memory. In *MEMSYS'18*. 190–205.
[52] Ellis R. Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *MSST'15*. 1–14.
[53] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *PLDI'18*. 46–61.
[54] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *ISCA'20*. 652–665.
[55] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. *Comput. Surveys* 13, 2 (June 1981), 223–242.
[56] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann.
[57] Paul Green. 2005. Multics Virtual Memory - Tutorial and Reflections. Retrieved from https://multicians.org/pg/mvm.html.
[58] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *USENIX ATC'19*. 913–928.

[59] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software Persistent Memory. In *USENIX ATC'12*. 319–331.
[60] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed Logless Atomic Durability with Persistent Memory. In *MICRO'19*. 466–478.
[61] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *ASPLOS'20*. 775–788.
[62] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.
[63] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming* (1st ed.). Morgan Kaufmann.
[64] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *EuroSys'17*. 468–482.
[65] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *USENIX ATC'17*. 703–717.
[66] Kaixin Huang, Yan Yan, and Linpeng Huang. 2019. EFLightPM: An Efficient and Lightweight Persistent Memory System. In *ICECCS'19*. 154–163.
[67] Taeho Hwang, Jaemin Jung, and Youjip Won. 2015. HEAPO: Heap-Based Persistent Object Store. *ACM Trans. Storage* 11, 1 (Dec. 2015), 3:1–3:21.
[68] Taeho Hwang and Youjip Won. 2019. Copy-on-Write with Adaptive Differential Logging for Persistent Memory. *IEICE Trans. Inf.& Syst.* E102.D, 12 (2019), 2451–2460.
[69] IBM. 2020. IBM i – A Platform for Innovators, by Innovators. Retrieved from https://www.ibm.com/it-infrastructure/power/os/ibm-i.
[70] Intel. 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual. Retrieved from https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.
[71] ISO/IEC. 2015. Technical Specification for C++ Extensions for Transactional Memory.
[72] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *ASPLOS'16*. 427–442.
[73] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *DISC'16*. 313–327.
[74] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory. In *MICRO'18*. 520–532.
[75] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *MICRO'15*. 660–671.
[76] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *ISCA'18*. 452–465.
[77] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-Volatile Memory through Hardware Logging. In *HPCA'17*. 361–372.
[78] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *EuroSys'16*. 1–16.
[79] Terence Kelly. 2019. Persistent Memory Programming on Conventional Hardware. *Queue* 17, 4 (Aug. 2019), 1–20.
[80] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *ASPLOS'16*. 385–398.
[81] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *ISCA'17*. 481–493.
[82] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, William Wang, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2019. Language Support for Memory Persistency. *IEEE Micro* 39, 3 (May 2019), 94–102.
[83] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *ASPLOS'16*. 399–411.
[84] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *MICRO'16*. 1–13.
[85] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with TimeStone. In *ASPLOS'20*. 335–349.
[86] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. 1991. The ObjectStore Database System. *Commun. ACM* 34, 10 (Oct. 1991), 50–63.
[87] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable Dram Alternative. In *ISCA'09*. 2–13.
[88] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010), 143–143.

[89] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *SOSP'19*. 462–477.
[90] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-Based B+tree with HTM. In *ICPP'19*. 1–10.
[91] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS 17*. 329–343.
[92] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *MICRO'18*. 258–270.
[93] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. 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 (ASPLOS '20)*. 1187–1202.
[94] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 411–425.
[95] David E. Lowell and Peter M. Chen. 1997. Free Transactions with Rio Vista. *ACM SIGOPS Operating Systems Review* 31, 5 (Oct. 1997), 92–101.
[96] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for Persistent Memory. In *ICCD'14*. 216–223.
[97] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *USENIX HotStorage' 17*.
[98] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. 2018. LibPM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage* 14, 4 (Dec. 2018), 34:1–34:18.
[99] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *EuroSys'17*. 499–512.
[100] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *ASPLOS'20*. 789–806.
[101] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *ICCD'18*. 413–422.
[102] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. 2008. Single Global Lock Semantics in a Weakly Atomic STM. *ACM SIGPLAN Not.* 43, 5 (May 2008), 15–26.
[103] Sparsh Mittal and Jeffrey S. Vetter. 2016. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Trans Parallel Distrib Syst* 27, 5 (May 2016), 1537–1550.
[104] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162.
[105] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-Addressable Non Volatile Main Memory. In *TRIOS'13*. 1–17.
[106] J.E.B. Moss. 1992. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Trans. Softw. Eng.* 18, 8 (Aug. 1992), 657–673.
[107] David Mulnix. 2020. Third Generation Intel® Xeon® Processor Scalable Family Technical Overview. Retrieved from https://www.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html.
[108] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *ASPLOS'17*. 135–148.
[109] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *FAST'19*. 31–44.
[110] Dushyanth Narayanan and Orion Hodson. 2012. Whole-System Persistence. In *ASPLOS'12*. 401–410.
[111] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. 2015. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. In *EDBT'15*. 689–694.
[112] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *DISC'17*. 37:1–37:16.
[113] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent Is Your Persistent Memory Application?. In *OSDI'20*. 1047–1064.
[114] Tri M. Nguyen and David Wentzlaff. 2018. PiCL: A Software-Transparent, Persistent Cache Log for Nonvolatile Main Memory. In *MICRO'18*. 507–519.

[115] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. 2019. SSP: Eliminating Redundant Writes in Failure-Atomic NVRAMs via Shadow Sub-Paging. In *MICRO'19*. 836–848.
[116] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *HPCA'18*. 336–349.
[117] Jim Pappas. 2018. Why Persistent Memory Matters. How Did We Get Here, and What Lies Ahead?. In *Flash Memory Summit*.
[118] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-Atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *EuroSys'13*. 225–238.
[119] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *ISCA'14*. 265–276.
[120] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *MEMSYS'19*. 304–315.
[121] Taciano D. Perez, Marcelo V. Neves, Diego Medaglia, Pedro H. G. Monteiro, and César A. F. De Rose. 2020. Orthogonal Persistence in Nonvolatile Memory Architectures: A Persistent Heap Design and Its Implementation for a Java Virtual Machine. *Software: Practice and Experience* 50, 4 (2020), 368–387.
[122] Gianlucca O. Puglia, Avelino Francisco Zorzo, Cesar A. F. De Rose, Taciano Perez, and Dejan Milojicic. 2019. Non-Volatile Memory File Systems: A Survey. *IEEE Access* 7 (2019), 25836–25871.
[123] Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 137:1–137:27.
[124] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL (Dec. 2019), 11:1–11:31.
[125] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA (Oct. 2019), 135:1–135:27.
[126] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *APSys'17*. 1–8.
[127] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *MICRO'15*. 672–685.
[128] Andry Rudoff. 2016. Deprecating the PCOMMIT Instruction. Retrieved from https://www.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html.
[129] Andy Rudoff. 2017. Persistent Memory Programming. *;login:* 42, 2 (2017).
[130] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. 1993. Lightweight Recoverable Virtual Memory. In *SOSP'93*. 146–160.
[131] Steve Scargall. 2020. *Programming Persistent Memory - A Comprehensive Guide for Developers* (1st ed.). Apress.
[132] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm_malloc: Memory Allocation for NVRAM. In *ADMS'15*. 61–72.
[133] Russell Sears and Eric Brewer. 2006. Stasis: Flexible Transactional Storage. In *OSDI'06*. 29–44.
[134] Margo Seltzer, Virendra Marathe, and Steve Byan. 2018. An NVM Carol: Visions of NVM Past, Present, and Future. In *ICDE'18*. 15–23.
[135] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *MICRO'17*. 178–190.
[136] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *ISCA'17*. 175–186.
[137] Thomas Shull, Jian Huang, and Josep Torrellas. 2018. Defining a High-Level Programming Model for Emerging NVRAM Technologies. In *ManLang'18*. 1–7.
[138] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *PLDI'19*. 316–332.
[139] SNIA Technical Position. 2017. NVM Programming Model Version 1.2.
[140] Yan Solihin. 2019. Persistent Memory: Abstractions, Abstractions, and Abstractions. *IEEE Micro* 39, 1 (Jan. 2019), 65–66.
[141] Kosuke Suzuki and Steven Swanson. 2015. A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014. In *IMW'15*. 1–4.
[142] Mark Tyson. 2019. Intel Optane DC Persistent Memory Launched. Retrieved from https://hexus.net/tech/news/storage/129143-intel-optane-dc-persistent-memory-launched/.
[143] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS'11*. 91–104.
[144] Hu Wan, Youyou Lu, Yuanchao Xu, and Jiwu Shu. 2016. Empirical Study of Redo and Undo Logging in Persistent Memory. In *NVMSA'16*. 1–6.

[145] Chao Wang and Michael Spear. 2016. Practical Condition Synchronization for Transactional Memory. In *Eurosys'16*. 1–16.
[146] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. 2017. Hardware Supported Persistent Object Address Translation. In *MICRO'17*. 800–812.
[147] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. 2015. Persistent Transactional Memory. *IEEE Comput. Archit. Lett.* 14, 1 (Jan. 2015), 58–61.
[148] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. 2020. MorLog: Morphable Hardware Logging for Atomic Persistence in Non-Volatile Main Memory. In *ISCA'20*. 610–623.
[149] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An Early Evaluation of Intel's Optane DC Persistent Memory Module and Its Impact on High-Performance Scientific Applications. In *SC'19*. 1–19.
[150] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (Dec. 2010), 2201–2227.
[151] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *FAST'21*. 141–153.
[152] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory. In *ASPLOS'18*. 70–83.
[153] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *PLDI'20*. 623–637.
[154] Fei Xia, De-Jun Jiang, Jin Xiong, and Ning-Hui Sun. 2015. A Survey of Phase Change Memory Systems. *J Comput Sci Technol* 30, 1 (Jan. 2015), 121–144.
[155] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST'20*. 169–182.
[156] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. 2015. WAlloc: An Efficient Wear-Aware Allocator for Non-Volatile Main Memory. In *IPCCC'15*. 1–8.
[157] Pantea Zardoshti, Michael Spear, Aida Vosoughi, and Garret Swart. 2020. Understanding and Improving Persistent Transactions on Optane™ DC Memory. In *IPDPS'20*. 348–357.
[158] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. 2019. Optimizing Persistent Memory Transactions. In *PACT'19*. 219–231.
[159] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *USENIX ATC'19*. 897–912.
[160] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *MICRO'13*. 421–432.