J-NVM: Off-heap Persistent Objects in Java

Anatole Lefort
Télécom SudParis
IP Paris

Yohan Pipereau
Télécom SudParis
IP Paris

Kwabena Amponsem
Télécom SudParis
IP Paris

Pierre Sutra
Télécom SudParis
IP Paris

Gaël Thomas
Télécom SudParis
IP Paris

Abstract

This paper presents J-NVM, a framework to access efficiently Non-Volatile Main Memory (NVMM) in Java. J-NVM offers a fully-fledged interface to persist plain Java objects using failure-atomic blocks. This interface relies internally on proxy objects that intermediate direct off-heap access to NVMM. The framework also provides a library of highly-optimized persistent data types that resist reboots and power failures. We evaluate J-NVM by implementing a persistent backend for the Infinispan data store. Our experimental results, obtained with a TPC-B like benchmark and YCSB, show that J-NVM is consistently faster than other approaches at accessing NVMM in Java.

CCS Concepts: • **Hardware** → **Non-volatile memory**; • **Software and its engineering** → **Runtime environments**.

Keywords: Non-Volatile Main Memory, Java.

ACM Reference Format:
Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, and Gaël Thomas. 2021. J-NVM: Off-heap Persistent Objects in Java. In *ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483579

1 Introduction

Many modern data stores and big data analytics platforms are written in Java [8, 9, 16, 32, 45, 58, 64, 68]. Because they manipulate large amount of persistent data, these systems can greatly benefit from the recent technological advances in Non-Volatile Main Memory (NVMM) [25]. Unfortunately, to date, accessing efficiently NVMM from the Java language is still an open challenge.

To access NVMM from Java, two designs exist so far. With the *external design*, NVMM remains outside the Java heap. The Java virtual machine (JVM) accesses it through a file system [30, 34, 67], or using the Java native interface (JNI) [46, 48]. This design is inefficient due to the cost of converting data back and forth between the NVMM and the Java representations. With the *integrated design*, the JVM stores plain Java objects in NVMM and the application directly accesses them with read and write instructions [59, 66]. While this design avoids the conversion cost, it also has a fundamental flaw: the JVM has to run a garbage collector (GC) in NVMM because it now contains Java objects.

Collecting a dataset at the scale of NVMM, that is hundreds of GBs to TBs, is expensive.[1] For instance, we show in §2.2.1 that collecting just 80 GB divides the completion time by 3. Moreover, persistent and volatile objects have different life cycles. Applications often contain many allocation and deletion sites for volatile objects. On the contrary, the deletion of a persistent object is often related to a specific event, e.g. discarding a tuple in a relational database. Such events are rare, explicit, and trigger well-defined paths in the application. As we confirm in §2.2.2, this makes the number of deletion sites small, and thus the use of a GC for NVMM superfluous.

To remedy these problems, we propose a direct NVMM access, as with the integrated design, but without collecting persistent objects at runtime. Implementing this design is challenging because the Java language was designed for garbage-collected objects. To address this challenge, we revisit how to manage persistent objects for Java in the NVMM era. We introduce a *decoupling principle* between the data structure of a persistent object and its representation in the Java world. Based on this principle, a persistent object now consists of two parts: a data structure stored off-heap in NVMM, and a proxy that remains in volatile memory. The data structure holds the fields of the persistent object, while the volatile proxy provides the methods that manipulate them. Because we store the persistent data structure outside the Java heap, using our own memory layout, they are not collected at runtime. Our design also removes the cost of converting objects by leveraging a JVM interface that inlines the low-level instructions that access NVMM directly in the Java methods.

We implement our decoupling principle in the J-NVM framework. J-NVM is entirely written in Java and it only requires the addition of three NVMM-specific instructions to the Hotspot JVM. Our framework offers to the programmer a low-level interface that focuses on performance and a high-level interface that trades performance for usability. The low-level interface defines the methods that allow a proxy to access the persistent data structure. The high-level interface

[1] The smallest Optane DC holds 128 GB of persistent memory.

additionally provides failure-atomic blocks, that is blocks of code executed entirely or not at all [12, 14, 15, 44, 49, 62]. To ease programming, J-NVM includes a code generator which takes as input a legacy Java class and automatically decomposes it into a persistent data structure and a volatile proxy. Our framework also includes the J-PDT library which contains optimized persistent data structures (e.g., arrays, maps and trees) implemented directly atop the low-level library. Internally, these data structures do not rely on failure-atomic blocks for performance, yet they remain consistent when a crash occurs.

We evaluate J-NVM with micro-benchmarks and by implementing several persistent backends for the Infinispan data store [41]. Our evaluation using a TPC-B like workload [61] as well as YCSB [13] shows that:

- Both the low-level and the high-level interfaces systematically outperform the external design. In YCSB, the low-level interface is at least 10.5x faster than using ext4 atop NVMM or the PCJ library [48], which relies on the native PMDK library [48], except in a single case where it is only 3.6x faster.
- While the failure-atomic blocks of the high-level interface offer an all-around solution, J-PDT, with its hand-crafted persistent data types, executes up to 65% faster. Compared to a volatile implementation, J-PDT is only 45-50% slower.
- Integrating NVMM in the language runtime hurts performance due to the cost of garbage-collecting the persistent objects. For a Redis-like application written with go-pmem [22], increasing the persistent dataset from 0.3 GB to 151 GB multiplies the completion time of YCSB-F by 3.4

In the rest of this paper, we first motivate our programming model and detail its usage (§2). Further, we present the key building blocks of J-NVM and the low-level interface (§3). The persistent heap is then detailed, as well as the internals of the J-PDT library (§4). The evaluation follows (§5). We then cover the related work (§6) before closing (§7).

2 Programming model

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

2.1 Overview

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

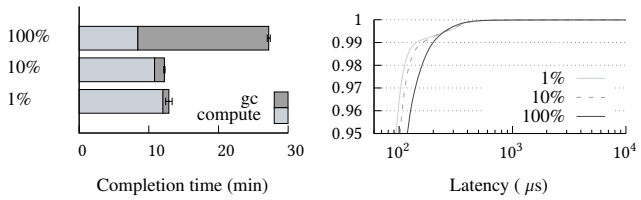

Figure 1. YCSB-F in Java with different cache ratios.

The above decoupling principle avoids running a garbage collector on persistent objects. Based on it, J-NVM implements a complete developer-friendly interface that offers failure-atomic blocks. To construct this interface, J-NVM reuses ideas and principles proposed in prior works, but assembles them differently. Our framework uses a class-centric programming model, that is the property of durability is attached to a class, and not to an instance. As common with prior frameworks (e.g., Thor [35]), a persistent object is live by reachability from a set of user-defined persistent roots. J-NVM garbage collects the unreachable persistent objects at recovery, but avoids running a GC at runtime for performance. Instead, objects are explicitly freed by the developer.

In what follows, we present evidences that garbage collecting persistent objects in a data store is not necessary. There are too few deletion sites in the code base and the existing GCs do not scale to the size of the persistent dataset. Then, we further detail our programming model and illustrates its usage with an example.

2.2 Memory management

Running a GC at runtime has a cost. For volatile objects, this cost is balanced by the usefulness of the GC: the GC avoids many bugs and simplifies substantially the code base. This section shows that this is no more the case with persistent objects in the context of a data store.

2.2.1 GC Overhead. First, we measure the cost of running a GC at the scale of a large persistent dataset. We consider G1, the default GC of Hotspot [17], then the tri-color concurrent marking algorithm of go-pmem [22], a recent persistent framework for the Go language.

G1. In this experiment, we evaluate the cost of collecting a large dataset with a state-of-the-art GC algorithm. We focus on the performance of a GC for volatile memory, because GCs for NVMM are not as optimized yet [22]. We evaluate G1, a modern GC that features many optimizations (generations, concurrent marking and parallel compaction). In particular, G1 pauses the application when it compacts the heap, but bounds the pause time by compacting the regions that contain more garbage first.

We evaluate G1 when running the YCSB-F workload with Infinispan. The experiment uses 15M persistent objects, which

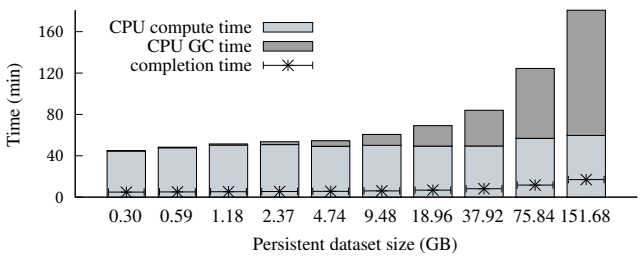

Figure 2. YCSB-F in go-pmem [22] when varying the size of the persistent dataset.

amounts for 15 GB of user data. Infinispan stores these objects in NVMM through the file system interface (DAX ext4). The workload is a mix of read and read-modify-write operations (50/50). In total, 10 threads executes 100M operations.[2]

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

Figure 1(left) reports the completion time of YCSB-F for the three cache ratios. With a cache of 100%, the completion time is roughly multiplied by two. When we analyze this result, we observe that the compute time alone becomes better when the cache ratio increases. This is expected: with a bigger cache, we decrease the number of accesses to the file system, which boosts performance. However, we also notice that, when every object is cached, 69% of the time is spent in GC, erasing the advantage of a larger cache (see breakdown in Figure 1(left)). This shows that, even with a modern optimized GC, collecting a large dataset comes at severe performance cost.

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

go-pmem. The go-pmem framework [22] offers access to NVMM in Go following an integrated design. Its GC collects jointly the persistent and volatile heaps, using a tri-color concurrent marking algorithm [18] adapted for NVMM. The heap is not compacted. It is automatically resized after each collection. Because of a flaw in the resizing policy, applications may end with an out of memory error. To avoid this problem, we force a collection every 10 GB of allocation.

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

Data store	SLOC	# sites
infinispan (this paper)	603, 800	4
cassandra-pmem [11]	334, 300	1
pmem-rocksdb [53]	314, 900	4
pmem-redis [52]	55, 900	1
pmemkv [54]	25, 600	2
go-redis-pmem [22]	8, 400	2
pmse (MongoDB)[50]	4, 800	3

Table 1. NVMM-ready data stores rarely delete persistent objects.

Figure 2 reports the performance of YCSB-F when using the go-redis-pmem data store. This data store is a feature-poor version of Redis [57] written by the authors of go-pmem. We execute the workload of Figure 1 and test different sizes for the persistent dataset. The black line in Figure 2 shows the completion time for each run. With a small dataset, YCSB-F lasts 5 min. The same experiment takes 3.4x more time (17 min) with a large dataset. To understand this drop of performance, Figure 2 shows the accumulated time spent by all the threads in GC (dark gray) and in compute (light gray).

In Figure 2, the compute time is relatively stable. This is expected since the exact same amount of operations is executed in each run. However, increasing the size of the persistent dataset also drastically increases the time spent in GC. With a small dataset, this time is negligible and accounts for less than 1% of the total CPU time. This shows that collecting the heap after 10 GB of allocated data is sufficient for the workload. With the largest dataset, despite that the number of operations remain the same across all runs, the CPU time spent in GC reaches 67% of the total time.

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

2.2.2 Deletion sites. The previous experiment shows that collecting a large heap has a non-negligible cost. Paying this cost is only interesting if the GC actually helps the developer, either by simplifying the code or by avoiding bugs. Table 1 reports the number of deletion sites in several NVMM-ready data stores. We observe that even for data stores with a voluminous code base, a handful of deletion sites exists. This shows that garbage collecting persistent objects at runtime has a limited interest to ease programming.

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

2.3 Data persistence model

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

On the contrary, AutoPersist [59] and Espresso [66] attach the durable property to each instance of a class. We call this approach the *instance-centric* model. Espresso relies on the `pnew` keyword to allocate an instance of a given class directly in persistent memory. Similarly to Thor [35], AutoPersist first allocates the object in volatile memory, and once it becomes referenced by another persistent object, it is moved to persistent memory.

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

Reliability. By hiding durability from the type system, the instance-centric model provides a form of orthogonal persistence [4]. Hiding durability can be harmful because neither the developer nor the compiler can easily identify the persistent state of the application. This problem is also underlined by George et al. [22]: "as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory". The developer can make mistakes by thinking that an object resides in persistent memory while it resides in volatile memory, which leads to data loss [29]. Conversely, the developer can also move to NVMM more objects than necessary, leading to memory leaks. Because these bugs only happen at runtime, identifying them is difficult [12, 37].

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

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

AutoPersist avoids cross-references by instrumenting the code. When the application writes a reference to `b` in an object `a`, if `a` is in NVMM and `b` in volatile memory, AutoPersist transparently migrates `b` to NVMM. The code instrumentation in AutoPersist has a non-negligible cost. Even if the application does not use NVMM, it induces an overhead of 51% (9% with the QuickCheck optimization [60]).

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

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

2.4 Liveness by reachability

Programming with NVMM requires to deal with two fundamental concerns. First, after a crash, an object may be lost, which leads to a memory leak. This may happen when an object is allocated but not yet reachable from a persistent root. Second, after a crash, a reachable object may be partially deleted and thus unusable. Such a situation occurs when an object is freed but not yet removed from the reachable graph. To avoid both problems, J-NVM focuses on simplicity. It considers that once created, a persistent object remains alive as long as it is reachable from a persistent root. This approach is called liveness by reachability and commonly found in many frameworks (e.g., [35, 62]).

Liveness by reachability is implemented in J-NVM using a recovery-time GC, similarly to Makalu [7]. The GC traverses the graph formed by the persistent objects when the application resumes after a crash. This does not happen at runtime to limit the impact on performance (see §2.2.1). During the collection pass, if J-NVM finds a reference to a partially deleted object inside the reachable graph, the reference is nullified. Then, to prevent memory leaks, the persistent objects that are unreachable from a persistent root are deleted.

2.5 Example usage

As illustrated in Figure 3, programming with J-NVM is straighforward. Any class annotated with `@Persistent` is durable. For instance, this is the case of `Simple` in Figure 3 (line 1).

To run the application, the developer compiles the sources as usual, then passes a code generator over the bytecode files (the `.class` files). Any class marked with `@Persistent` is

transformed.[3] The code generator replaces the volatile fields with persistent ones (lines 3 and 4 in Figure 3). Accordingly, accesses to such fields are replaced by persistent accesses (lines 8, 9, 12, 27 and 28). If a field is marked `transient` (line 5), the code generator keeps it in volatile memory, making no transformation. The developer may use transient fields to optimize the application, e.g., to deduce a volatile value from the persistent state (see §3.1).

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

In the `Main` class of Figure 3, the application manipulates a `Simple` object. It starts by creating (or retrieving) a persistent memory region of 1 MB called `"/mnt/pmem/simple"` (line 17). A persistent memory region contains by default the persistent map `JNVM.root`. This map associates names with the root persistent objects used by the application. The `main` method uses this map to retrieve the persistent object associated with the name `"simple"`. If the object does not exist (line 19), the method allocates a new `Simple` object and records it in the map (line 20). Further, `main` retrieves the `simple` object, increments its `x` field, sets its `y` field and prints its content (lines 22-28). Line 30 creates a second `Simple` object and inserts it in the root map. The code then frees the first object still referenced by the local variable `s` (lines 31-32).

2.6 Summary

J-NVM offers a general framework to inject durability in a Java application. Starting from a set of legacy classes to persist, the developer annotates them to generate appropriate proxies. Once transformed, the methods of the proxies are failure-atomic, that is they execute entirely, or not at all, despite system failures. Because J-NVM exposes a class-centric model that favors reliability over re-usability, the developer cannot directly use the volatile classes from the Java runtime in a persistent object (e.g., native arrays, or `java.lang.String`). Instead, the developer should use the drop-in persistent replacements provided in the J-PDT library (such as `PString` at line 9 in Figure 3).

With J-NVM, the developer has to manage differently the life cycle of a persistent object than a volatile one. Liveness by reachability requires a persistent object to be reachable from the root map (`JNVM.root`). This restriction is commonly found in prior frameworks, such as the PMDK library [49]. Persistent objects have also to be explicitly deleted where appropriate. As illustrated at lines 31-32 in Figure 3, explicit deletion makes the code slightly more complex than with an integrated design. However, as shown in §2.2.2, such events are rare in persistent data stores.

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

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

Figure 3. How to use J-NVM.

3 System Design

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

Figure 4 shows a simplified version of the persistent object generated by the code generator of J-NVM from the example in Figure 3. The code generator is based on the ASM framework [3, 10], which provides helpers to simplify the rewriting of bytecode. To obtain Figure 4, the code generator first adds the `POBject` interface, which marks objects of the `Simple` class persistent, then it removes all the non-transient volatile fields. Any access to a non-transient field (e.g., lines 8 and 9 in Figure 3) is then transformed into a call to a generated method that accesses the persistent data structure (lines 7, 8 and 15 in Figure 4). Because of the `non-private` arguments at line 1 in Figure 3, the code generator wraps each method into a failure-atomic block. Such blocks (lines 6-10 and 14-16 in Figure 4) are delimited with `faStart()..faEnd()`.

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

3.1 Life cycle

Association. By design, J-NVM decouples the persistent data structure of an object from the proxy that represents it in the Java world. J-NVM has thus to maintain an *association* between a proxy and the persistent data structure it gives access to. The `addr` field (line 20 in Figure 4) maintains such an association: it contains the address of the persistent data structure associated with the proxy. The getters and setters use this address to execute operations over the persistent fields. For instance, `x` is the second persistent field of `Simple` in Figure 4. As a consequence, `getX` returns the integer located at offset 8 in the persistent data structure at address `addr`.

Allocation. The data structure of a persistent object is stored in NVMM. J-NVM allocates it in the constructor, using `JNVM.alloc` (line 7 in Figure 4). Once allocated, the persistent data structure is associated with the corresponding proxy. The method `alloc` takes two arguments: the Java class of the proxy, which is used during resurrection (see details below), and the size of the data structure.

Persistent references and resurrection. A persistent object may hold a reference to another persistent object. As with primitive types, J-NVM provides the `readPObject` and `writePObject` methods to manipulate them (see for instance lines 27-28 in Figure 4). Because these methods manipulate proxies, the Java type system ensures that NVMM contains only references to persistent objects, and not to volatile ones.

To store a reference to an object `a`, `writePObject` stores `a.addr` in NVMM. Upon dereferencing `a`, `readPObject` dynamically creates a proxy associated to its persistent data structure. In detail, `readPObject` reads the address of the persistent data structure in NVMM, retrieves the name of the proxy class in its header, then allocates the corresponding proxy class. Once the proxy is allocated, `writePObject` calls the constructor generated at line 22 in Figure 4, before returning the proxy to the caller.

We call this constructor the *resurrect constructor*.[4] The resurrect constructor first associates the proxy with the persistent data structure (line 23). Then, it calls the `resurrect` method (line 24). This method, if overridden, may initialize transient fields upon resurrection (e.g., the `y` field in our example).

Free. As seen at lines 31-32 in Figure 3, calling `JNVM.free` frees the persistent object. This method takes a proxy as argument. It frees the persistent data structure associated with the proxy and writes 0 in its `addr` field, which makes the proxy invalid: after a call to `free`, accessing the proxy throws an exception.

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

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

Figure 4. A generated persistent object.

3.2 Low-level interface

Prior research [5, 21, 24, 39, 70] shows that constructs such as failure-atomic blocks and transactions are costly, and in many cases not required by the application. For this reason, J-NVM also exposes a low-level interface that trades the simplicity of the high-level interface for better performance.

To use the low-level interface, the developer omits the `fa` argument in the `Persistent` annotation at line 1 in Figure 3. In that case, the code generator performs the same transformation as described above, but it does not wrap methods in failure-atomic blocks. To still create such blocks, the developer can call `faStart()..faEnd()` explicitly. It is also possible to fine-grained manage data persistence without failure-atomic blocks at all, as we detail next.

Outside a failure-atomic block, the field accessors behave differently. To achieve this, J-NVM maintains a per-thread counter that tracks the nested level of failure-atomic blocks. At runtime, J-NVM checks this counter when it loads or stores a field. If the counter is strictly greater than 0, J-NVM instruments the load/store to ensure that the failure-atomic blocks execute atomically. Otherwise, it grants a direct access to NVMM without mediation. We measured that checking this counter for each access has almost no performance impact because the counter is always in the L1 cache of

the processor and the branch predictor makes correct predictions. For that reason, we have not investigated static analysis methods to eliminate them.

Accessing NVMM without mediation requires to ensure data persistence by hand. The remainder of the section is devoted to describing the interface that J-NVM provides to help the developer in this task.

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

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

When using the low-level interface, J-NVM does not enforce any propagation order, allowing the developer to make optimizations. J-NVM exposes three operations to control propagation to NVMM: `pwb`, `pfence` and `psync`. These operations implement the architecture-agnostic instructions defined by Izraelevitz et al. [24]. We adapted them to work with the Java memory model [26, 40], as also proposed in JEP 352 [27].

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

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

3.2.3 Validation and recovery. Calling `pfence` prevents out-of-order execution inside the processor. This drastically decreases the instruction-level parallelism. As a consequence, reducing the number of `pfences` in the application is paramount for performance [14].

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

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

Figure 5. The low-level interface.

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

Figure 5 illustrates how the developer may use the validation mechanism internal to J-NVM. The optimization consists of deferring validation after the pfence at line 16, to allocate and make reachable several objects with a unique `pfence`. In detail, the code allocates two objects `a` and `b` (lines 14-15), which themselves allocate a sub-object each (line 6). The two objects are added to the root map (line 10) using `wput`. This operation is weak, in the sense that it does not rely internally on a failure-atomic block, and thus does not executes `pfences`. The calls to `pwb` at lines 7 and 9 ensures that the cache lines of `a`, `b`, `a.o` and `b.o` are all added to the write pending queue. The call to `validate` at line 8 validates `a.o` and `b.o`. This call does not execute `pfence`: it just changes the `valid` state of the object and adds the cache line of the header to the write pending queue.

If a crash happens before line 16, since `a` and `b` are invalid, J-NVM will free them at recovery, even if they are already reachable from the root map. J-NVM will also delete `a.o` and `b.o` because they are not reachable from a live object. As a result, in case of a crash before line 16, all the allocated

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

Table 2. The block header and its associated states.

objects are deleted. Thus, by not executing any `pfence` before line 16, the code is correct. The unique `pfence` at line 16 ensures that if `a` (resp. `b`) is valid (lines 17 and 18), then so do `a.o` (resp. `b.o`).

4 Implementation

This section details the implementation of J-NVM. After a description of the persistent heap, the section presents the algorithm used to ensure failure atomicity and the library of persistent data types.

4.1 The persistent heap

Designing a persistent heap requires to address fragmentation: after multiple object allocations and releases, the free space is divided into small pieces, making allocation of large objects impossible. Solving this problem is essential for a persistent memory since, by definition, it is long lived.

Usually, we eliminate fragmentation in managed languages such as Java by executing a compacting phase during a GC cycle [28]. However, J-NVM avoids the use of a GC at runtime to deliver better performance. To deal with fragmentation, J-NVM relies instead on a memory layout inspired by the work of Pizlo et al. [51]. This layout splits the heap in blocks of fixed size, exactly as we do with the blocks of a file system. If a large object does not fit into a single block, J-NVM creates a linked-list of blocks to store its content.

Using blocks of fixed size eliminates the fragmentation problem by design since we can always allocate large objects. However, this memory layout also increases the complexity of accessing large objects. J-NVM hides this complexity behind proxies. Instead of keeping a single address for the persistent data structure (line 20 in Figure 4), the proxy actually contains an array that holds the addresses of all its blocks. The array is populated during the association between the proxy and the persistent data structure. Once the proxy is initialized, retrieving the block that contains a given field simply requires a division.

4.1.1 Block header. A block starts with a header of a single word that provides its state (see Table 2). For allocated blocks, `next` gives the next block that belongs to the object. When `id` is not equal to 0, the block is the first block of a persistent object, called the *master block*. In this case, J-NVM uses `id` as an index in a persistent array to retrieve the name of the proxy class during resurrection (§3.1). Otherwise, when `id` equals 0, `valid` is necessarily equal to 0. We have then two possibilities. The block can be a *slave* block, which means that it belongs to a persistent object but it is not the first block. Alternatively, the block may also be free in which case it does not belong to any object.

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

4.1.3 Recovery. At startup, J-NVM executes a recovery procedure. To create the volatile free queue, this procedure uses a volatile bitmap. For each block, the bitmap indicates with a bit whether the block is free or not. Starting from the root map, J-NVM traverses the live object graph. As any graph traversal, this procedure has thus a complexity linear in the number of live objects. When it finds a reference to a valid object, J-NVM marks its blocks as alive in the bitmap, and calls the `recover` method of the object. Otherwise, since the referenced object is invalid, the reference is set to null.

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

4.1.4 Object allocation. When J-NVM allocates an object, it first allocates its blocks using the free queue and the bump pointer. Then, J-NVM writes its `id` in the master block and links appropriately the slave blocks. During the allocation of an object, J-NVM does not use any fence since a master block is necessarily in the invalid state.

4.1.5 Object deletion. To delete an object, the application explicitly calls `JNVM.free`. This method invalidates the master block and adds all its blocks to the volatile queue. J-NVM does not execute a `pfence` in `JNVM.free`, which allows a developer that uses the low-level interface to use a single `pfence` to free a graph of objects. For instance, to free `a` and `a.o` in Figure 5, the developer marks `a` as invalid by calling `JNVM.free` and then explicitly triggers a single `pfence`. In case of a crash after the `pfence`, J-NVM will delete `a` because it is invalid, and `a.o` because it is not reachable by valid objects. Executing a `pfence` in `JNVM.free` for `a.o` is thus useless.

4.1.6 Atomic update. For a developer that uses the low-level interface, J-NVM provides a method that atomically updates a reference. This method ensures that the collection

```
1  void update0(PObject n) {
2   n.validate();
3   pfence();
4   set0(n);
5  }
```

Figure 6. Atomic update.

pass executed at recovery cannot nullify the reference. As shown in Figure 6, the implementation of this method is straightforward. It simply validates the new reference, executes a `pfence` and updates the reference. Calling `pfence` ensures that the new object is valid before being reachable. The code generator creates this method for each field that references a persistent object. It also generates a second helper that updates a reference and additionally frees the old referenced object atomically. Our NVMM portage of the Infinispan key-value store (see §5.1) uses these methods to ensure at all time a sound association between a key and its values.

4.2 Failure atomic blocks

As indicated in §2.5, the high-level programming model of J-NVM provides failure-atomic blocks. Many systems already offer such a construct [12, 14, 15, 22, 44, 49, 62]. Our algorithm is not new by itself. It is inspired by Romulus [14] and adapted to our persistent memory layout. We have implemented failure-atomic blocks not to advance the state of the art, but instead to verify that we can build a developer-friendly system based on our decoupling principle.

J-NVM implements a standard redo log. At a high level, during the execution of a failure-atomic block, J-NVM adds all the modifications (allocations, writes and frees) to a per-thread persistent redo log, leaving original data intact. When reaching the end of the failure-atomic block, J-NVM commits the log then applies its modifications to NVMM.

Before committing the log, J-NVM does not uses any `pfence` as data in NVMM is unchanged [14]. To commit the log, J-NVM first executes a `pfence` to ensure that the state of the log is persisted. Then, it marks the log as committed and executes a second `pfence` to ensure that its new status reaches NVMM. Finally, it applies the operations recorded in the log without `pfence`. If a crash occurs during this last step, the log will be replayed.

To update a persistent object, J-NVM considers two cases. If the object is invalid, J-NVM directly modifies the object. This may happen, for instance, if the object is allocated then modified in the same failure-atomic block. Modifying an invalid object is safe because it is deleted at recovery when a crash occurs before commit. Now if the object is valid, J-NVM maintains two versions of each modified block of the object, an original one and an in-flight one. Upon reading, J-NVM uses the in-flight block if it exists, and the original block otherwise. Upon writing, if the in-flight block is already present, J-NVM directly performs writes to it. Otherwise, it allocates an in-flight block, adds the pair (original, in-flight) to the log then performs writes to the in-flight block.

At the end of a failure-atomic block, J-NVM marks the persistent log as committed then plays the operations recorded in it. If it finds an allocation, J-NVM transparently validates the new object, which makes the object alive if and only if it is reachable. If it finds a deletion, J-NVM calls the `JNVM.free` method. If it finds an update, J-NVM copies the in-flight block into the original block.

After a failure, J-NVM first handles the per-thread logs of failure-atomic blocks, then it executes the recovery procedure (see §4.1.3). If a crash occurs before the block was fully played, J-NVM replays its operations. If the crash occurred before the block was committed, J-NVM aborts it. J-NVM erases the log and lets the recovery procedure garbage collect the in-flight blocks as well as all the objects allocated in the block (these objects are still in the invalid state).

4.3 J-PDT

J-PDT is a stand-alone library of persistent data types built on top of the low-level interface. This section gives an overview of the main data types present in the library.

4.3.1 Persistent arrays. J-PDT provides arrays of fixed sizes. An array contains its length at offset 0 and the elements afterward. This class provides a constructor to initialize its content appropriately, accessors to retrieve the elements, and methods to flush either an element, or the array in full. In addition, J-PDT provides extensible arrays similar to the `ArrayList` class of the standard Java library. To extend an array, we rely on the low-level atomic update methods described in §4.1.6.

4.3.2 Maps and sets. J-PDT includes several set and map abstractions. Implementing these data structures is more challenging than implementing the arrays and extensible arrays detailed previously. However, the decoupling principle introduced by proxies offers a general pattern of solution: the content of a persistent object is stored in NVMM, while its logic remains in volatile memory.

Overview. We first implement a persistent set as a persistent map that associates each key with itself. Then, to construct a persistent map, J-PDT stores the references to the persistent key/value pairs in a persistent extensible array. In the proxy, J-NVM maintains two volatile data structures: a *free queue* that stores the empty cells in the persistent array, and a *mirror map* that mirrors the persistent array in volatile memory. The mirror map implements the logic of the data structure. For instance, for a hash table, we use a Java `HashMap`, and for a persistent binary tree, we use a Java `TreeMap` (a red-black tree).

During resurrection (see §3.1), J-PDT inspects each cell of the persistent array. If it finds a non-null reference to a

pair `(k,v)` at index `n`, it adds the mapping `(k,n)` to the volatile mirror. Otherwise, `n` is added to the volatile free queue. To add a key/value pair, J-PDT first removes a free cell index `n` from the volatile free queue. If the queue is empty, the persistent array is extended and the queue populated accordingly. Then, J-PDT allocates a new pair in persistent memory, and writes its references at index `n` in the persistent array. To remove a key/value pair, J-NVM adds its index, say `n`, to the volatile free queue. Then, it writes a null reference at index `n` in the persistent array. The persistent data structure is always in a consistent state because modifying it incurs a single write to NVMM (at the right index in the persistent array).

Base, cached and eager maps and sets. Resurrecting a persistent object has a performance cost. Indeed, J-NVM needs to allocate a proxy, traverse the linked-list of blocks of the persistent object, and, for some object, deduce a volatile state from the persistent state.

To avoid this cost for values stored in maps and sets, J-PDT proposes different implementations of maps and sets with different trade-offs between performance and memory consumption.

The default implementation presented above is called the *base* implementation, and it favors memory consumption. For each key in persistent memory, the base implementation keeps a proxy in volatile memory, but it systematically allocates a new proxy when the application retrieves a value associated with a key.

Both the *cached* and *eager* maps and sets trade memory consumption for better performance. They maintain a cache of the proxies to the values. The eager implementation populates the cache during resurrection, while the cached implementation populates the cache on demand. In our implementation, the cache contains all proxies but it would be possible to extend this code to include only the hottest proxies.

4.4 Implementation details

This section completes the presentation with implementation details.

NVMM access. J-NVM leverages the `Unsafe` interface to access NVMM. This interface directly inlines assembly instructions in the generated code, similarly to the magic interface of JikesRVM [20]. J-NVM uses this interface to read and write NVMM. We also implemented `pwb`, `pfence` and `psync` with this interface. For the recent Intel architecture used in our experiments, we implemented `pwb` with the `clwb` instruction, and, even though they are conceptually different, we implemented both `pfence` and `psync` with the same `sfence` instruction. As a result, by using the intrinsic mechanism and adding three new instructions, J-NVM accesses NVMM at nearly native speed (see our evaluation in §5.3.5).

Small immutable objects. As presented above, J-NVM stores objects in blocks of fixed size. Consequently, the system is subject to internal fragmentation, that is each object consumes a whole block regardless of its size, potentially wasting NVMM. To avoid internal fragmentation for small immutable objects, e.g., `PString` in Figure 3, J-NVM uses memory pool allocators built atop the default one (§4.1). These allocators are able to pack several objects of the same size in a single block.

Memory pool allocators handle only immutable objects. This comes from the fact that the failure-atomic algorithm described in §4.2 works at the block level and not at the object level. To understand why, consider that two threads execute each a failure-atomic block that modifies an object. If the two updated objects are located in the same block, the block will be replicated twice, and the content of the two replicas will diverge. As reconciling these replicas requires complex algorithms, J-NVM avoids such a situation by packing only immutable objects in the same block.

Relocation. J-NVM ensures that the persistent heap is relocatable. For that, instead of storing absolute addresses in NVMM, it stores only offset relative to the beginning of the heap.

5 Evaluation

In this section, we present the performance of J-NVM across a variety of workloads and provide a detailed comparison against other existing approaches.

5.1 Experimental setup

Hardware and system. The test machine is a quad-Intel CLX 6230 hyperthreaded 80-core server with 128 GB of DRAM and 512 GB of Intel Optane DC (128 GB per socket). It runs Linux 4.19 with gcc 8.3.0, glibc 2.28 and Hotspot 8u232-b03 (commit c5ca527b0afd) configured to use G1. The patch for Hotspot that adds the three NVMM-specific instructions to `Unsafe` (namely, `pwb`, `pfence` and `psync`) contains 200 SLOC. Besides this patch, J-NVM, J-PDT and J-PFA that all together implement our NVMM object-oriented programming framework, encompass about 4000 SLOC.

NVMM runs in App Direct mode and is formatted with the ext4 file system. In this mode, software has direct byte-addressable access to NVMM.

Infinispan. Our experiments use Infinispan, an open-source industrial-grade data store maintained by Red Hat. Infinispan exposes a cache abstraction to the application that supports advanced operations, such as transactions and JPQL requests. We use Infinispan version 9.4.17.Final [41], which contains around 600,000 SLOC (see Table 1). Infinispan runs either with the application (embedded mode), or as a remote storage (server mode). Unless stated otherwise, we use the embedded mode during our experiments and cache up to

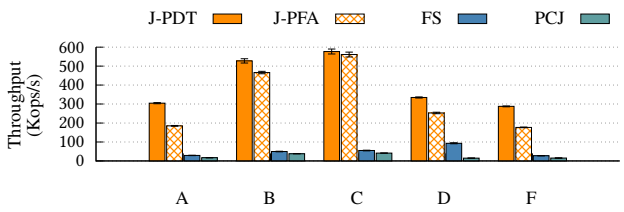

Figure 7. The YCSB benchmark.

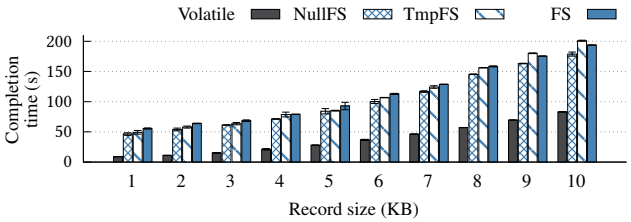

Figure 8. The price to access NVMM from the file system.

10% of the data items. As seen in §2.2.1, a larger ratio would significantly harm performance. Accordingly, we also cap the volatile heap to 22 GB. This size gives the best performance with our YCSB workload on a file system backend atop NVMM (precisely, less then 3.7% of the total time is spent in GC in Figure 7).

For each experiment, we report the average over at least 6 runs along with the standard deviation.

Persistent backends. We evaluate different NVMM-ready persistent backends for Infinispan: ***(J-PDT)*** A backend using the J-PDT standalone library. ***(J-PFA)*** A backend built with the failure-atomic blocks of J-NVM. ***(FS)*** The default file system backend of Infinispan using NVMM formatted in ext4. ***(PCJ)*** An implementation that relies on the Persistent Collections for Java library [48]. PCJ uses the native PMDK 1.9.2 library [49] through the Java Native Interface. For reference purposes, we also consider the following dummy backends without persistence: ***(TmpFS)*** A file system stored in volatile memory. ***(NullFS)*** A virtual file system that treats read and write system calls as no-ops [1]. ***(Volatile)*** A configuration in which persistence is simply disabled. Volatile behaves as NullFS, except that the marshalling/unmarshalling phase is avoided.

The persistent backend using PCJ is 274 SLOC long. The J-PFA and J-PDT backends use the same code base which contains 271 SLOC.

5.2 YCSB

Benchmark. We compare J-NVM against the other approaches by running version 0.18 of the Yahoo! Cloud Serving Benchmark (YCSB) [13] YCSB is a key-value store benchmark that consists of six workloads (A to F) with different access patterns. A client can execute six types of operations (`read`, `scan`, `insert`, `update` and `rmw`) on the key-value store. Workload A is update-heavy (50% of `update`), B is read-heavy (95% of `read`) and C is read-only. Workload D consists of repeated reads (95% of `read`) followed by insertions of new values. In the workload E, the client executes short scans. Workload F is a mix of read and read-modify-write (`rmw`) operations. We evaluate all workloads except E. Infinispan only provides `scan` through the JPQL interface, hence workload E is not comparable with the others that use a direct interface. If not otherwise specified, YCSB executes in sequential mode (single-threaded client).

YCSB associates a key with a data record that contains fixed length fields. Unless otherwise stated, we use the default parameters of 3M records, each having 10 fields of 100 B. YCSB runs with the default access patterns (namely, zipfian and latest). Compared to a uniform distribution, these patterns improve the cache hit ratio, and makes thus the FS backend more efficient.

J-PDT, J-PFA and PCJ all require to use persistent keys and values in YCSB. To achieve this, we modified the Infinispan client, which represent less than 30 SLOC from the vanilla version.

Results. Figure 7 presents the throughput of the YCSB benchmark with the different persistent backends. In this figure, we observe first that J-PDT systematically outperforms the other approaches. Except in workload D, J-PDT is consistently 10.5x faster than FS. In comparison to PCJ, the difference ranges between 13.8x and 22.7x faster. In workload D, J-PDT executes at least 3.6x more operations per second than FS and PCJ.

The low performance of FS comes from the cost of marshalling persistent objects back and forth between their file system and Java representations. Figure 8 highlights this phenomenon. In this figure, 1 KB corresponds to Figure 7. Compared to Volatile, the three file system backends (NullFS, TmpFS and FS) have similar performance. The completion time is between 2.11-6.26x higher than the volatile base line. In particular, NullFS, which fully ignores reads and writes, is just slightly faster than FS. This shows that the main cost comes from data marshalling and not from the file system itself.

In Figure 7, the lower performance of PCJ is due to the Java native interface that requires heavy synchronization to call a native method [46]. J-NVM avoids this cost by leveraging the `Unsafe` interface (§3.2.2), which does not have to synchronize the whole JVM to escape the Java world.

Overall, the results in Figure 7 outline that NVMM drastically changes the way to access persistent data from the Java runtime: while JNI calls or marshalling/unmarshalling operations were negligible with slow storage devices, this is no more the case with NVMM. They must be avoided where possible.

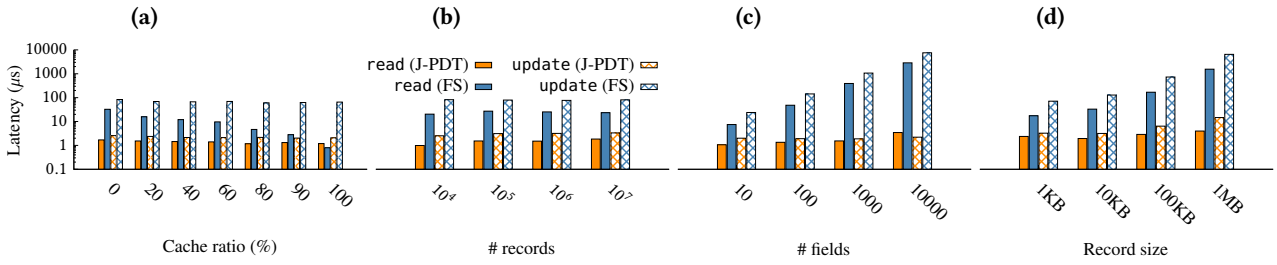

Figure 9. Impact of the cache ratio, the number of records, their composition and size (from left to right).

In Figure 7, J-PFA also systematically outperforms FS and PCJ for the same reasons as mentioned above. Nevertheless, J-PDT is still up to 65% faster. This result shows that handcrafted crash-consistent data structures can be more efficient than a generic approach.

5.3 Performance analysis

This section provides a comprehensive analysis of the performance of J-NVM. We detail the importance of the workload, how J-NVM scales with the number of threads, the time to recover from a crash and the performance of the low-level interface.

5.3.1 Sensitivity to the workload. In what follows, we analyze how J-NVM reacts to workload variations. We use the settings presented in §5.1, but change one parameter at a time. Figure 9 presents our results with YCSB-A. This figure reports only the performance of J-PDT and FS since, as presented earlier, they are respectively faster than J-PFA and PCJ.

Caching. Figure 9a measures the impact of caching persistent data in Infinispan. In this figure, we observe that changing the cache size does not much impact the performance of J-PDT. This observation holds for both reads (from 1.7 μs to 1.2 μs) and updates (from 2.6 μs to 2.1 μs). With J-PDT, only proxies are kept in the cache. In particular, J-PDT never marshal/unmarshal the persistent data structures themselves. As caching brings almost no performance benefits, it is disabled in all our experiments using J-NVM as a backend for Infinispan.

For FS, improving the cache size has almost no performance impact on updates. This comes from the fact that, as Infinispan uses a write-through policy for durability, updates need to access the file system in the critical path. On the contrary, having a larger cache benefits to reads (from 32.5 μs to 0.8 μs) . At 0%, a read systematically fetches data from the file system, which becomes less and less likely when the cache size increases.

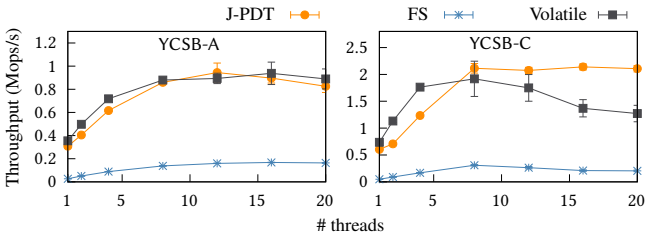

Figure 10. Multi-threaded performance.

With a cache of 100%, J-PDT pays the cost of reading NVMM through proxies, while FS directly uses volatile objects stored in the cache. As a consequence, FS is slightly better than J-PDT for reads in this case (0.8 μs versus 1.2 μs). In this experiment, the dataset is small (3 GB) and thus garbage collection has a limited impact on performance. As underlined in §2.2.1, this caching policy would be problematic with a larger dataset.

Number of records. We now turn our attention towards the impact of the dataset size on performance. Figure 9b presents the access latency when the number of records increases. Overall, we observe that the performance of both J-PDT and FS is stable. The number of records does not impact performance because each operation works on a single record at a time. Furthermore, as neither J-PDT nor FS use a GC to collect the persistent state, the overheads of Figure 2 are avoided.

Record composition. In this experiment, we consider additional fields (Figure 9c) and larger fields (Figure 9d). In both cases, we adjust the number of records to keep a constant dataset size.

We first observe that changing the record composition only moderately impacts the performance of J-PDT. For reads, the latency grows from 1.7 μs to 7.0 μs with more fields (Figure 9c), and from 2.4 μs to 4.0 μs with larger fields (Figure 9d). With updates, the latency grows from 3.6 μs to 4.1 μs with more fields (Figure 9c), and from 3.2 μs to 14.6 μs with larger fields (Figure 9d). This slight increase in latency

comes from the fact that J-NVM has to resurrect more fields, or larger ones.

In Figures 9c and 9d, the read performance of FS significantly degrades when the number of fields increases (from 17.7 μs to 22.3 ms in Figure 9c). This is also the case when the size of each field increases (from 17.5 μs to 1.6 ms in Figure 9d). Updates show a similar pattern: from 71.3 μs to 71.3 ms with more fields, and from 71.0 μs to 6.5 ms with larger fields. As in Figure 8, this degradation comes from the increasing cost of marshalling/unmarshalling voluminous records.

5.3.2 Multi-threading. This section evaluates how J-PDT behaves when the persistent objects are accessed concurrently. Figure 10 presents the throughput achieved in YCSB-A and YCSB-C using 1M records when the number of threads increases from 1 to 20. For both J-PDT and FS, accesses to the persistent state are protected by the locks of Infinispan. Notice that since Infinispan runs in embedded mode, a YCSB thread is also an Infinispan thread.

In Figure 10, the peak performance of J-PDT is slightly higher than Volatile in the two workloads. This surprising result comes from the increased preasure on GC in the Volatile implementation. In YCSB-A, J-PDT saturates Infinispan with 12 threads, while Volatile needs 16 of them. In YCSB-C, J-PDT and Volatile both saturate Infinispan with 8 threads. These results show that J-PDT, with its design based on proxies to access NVMM, does not introduce additional scalability bottlenecks with respect to the volatile implementation. Figure 10 also shows that FS, with a realistic cache ratio of 10%, scales up to 16 threads in YCSB-A and up to 8 threads with YCSB-C. In both workloads, at its peak performance, FS remains more than 5x slower than J-PDT.

5.3.3 Performance of the recovery procedure. In this experiment, we evaluate the time to recover from a crash failure. Figure 11 presents our results. We use a bank application inspired from the TPC-B benchmark [19, 61]. The bank server holds 10M accounts of 140 B each. It provides a single operation to execute a transfer between two accounts in a failure-atomic block. The server runs in a container and exposes a REST interface to remote clients. In Figure 11, the load injector continuously performs transfers between two randomly-selected accounts. It runs on the same machine as the bank server.

After a minute, the container holding the bank server is crashed with SIGKILL, then immediately restarted. Volatile, which only stores the state in DRAM, resumes processing requests after 2.4 s. Because the server restarts from a blank state, accounts are recreated on demand with a 0€ balance after recovery. Volatile is back to its nominal throughput (9.5K ops/s on average) 5.5 s after the crash.

J-PFA restarts processing requests 8.5 s after the crash, and returns to its nominal throughput (8.8K ops/s on average) a few seconds later. J-PFA needs 6.1 s more than Volatile

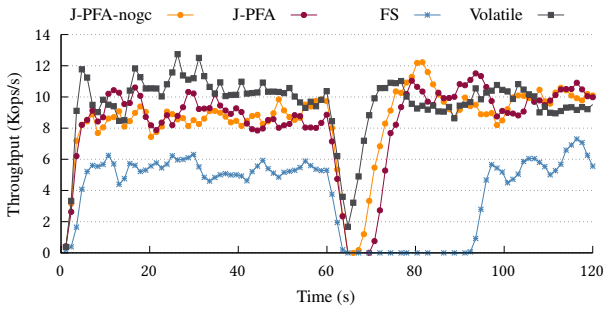

Figure 11. Recovery time with a TPC-B like workload.

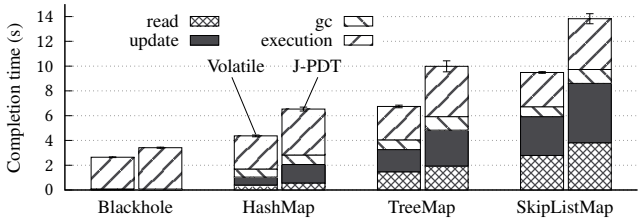

Figure 12. Persistent vs. volatile data types.

to restart because the recovery procedure has to run the recovery GC over the 10M accounts. Volatile does not exhibit this cost because the bank server simply restarts from an empty state.

For completeness, Figure 11 also includes J-PFA-nogc. With J-PFA-nogc, the recovery procedure does not trigger the traversal of the object graph to delete invalid reachable objects. Instead, the recovery procedure only inspects each block, adding invalid ones to the volatile free queue (see §4.1.3). Avoiding the graph traversal is correct in this experiment because the application can not create invalid reachable objects: the server executes both the allocation and the insertion of an account in the database in the same failure-atomic block. We observe that without the graph traversal, J-PFA-nogc restarts processing requests 2.8 s faster than J-PFA.

In Figure 11, FS takes 28.8 s to restart and 34 s in total after the crash to return to normal (4.7K ops/s). This long delay comes from reconstructing eagerly the cache in memory. Upon restart, Infinispan reloads 10% of the accounts (1M) from NVMM. When this occurs, J-PFA pays a lower price because it creates proxies instead of reloading data in full.

5.3.4 Persistent data types. In Figure 12, we compare the persistent maps available in J-PDT against their volatile counterparts in Hotspot (`java.util.*`). Three data types are considered: a hash table, a red-black tree and a skip-list map. In total, this code base covers 629 SLOC.

In Figure 12, we run YCSB-A directly on the data types themselves, without Infinispan. The "Blackhole" histogram

	Sequential		Random	
	Read	Write	Read	Write
J-NVM	3.21 GB/s	0.74 GB/s	0.71 GB/s	0.38 GB/s
C	4.01 GB/s	0.78 GB/s	1.94 GB/s	0.40 GB/s

Table 3. Access to a persistent 256 B-long block.

in Figure 12 corresponds to an execution in which the operations are not applied. In other words, this histogram measures the time spent by the benchmark to inject the workload. Figure 12 shows that J-PDT is 45-50% slower than a volatile implementation. The rationale behind this drop of performance is the following: *(i)* J-PDT handles crashes which requires `pfences` in the critical path; *(ii)* NVMM is slightly slower than volatile memory [25]; and *(iii)* J-PDT relies on proxy objects to access NVMM.

5.3.5 Block size and NVMM access performance. During our experiments, we use a block size of 256 B. We measured that this size provides the best overall performance, because NVMM uses internally also a cache line of 256 B. A YCSB record contains 10 fields. With small fields (100 B) the NVMM space lost due to the block headers and the internal fragmentation accounts for 21.2% per record. This reduces to 9.4% with larger fields (10 KB).

Table 3 presents the throughput to access blocks of 256 B using J-NVM and C. For writes, the benchmark triggers a `pwb` after each CPU cache line (64 B) and it executes a `pfence` after a full block. In Table 3, J-NVM is at most 24% slower than C, except with random reads where it is 2.8x slower. These results show that the `Unsafe` interface allows most of the time to access NVMM at nearly native speed.

6 Related Work

Managing persistent data structures directly from the application through mapped files is an old subject largely studied in operating systems literature [6, 36, 42, 47, 65]. This line of research was fully renewed with the arrival of NVMM. Several file systems tailored for NVMM exist (e.g., NOVA-Fortis [67], SplitFS [30] or Strata [31]). As seen in §5.2, using NVMM as a file system leads to costly marshalling/unmarshalling operations.

Many works focus on offering to the developer a transactional interface inspired by databases [12, 14, 15, 22, 44, 49, 62]. Others directly deal with the low-level NVMM semantic [55, 56] to implement specific data types [5, 21, 23, 24, 39, 70]. A third category of works propose recipes to build persistent data types upon the prior knowledge about volatile constructions [23, 24, 33, 69]. Overall, these works show that general techniques and hand-tuned persistent data types have their pros and cons. In particular, as confirmed by our comparison between J-PFA and J-PDT, failure-atomic blocks are easy to use but often less efficient than hand-tuned data types.

All these prior works target native programming languages and they cannot be readily used in a managed language such as Java. Some recent efforts try to fill this gap by using an integrated design [22, 59, 66], or an external design [38, 48]. We discuss Espresso [66], AutoPersist [59] and Go-pmem [22] in §2.2 and §2.3. We show experimentally that their approaches lead to collecting very large datasets, which negatively impacts performance. In §5, we evaluate PCJ [48], which is in essence similar to LLPL [38], and show that it performs less efficiently than a file system interface.

Panthera [63] stores Java objects in NVMM not to make them durable but to save energy. This system detects cold objects and leverages the GC to migrate them to NVMM. Such an approach is complementary to J-NVM. Interestingly, the experiments reported in [63] confirm that, for a large dataset, the GC has a significant impact on performance. In particular, for a heap of 64 GB, up to roughly a third of the execution time is spent garbage collecting objects.

Regardless of NVMM, several applications avoid running a GC by storing part of their datasets outside the heap. This is notably the case of modern data stores (e.g., Spark [68] and Cassandra [32]). Apache Arrow [2] aims at addressing the problem of making such data structures portable. In [43], the authors propose an efficient volatile lock-free off-heap map. The map allows to modify directly off-heap objects, that is, without copying data between the on- and off-heap spaces. Contrarily to J-NVM, it only manipulates arrays of bytes, and thus requires to marshal/unmarshal its content.

7 Conclusion

This paper presents J-NVM, a principled approach to directly access NVMM outside the Java heap with volatile proxies. Our evaluation using micro-benchmarks and the Infinispan data store shows that J-NVM delivers better performance than other existing solutions.

8 Acknowledgements

We thank our reviewers at SOSP and OSDI, our shepherd, Sasha Fedorova, as well as Jonathan Halliday from Red Hat. This work has been partially supported by the CloudButton EU Horizon 2020 program (No 825184), the ANR Scalevisor project, as well as the Futur and Ruptures program of Fondation Mines-Télécom.

References

[1] A black hole file system that behaves like /dev/null. URL https://github.com/abbbi/nullfsvfs.
[2] Apache Arrow. URL https://arrow.apache.org.
[3] ASM. URL https://asm.ow2.io.
[4] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, July 1995. ISSN 1066-8888.
[5] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-Free Concurrency on Faulty Persistent Memory. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures,*

SPAA'19. ACM, 2019.

[6] André Bensoussan, Charles T. Clingen, and Robert C. Daley. The multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5), 1972.

[7] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'16*. ACM, 2016.

[8] Shamim Bhuiyan, Michael Zheludkov, and Timur Isachenko. *High Performance In-Memory Computing with Apache Ignite*. Lulu.com, 2017. ISBN 1365732355.

[9] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'11*, 2011.

[10] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Proceedings of the Adaptable and extensible component systems*, 2002.

[11] Cassandra-pmem. https://github.com/intel/cassandra-pmem/tree/13981.

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'11*. ACM, 2011.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud Computing, SoCC'1*. ACM, 2010.

[14] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA'18*. ACM, 2018.

[15] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'20*. ACM, 2020.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'7*. ACM, 2007.

[17] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the International Symposium on Memory Management04*, page 37–48. ACM, 2004.

[18] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11), 1978.

[19] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21:1793–1807, 12 2010.

[20] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the international conference on Virtual Execution Environments, VEE'09*. ACM, 2009.

[21] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'18*. ACM, 2018.

[22] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *Proceedings of the USENIX Annual Technical Conference, ATC'20*, 2020.

[23] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20*. ACM, 2020.

[24] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the international conference on Distributed Computing, DISC'16*. Springer, 2016.

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[26] Goslingn James, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java™ language specification*, chapter 17.4. Java SE 14 edition, 2020.

[27] JEP 352: Non-Volatile Mapped Byte Buffers. URL https://openjdk.java.net/jeps/352.

[28] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[29] Mick J. Jordan and Malcolm P. Atkinson. Orthogonal persistence for java? - a mid-term report. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*, page 335–352, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558605851.

[30] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'19*. ACM, 2019.

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'17*, page 460–477. ACM, 2017.

[32] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[33] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'19*. ACM, 2019.

[34] Linux. Direct access for files. URL https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[35] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, 1996.

[36] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3), 1988.

[37] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'21*. ACM, 2021.

[38] Low Level Persistence Library. URL https://github.com/pmem/llpl.

[39] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment*, 13(8), 2020.

[40] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the symposium on Principles Of Programming Languages, POPL'05*. ACM, 2005.

[41] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[42] Dawn May. IBM I single level store . . . In Lieu of a Crystal Ball, 2013.

[43] Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, and Yoav Zuriel. Oak: A scalable off-heap allocated key-value map. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'08*. ACM, 2020.

[44] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20*. ACM, 2020.

[45] Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeff Markham. *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 1st edition, 2014. ISBN 0321934504.

[46] Oracle. *Java Native Interface Specification*. Java SE 14 edition, 2020.

[47] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, 1972.

[48] Persistent Collections for Java. URL https://github.com/pmem/pcj.

[49] Persistent Memory Development Kit, 2018. URL https://pmem.io/pmdk.

[50] Persistent Memory Storage Engine for MongoDB. https://github.com/pmem/pmse.

[51] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI'10*. ACM, 2010.

[52] Pmem-Redis. https://github.com/pmem/pmem-redis.

[53] pmem rocksdb. https://github.com/pmem/pmem-rocksdb.

[54] pmemkv. https://github.com/pmem/pmemkv.

[55] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-X86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2019.

[56] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.

[57] Redis. https://redis.io/, 2009.

[58] Haytham Salhi, Feras Odeh, Rabee Nasser, and Adel Taweel. Open source in-memory data grid systems: Benchmarking hazelcast and infinispan. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 163–164, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344043. doi: 10.1145/3030207.3053671. URL https://doi.org/10.1145/3030207.3053671.

[59] Thomas Shull, Jian Huang, and Josep Torrellas. AutoPersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI'19*. ACM, 2019.

[60] Thomas Shull, Jian Huang, and Josep Torrellas. QuickCheck: Using speculation to reduce the overhead of checks in nvm frameworks. In *Proceedings of the international conference on Virtual Execution Environments, VEE'19*. ACM, 2019.

[61] TPC Benchmark B. URL http://www.tpc.org/tpcb.

[62] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'11*. ACM, 2011.

[63] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 347–362, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314650. URL https://doi.org/10.1145/3314221.3314650.

[64] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384777. URL https://doi.org/10.1145/2384716.2384777.

[65] Michael Wu and Willy Zwaenepoel. ENVy: A non-volatile, main memory storage system. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'94*. ACM, 1994.

[66] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'18*. ACM, 2018.

[67] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'17*. ACM, 2017.

[68] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11), 2016.

[69] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, OSDI'20*, 2020.

[70] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.