Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory

Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University

Abstract

Fast, byte-addressable non-volatile memory (NVM) embraces both near-DRAM latency and disk-like persistence, which has generated considerable interests to revolutionize system software stack and programming models. However, it is less understood how NVM can be combined with managed runtime like Java virtual machine (JVM) to ease persistence management. This paper proposes *Espresso*[1], a holistic extension to Java and its runtime, to enable Java programmers to exploit NVM for persistence management with high performance. Espresso first provides a general persistent heap design called Persistent Java Heap (PJH) to manage persistent data as normal Java objects. The heap is then strengthened with a recoverable mechanism to provide crash consistency for heap metadata. *Espresso* further provides a new abstraction called Persistent Java Object (PJO) to provide an easy-to-use but safe persistence programming model for programmers to persist application data. Evaluation confirms that *Espresso* significantly outperforms state-of-art NVM support for Java (i.e., JPA and PCJ) while being compatible to data structures in existing Java programs.

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

Keywords Non-Volatile Memory, Crash Consistency, Java Virtual Machine

ACM Reference Format:
Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of ASPLOS '18: Architectural Support for Programming Languages and Operating Systems (ASPLOS '18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173201

1 Introduction

Due to promising features like non-volatility, byte-addressability and close-to-DRAM speed, emerging non-volatile memories (NVM) are projected to revolutionize the memory hierarchy in the near future. In fact, battery-backed non-volatile DIMM (NVDIMM) [30] has been available to the market for years. With the official release of Intel and Micron's 3D-Xpoint [17] to the market, it is foreseeable to see NVM to be widely deployed soon.

While there have been considerable interests to leverage NVM to boost the performance and ease the persistence management for native code [8, 14, 16, 27, 29, 32, 38, 42], how NVM can be exploited by high-level programming languages with managed runtime like Java is less understood. Despite their attracting features such as automatic memory management, portability and productivity, the additional layer of abstraction brought by the language virtual machine (e.g., JVM) complicates the persistence management.

The mainstream persistent programming model leverages a coarse-grained abstraction like Java Persistence API (JPA) [9] to provide easy-to-use transactional APIs for programmers to persist their data. However, it does not consider the emergence of NVM and creates unnecessary transformation overhead between Java objects and native serialized data. In contrast, the recent proposed Persistent Collections for Java (PCJ) [15] provides a fine-grained programming model to enable users to manipulate persistent data at the object level. However, it provides an independent type system against the original one in Java, which is incompatible with existing Java programs since it mandates the use of the *collections* defined by itself. Furthermore, PCJ manages persistent data as native objects on their own, which ends up

[1] Espresso coffee contains more non-volatile chemicals (such as caffeine); we use it as an analog to our work where data becomes more non-volatile

This work is supported in part by China National Natural Science Foundation (No. 61672345) and National Key Research & Development Program of China (No. 2016YFB1000104). Corresponding author: Haibo Chen (haibochen@sjtu.edu.cn).

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173201

with poor performance[2]. Besides, these two approaches target different application scenarios and programmers cannot uniformly use one approach to applications that have both requirements.

This paper proposes *Espresso*, a unified persistence framework that supports both fine-grained and coarse-grained persistence management while being mostly compatible with data structures in existing Java programs and notably boost the performance in persistence management. *Espresso* provides *Persistent Java Heap (PJH)*, an NVM-based heap to seamlessly store persistent Java objects. PJH allows users to manipulate persistent objects as if they were stored in a normal Java heap and thus requires no data structure changes. To allocate data on PJH, *Espresso* provides a lightweight keyword *pnew* to create Java objects in NVM.

PJH serves as an NVM-aware allocator, which should tolerate machine crashes to create a safe runtime environment for programmers. Hence, PJH provides crash-consistent allocation and deallocation (garbage collection), which guarantee that the metadata of the heap is crash-consistent.

To further ease the programming for applications that require coarse-grained persistence, *Espresso* provides Persistent Java Object (PJO), a new persistent programming abstraction atop PJH as a replacement of JPA for NVM. PJO provides backward-compatibility by reusing the annotations and transactional APIs in JPA, yet with additional optimizations to eliminate unnecessary overhead in the original JPA to boost the performance.

We have implemented the design of PJH and PJO atop OpenJDK 8. To confirm the effectiveness of our design, we provide a set of evaluation against JPA and PCJ. The result indicates that *Espresso* achieves up to 256.3x speedup compared with PCJ for a set of microbenchmarks. Furthermore, PJO can provide support for different kind of data types in the JPAB benchmark, which gains up to 3.24x speedup over the original JPA for the H2 database.

In summary, this paper makes the following contributions:

- A persistent Java heap design (PJH) that enables Java programs to exploit NVM for persistence management without massive reengineering.
- A new abstraction for persistent programming (PJO) for simple and safe manipulation on persistent data objects.
- An implementation of PJH and PJO atop OpenJDK and a set of evaluations that confirm its effectiveness.

The rest of our paper is organized as follows. Section 2 reviews two main approaches for persistence management and discusses its deficiencies, which motivates the design of *Espresso*. Section 3 introduces an overview of our PJH and language extension to manipulate persistent data objects. Section 4 further describes our mechanism to guarantee the crash consistency of PJH. Section 5 presents the abstraction PJO together with an easy-to-use persistent programming model for programmers who require safe ACID semantics. We evaluate our design in section 6, discuss related work in section 7 and finally conclude in section 8.

[2]The home page (https://github.com/pmem/pcj) of PCJ acknowledged that "The breadth of persistent types is currently limited and the code is not performance-optimized".

2 Background and Motivation

In this section, we briefly review two main approaches to persistence management in Java, which provide coarse-grained and fine-grained persistence accordingly. We show that both approaches have some deficiencies in providing a compatible and efficient way for persistence with respect to NVM.

2.1 Coarse-grained Persistence with JPA

Database is a very appealing application for NVM and has been intensively studied by prior work [32, 40, 42, 45]. Many databases [2] are written in Java due to the portability and easy programming. For ease of persistent programming, such databases usually provide a *persistent layer* to keep programmers away from the messy work on persistent data management. This layer can be implemented according to Java official specification (such as JDO [18] and JPA [9]) or a customized one for different use cases. Overall, it mainly serves data transformation between Java runtime and persistent data storage. We choose JPA as an example in this paper since it is commonly used and more recent than JDO.

JPA (also known as Java Persistent API) is a specification officially offered by the Java community for persistence programming, especially in relational database management system (RDBMS). With JPA, programmers are allowed to declare their own classes, sub-classes and even collections with some annotations. JPA is responsible for data transformation between Java applications and RDBMSes: it serves applications with objects while it communicates with RDBMSes via the Java Database Connectivity (JDBC) interface. JPA also provides the abstraction of ACID transactions for programmers, which guarantees that all updates related to persistent data will be persisted after a transaction commits.

To understand the performance of JPA atop NVM, we present a case study with DataNucleus [1], a widely-used open-source implementation of JPA. The architecture is illustrated in figure 1.

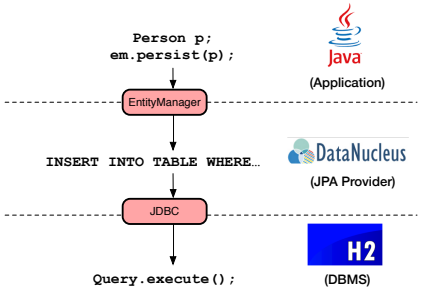


Figure 1. The infrastructure of DataNucleus

DataNucleus requires all classes related to persistent data to implement the *Persistable* interface. Programmers should mark their classes with the annotation *@persistable*. Suppose a programmer wants to declare a simple class *Person* which contains two fields: *id* (Integer) and *name* (String), she should write code similar to that shown in figure 2. Note that we will use the *Person* class throughout the paper as a running example. DataNucleus has provided a bytecode instrumentor named *enhancer* to transparently transform arbitrary classes annotated with *@persistable* into those with *Persistable* interface implemented. Afterwards, the implementation of APIs required by *Persistable* interface will also be automatically generated by the *enhancer*. The enhancer will also insert some *control fields* (corresponding to *data fields* that store user data) into *Persistable* objects and instrument auxiliary methods (*getId* in this example) for ease of management.

```
1  @persistable
2  public class Person {
3      // fields
4      private Integer id;
5      private String name;
6
7      // constructor
8      public Person(Integer id, String name) {
9          this.id = id;
10         this.name = name;
11     }
12
13     // an auxiliary method example
14     public Integer getId() {
15         return this.id;
16     }
17
18     ......
19 }
```

Figure 2. The declaration for class *Person* under JPA

Unfortunately, the data layout in RDBMSes is not compatible with that in Java objects. To persist user data, DataNucleus needs to initiate a transformation phase to translate all updates on them into SQL statements. It subsequently sends the statements to the backend RDBMSes through JDBC to update data in the persistent storage. Note that only SQL statements are conveyed to DBMSes, so even RDBMSes written in pure Java (like H2 [31]) in figure 1 can only update databases with SQL instead of real data stored in objects.

Deficiencies of JPA on NVM. The transformation phase in DataNucleus (JPA) induces significant overhead in overall execution. We test its *retrieve* operation using the JPA Performance Benchmark (JPAB) [34]. Figure 3 illustrates a breakdown of performance. The result indicates that the user-oriented operations on the database only account for 24.0%. In contrast, the transformation from objects to SQL statements takes 41.9%. This indicates that the JPA incurs notable performance overhead.

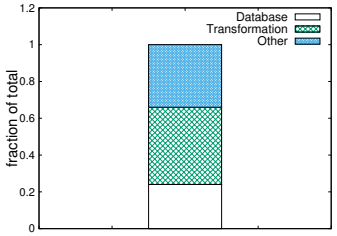


Figure 3. Breakdown for commit phase of DataNucleus

2.2 Fine-grained Persistence with PCJ

Persistent Collections for Java (PCJ [15]) by Intel is an active project designed for Java programmers to store their data in NVM. However, our study shows that PCJ has several deficiencies due to its design. Other libraries like MDS [13] also allows Java users to manipulate data within NVM with lightweight Java bindings. However, they are mostly designed for C/C++ programs, and they share similar problems with PCJ, as shown below.

Separated type system. PCJ implements a new type system based on a persistent type called *PersistentObject*, and only objects whose type is a subtype of *PersistentObject* can be stored in NVM. Users who want to use PCJ must extend *PersistentObject* to implement their own types. Figure 4 illustrates the declaration of the *Person* class on PCJ[3]. The class *Person* must first extend *PersistentObject* to fit PCJ. Furthermore, the type of *id* and *name* should be modified into *PersistentInteger* and *PersistentString* respectively, both of which are subtypes of *PersistentObject*. Hence, using PCJ mandates a non-trivial reengineering to transform existing data structures to a form supported by PCJ.

```
1  public class Person extends PersistentObject {
2      // fields
3      private PersistentInteger id;
4      private PersistentString name;
5
6      // constructor
7      public Person(Integer id, String name) {
8          this.id = new PersistentInteger(id.intValue());
9          this.name = new PersistentString(name);
10     }
11
12     // a method example
13     public Integer getId() {
14         return this.id.intValue();
15     }
16
17     ......
18 }
```

Figure 4. The declaration for a simple class *Person* in PCJ

[3]The original declaration of *Person* is much more complex with a bunch of static variables and helper methods. We have simplified the declaration for ease of understanding.

Off-heap data management. Due to the lack of support from Java, PCJ stores persistent data as native off-heap objects and manage them with the help of NVML [16], a C library providing ACID semantics for accessing data in NVM. Therefore, PCJ has to define a special layout for native objects and handle synchronization and garbage collection all by itself. This may lead to non-trivial management overhead and suboptimal performance. We have implemented a simple example where we create 200,000 *PersistentLong* objects (the equivalent of Java *Long* object in PCJ) and analyzed its performance in figure 5.

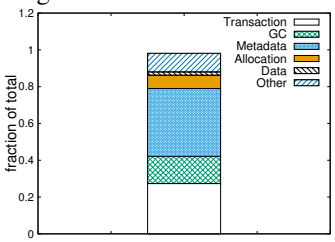


Figure 5. Breakdown analysis for create operations in PCJ

First, the operation related to real data manipulation only accounts for 1.8% over the whole execution time. In contrast, operations related to metadata update contributes 36.8%, most of which is caused by type information memorization. In a normal Java heap, the type information operation only contains a reference store, which is much simpler.

Furthermore, it takes 14.8% of the overall time to add garbage collection related information to the newly created object. PCJ needs this step because it is based on a reference counting GC algorithm, which needs to update GC-related information for each initialization. A normal Java heap leverages more mature garbage collectors and takes less time to bookkeep objects.

The last source of overhead comes from transactions, which mainly contain synchronization primitives and logging. This phase can also be optimized with the reserved bits in object headers and transaction libraries written in Java, if the objects are managed within Java heap.

In summary, most overhead in PCJ is caused by its off-heap design, which could be notably optimized with an on-heap design.

2.3 Requirements for Persistence Management in Java

From our study, we can see that there is currently no unified framework to provide persistence in Java. JPA is mostly useful for databases that require coarse-grained persistence, while PCJ mandates the use of its defined collections in order to enjoy fine-grained persistence, which would incur non-trivial porting efforts due to a shift of data structures. Besides, both solutions suffer from notable performance overhead and thus cannot fully exploit the performance benefit of NVM.

In light of this, we believe an ideal persistent framework for Java should satisfy the following requirements.

- *Unified persistence:* The framework should support both fine-grained and coarse-grained persistence so as to support a wide range of applications.
- *High performance:* The framework should incur only a small amount of overhead for persistence to harness the performance advantage of NVM.
- *Backward compatibility:* The framework should not require major database changes so that existing applications can be ported with small effort to run atop it.

3 Persistent Java Heap

Being aware of above requirements, *Espresso* provides a unified framework for Java to support both fine-grained and coarse-grained persistence management. It mainly contains two parts: Persistent Java Heap (PJH) to manage persistent objects in a fine-grained way, while Persistent Java Object (PJO) helps programmers to manage persistent data with handy interfaces. This section will mainly describe the design of PJH.

3.1 Overview

PJH is an extension to the original Java heap by adding an additional persistent heap. We have built PJH in the *Parallel Scavenge Heap* (PSHeap), the default heap implementation for OpenJDK. Figure 6 illustrates the modified layout of PSHeap with PJH.

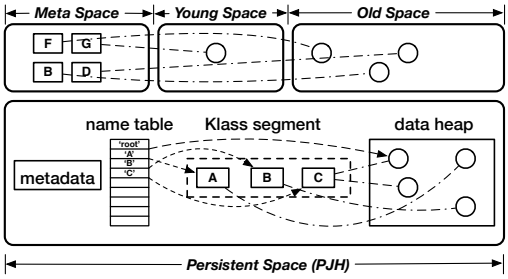


Figure 6. The Java heap layout with PJH

The original implementation of PSHeap contains two different spaces to store objects (circles in figure 6): Young Space and Old Space. Objects will be initially created at the Young Space and later promoted to the Old Space if they have survived several collections. The garbage collector, namely Parallel Scavenge Garbage Collector (PSGC), also provides two different garbage collection algorithms. Young GC only collects the garbage within the Young Space, which happens frequently and finishes soon. In contrast, Old GC collects the whole heap, which takes more time and happens infrequently.

In Java, each object should hold a class pointer to its class-related metadata, which is called a *Klass* in OpenJDK (rectangles with capital letters in figure 6). The class pointer is stored in the header of an object, right next to the real data fields (dashed lines in figure 6). JVM has maintained a *Meta*

Space to manage the Klasses. Klasses are very important because they store the layout information for objects. If the class pointer in an object is corrupted, or the metadata in Klass is lost, data within the object will become uninterpretable.

PJH is implemented as an independent *Persistent Space* against the original PSHeap. It is designed as a non-generational heap since we believe that NVM will be mostly used to store long-lived data due to its persistence guarantee and inferior performance compared to DRAM as reported by previous work [21, 44]. The garbage collection algorithm for PJH resembles the old GC in PSGC in that it is designed for long-lived objects and infrequent collections. The main components of PJH include metadata area, name table, Klass segment and data heap. All the components should be persisted in NVM to guarantee the crash-consistency of the PJH.

Data heap and Klass segment. Java objects required to be persisted are stored in the *data heap*. The object header layout is the same as that in the normal Java heap; so each persistent object still holds a class pointer to its class-related metadata to its Klass. All Klasses used by persistent objects are stored in the *Klass segment* and managed separately from the original Meta Space.

Name table. The name table stores mappings from string constants to two different kinds of entries: Klass entries and root entries. A Klass entry stores the start address of a Klass in the Klass segment, which is set by JVM when an object is created in NVM while its Klass does not exist in the Klass segment. A root entry stores the address of a *root object*, which should be set and managed by users. Root objects are essential especially after a system reboot, since they are the only known entry points to access objects in data heap.

Metadata area. The metadata area shown in figure 7 is kept for memorizing heap-related metadata to build a reusable and crash-consistent heap. The *address hint* stores the starting virtual address of the whole heap for future heap reloading, while the *heap size* stores the maximum address space the PJH can occupy. The *top address* can be used to calculate the allocated bytes of PJH. Other information is essential to implement a recoverable garbage collector for PJH and will be discussed in detail later.

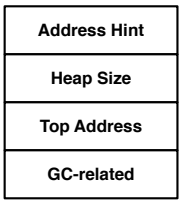


Figure 7. The components for the metadata area in PJH

3.2 Language Extension: pnew

To allow users to create objects on NVM, we add a keyword *pnew* to the Java programming language. The keyword has similar syntax rules to *new* except that the corresponding objects will be laid on NVM. We have modified Javac to convert *pnew* into four different bytecodes accounting for different syntaxes: *pnew* (normal instances), *panewarray* (object arrays), *pnewarray* (primitive arrays) and *pmultianewarray* (multi-dimensional arrays). Those bytecodes will put objects into PJH regardless of their types. Note that the keyword *pnew* only allocates an object on NVM without considering its fields. If users want to make certain fields of an object persistent, they may need to implement a new constructor with *pnew*.

The keyword *pnew* enables programmers to tackle with NVM in a very familiar way. Figure 8 shows how to define the class *Person* in section 2. Since it does not impose restrictions on which type can be persisted, the class *Person* does not need to be extended from any particular types, nor do its fields need to be changed. The resulting code is very similar to that written for original Java except for the *pnew* keyword and the newly added constructor for *String*, so it is easy to understand.

Note that the *pnew* keywords in the constructor can be freely replaced with *new*, as we do not force the invariants for references at the language level. Users are allowed to define references to volatile memory to support applications using a mix of NVM and DRAM, as required by prior work [21, 44].

```
1  public class Person {
2      // fields
3      private Integer id;
4      private String name;
5
6      // constructor
7      public Person(Integer id, String name) {
8          this.id = pnew Integer(id);
9          this.name = pnew String(name, true);
10     }
11
12     // a method example
13     public Integer getId() {
14         return this.id;
15     }
16
17     ......
18 }
```

Figure 8. The declaration for class *Person* atop PJH

Alias Klasses: Our design allows objects of the same type to be stored in both DRAM and NVM, which violates the assumption of original Java runtime. In Java, each Klass will contain a data structure called *constant pool* [24]. Constant pools store important symbols which will be resolved during runtime. For each class symbol, a constant pool will initially create a slot and store a reference to its name (a string constant). If the symbol is resolved, the slot will instead store the address of the corresponding Klass.

```
1  Person a = new Person(...);
2  Person b = pnew Person(...);
3  somefunc((Person)a);
4  // ClassCastException here!
```

Figure 9. A simple program encountering wrong exception when using *pnew*

This design works perfectly in the stock JVM, but it induces some problem in PJH. Consider the code in figure 9 where we subsequently create two objects a and b of type *Person* with *new* and *pnew* respectively. Afterwards, code in line 3 tries to cast the object type into *Person*, which should have been a redundant type casting operation. Nevertheless, the program ends up with a *ClassCastException*.

The problem happens because *Person* objects are stored in both volatile and non-volatile memory, resulting in two different Klasses. Meanwhile, the constant pool keeps only one slot for each class symbol. In the example, JVM will find that the object is volatile and allocate the corresponding Klass for *Person* (denoted as K_p) in DRAM when creating a. Afterwards, JVM soon realizes that object b should be persistent, so it also creates a Klass for *Person* again in the Klass Segment in PJH (K'_p). Since the addresses for two Klasses differ, the constant pool has to store the address of K'_p to replace that of K_p. On type casting, JVM finds that the resolved class in its constant pool (K'_p) is at odds with the type of a (K_p), so it throws an exception.

We introduce a concept named *alias Klass* to handle this problem. Two Klasses are an alias to each other if they are logically the same class but stored in different places (NVM and DRAM). It is implemented by introducing a new field in all Klasses referring to its alias Klass. Aliases will share metadata like static members and methods to ensure correctness. We add the alias check into type checking within JVM to avoid wrong exceptions. We have also extended the type lattice in the OpenJDK Server Compiler [36] to consider aliases during JIT optimizations.

Original type-related checks like subtype check and classloader check [23] are also extended. Taking subtype check as an example. Suppose we want to check the subtype relationship between class A and B, and they both have an alias (A' and B' respectively), then we should consider the subtype relationship for four pairs of classes: A and B, A' and B, A and B', A' and B'. The check will be passed if any of those pairs is proved to have a subtype relationship. The algorithm seems more complicated than before, but in most cases where both A and B have no alias, it runs exactly the same as the original one.

3.3 Heap Management

In our programming model, users are allowed to create multiple PJH instances served for various applications. They are also required to define root objects as handles to access the persistent objects even after a system reboot. We have implemented some basic APIs (shown in table 1) in Java standard library (JDK) to help them manage the heap instances and root objects. Those APIs can be classified into two groups: *createHeap*, *loadHeap* and *existsHeap* are heap-related while *setRoot* and *getRoot* are root-related. Figure 10 shows a simple example where we want to locate some data in a heap or initialize the heap if it does not exist.

API	Args	Description
createHeap	name, size	create a PJH instance
loadHeap	name	load a PJH instance into current JVM
existsHeap	name	check if a PJH instance exists
setRoot	name, object	mark an object as a root
getRoot	name	fetch a root object

Table 1. APIs for PJH management

```
1  // Check if the heap exists
2  if (existsHeap("Jimmy")) {
3      // If so, load the heap and fetch objects
4      loadHeap("Jimmy");
5      Person p = (Person) getRoot("Jimmy_info");
6  } else {
7      // Otherwise, create a new heap and objects
8      long size = 1024 * 1024;
9      createHeap("Jimmy", size);
10     Person p = pnew Person(...);
11     setRoot("Jimmy_info", p);
12 }
```

Figure 10. A simple example using heap management APIs

Heap-related APIs. Java programmers can invoke *createHeap* (line 9) to create a PJH instance with specified name and size (in bytes). We have implemented an external name manager responsible for the mapping between the name and real data of a PJH instance. *createHeap* will notify the name manager to insert a new mapping into the table. Furthermore, the starting (virtual) address should also be stored as *address hint* in the metadata area of the PJH instance for future use. Afterwards, users can use *pnew* to allocate objects on the newly created heap (line 10).

Users are allowed to load pre-existing PJH instances into current JVM by invoking *loadHeap*. They can optionally call *existsHeap* in advance (line 2) to check if a PJH instance has already existed. When *loadHeap* is finally invoked at line 4, the external name manager will locate the PJH instance and return its starting address by fetching the address hint. Afterwards, JVM will map the whole PJH at the starting address. If the map phase fails due to the address occupied by the normal heap, we have to move the whole PJH into another virtual address. Since all the pointers within heap become trash, a thorough scan is warranted to update pointers. The remap phase might be very costly, but it may rarely happen thanks to the large virtual address space of 64-bit OSes. If the map operation succeeds, it will be followed by a class reinitialization phase.

The stock JVM will allocate a new Klass data structure in its Meta Space for each class initialization. However, if we

bluntly create new Klasses in the Klass segment during class reinitialization, all class pointers in PJH will become trash, which is unacceptable. To avoid invalidating class pointers, we require that all Klasses in PJH stand for a place holder and be initialized in place. In this way, all objects and class pointers will become available after class reinitialization. Our design makes the load phase of PJH very fast because the time overhead is directly proportional to the number of Klasses instead of objects. Meanwhile, the number of Klasses in the Klass Segment is usually trivial. For example, a typical TPCC [41] workload only requires nine different data classes to be persisted. After class reinitialization, *loadHeap* will return and users are free to access the persistent data in the loaded PJH instance.

Root-related APIs. *Root objects* marks some known locations of persistent objects and can be used as entry points to access PJH especially when a PJH instance is reloaded. *getRoot* and *setRoot* serve as getter/setter for the root objects. When *getRoot* is called at line 5, the corresponding object *p* will be returned. Since we don't store the type of the root object, the return type will be *Object*, and users are responsible for type casting. After that, users can fetch other persistent data by accessing *p*. Similarly, *setRoot* at line 11 receives an object in arbitrary type and stores its address in the root table with the specified name for future use.

3.4 Referential Integrity

The design of PJH has decoupled the persistence between an object and its fields: an object can be stored in NVM with a reference to DRAM. We allow those Non-Volatile-to-Volatile (NV-to-V) pointers to reflect the fact that not all fields of a data structure need to be persisted. Programmers should be permitted to lay those fields, such as data cache and locks, in volatile memory for the sake of performance. Unfortunately, this design violates *referential integrity* [8] in Java, which guarantees all references point to valid data after crash recovery. If users try to access a reference to volatile memory after heap reloading, the reference can point to anywhere and modifications of the referenced data can cause undefined consequences. In contrast, an over-restricted invariant on references can ensure correctness, but makes it difficult to leverage DRAM. To this end, we have provided three different safety levels according to various requirements on usability and safety.

User-guaranteed safety. Users need to be aware of the presence of volatile pointers and avoid directly using them after a reload of PJH. This safety level lays the burden of checking on programmers and may cause unknown errors. However, it provides the best performance compared to others.

Zeroing safety. A PJH instance will first step into a checking phase before loading, and all out pointers will be nullified. In this way, applications can easily tell if they have suffered a Java execution context loss with null-checks. Even the worst case for a careless access on invalid volatile pointers will only get a *NullPointerException*, which is much better than one could experience in user-guarantee safety level. Zeroing safety is enabled by default in our current implementation.

```
1  Person x = pnew Person(...);
2  Person[] z = pnew Person[10];
3  // After some operations...
4  ......
5  Field f = x.getClass.getDeclaredField("id");
6  // Newly added APIs below:
7  f.flush(x);          // for normal fields (flush x.id)
8  Array.flush(z, 3);   // for arrays (flush z[3])
```

Figure 11. A simple program to illustrate our flush APIs

The major disadvantage of zeroing safety is that the checking phase will traverse the whole heap and slow down the heap loading. To mitigate the traversal overhead, PJH can maintain a card table to bookkeep the out pointers in a coarse-grained fashion. Once reloading, only memory areas with out pointers should be scanned. Users can additionally launch a helper Java process to nullify the out pointers in background by invoking *loadHeap* after the PJH is unloaded so as to move the check phase off the critical path.

Type-based safety. For users who really want to access NVM safely, we have implemented a library atop Java to allow them to define classes with simple annotations, and only objects with those classes will be persisted into PJH (introduced in section 5). This safety level guarantees that no pointers within PJH will point out of it, and thus provides a similar safety level to NV-Heaps [8]. However, it requires applications to be modified and annotated to fit NVM.

3.5 Persistence Guarantee

Mainstream computer architectures only have volatile caches and thus require cache flush operations like *clflush* to ensure data persisted in NVM. To preserve persistence ordering, we may further require memory fence instructions (such as *sfence*). The *pnew* keyword is only used for object allocation, so we can only provide persistence guarantee for heap-related metadata to build a recoverable heap regardless of crashes (discussed in section 4) with those instructions. As for the application-level guarantee, we have provided some basic field-level APIs to manage the persistence of objects in a fine-grained way. Figure 11 illustrates an example to leverage our APIs. To persist field *id* in object *x*, we must fetch the incarnation of *id* at runtime with Java Reflect APIs, such as *getDeclaredField*. After that, we can use the newly added *flush* interface to persist *x.id*. If applications want to manipulate arrays, they can use *Array.flush* to flush certain object with offset *i*. The largest work set for those two APIs are restricted to 8 bytes to preserve atomicity. Besides, the implementation of those two APIs should add a fence instruction for ordering guarantee.

Additionally, we have also added a coarse-grained *flush* method in the implementation of *Object* class for performance consideration. This method will flush all the data fields in the object into NVM with only one fence instruction in the end. It is suitable for scenarios where the persistent order among fields of an object doesn't matter. Other advanced features, such as transitively persist all data reachable from an object, can be easily implemented with such basic methods.

Note that the APIs mentioned above are platform-independent and can be adaptively implemented for various architectures. For example, the *flush* calls can be transformed to no-ops if the JVM is running on an architecture equipped with persistent cache. The fence instructions can also be eliminated atop a strict memory model.

4 Crash-consistent Heap

The design of PJH should consider crashes which can happen at any time to avoid inconsistency. To this end, *Espresso* enhances the allocation and garbage collection phase to ensure that the heap can be recovered to a consistent state upon failure.

4.1 Crash-consistent Allocation

The persistent heap maintains a variable named *top*[4] to memorize how much memory resource has been allocated. The value of *top* is replicated in the PJH for future heap reloading. As we mentioned before, users are permitted to exploit *pnew* to create a persistent object, which has an impact on the heap-related metadata. The allocation can be divided into three steps:

- (1) Fetching the Klass pointer from the constant pool;
- (2) Allocating memory and updating the value of *top*;
- (3) Initializing the object header.

Since the Java compiler *Javac* guarantees that an object will not become visible until the header is initialized, *Espresso* does not need to consider inferences with other threads. To make the allocation crash-consistent, the replica of the *top* value in PJH should be persisted as soon as the modification on the volatile one in step (2), e.g., through cache flush and fence instructions. Otherwise, some created objects may be treated as unallocated and truncated during recovery due to the stale top value. Further, the Klass pointer update should be persisted in step (3) to avoid the situation where an initialized object refers to some corrupted Klass metadata.

4.2 Crash-consistent Garbage Collection

Since the life cycles for persistent objects are often long, we reuse the old GC algorithm in PSGC to collect them. However, the original algorithm needs to be carefully enhanced for crash consistency consideration.

[4] *top* is very similar to *brk* in Linux and it is named by the stock JVM.

A Brief review of PSGC. PSGC exploits a three-phase region-based algorithm for its old GC. The whole heap has been divided into many small areas named *regions*. The first marking phase will mark live objects from all roots. PSGC has maintained a read-only bitmap called *mark bitmap* to memorize all live objects in a memory-efficient way.

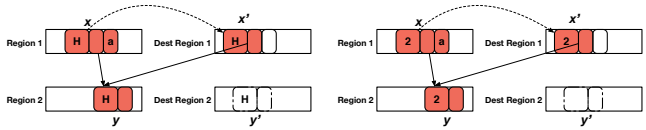
The second phase, namely *summary phase*, will summarize the heap spaces and generate region-based indices to store the destination address of all live objects. After the summary phase, the destination address for each live object is determined. Note that the summary phase is *idempotent*: the indices are derived only from the mark bitmap; so the result of summary phase will be the same no matter how many times it executes, as long as the mark bitmap keeps intact.

In the last *compact phase*, GC threads will pick out unprocessed regions and copy live objects into their destinations. The regions will be processed concurrently, but each region will only be processed by one unique worker thread. For each object, a GC thread will first get its destination address by querying the region-based indices and copy its content there. Afterwards, it will move to the copied object at the destination address, look into all references within it, and correct them respectively with the help of indices.

Crash-consistent GC. An important feature of PS old GC is that the heap state remains inconsistent throughout the compact phase. Therefore, a reasonable method to recover upon a crash is to continue GC from the crash point to the end of compaction where the whole heap becomes consistent again. We have enhanced the PS old GC with this method and proposed a crash-consistent garbage collector. It is mainly built on two components: a snapshot taken before compact phase to guarantee a collection can finally complete regardless of crashes, and a timestamp-based algorithm to infer and recover from the crash state.

Before the compact phase, *Espresso* will take a consistent snapshot over the whole heap and persist it into NVM. It is achieved by persisting the *mark bitmap* generated by the mark phase; *Espresso* has extended PSGC to also mark live objects in the persistent space. Since the summary phase is deterministic and idempotent, the snapshot is enough to recalculate region-based indices and thereby determine destination addresses for all live objects. If a crash happens, *Espresso* can guarantee that the destination address is still available for any object, by reloading the snapshot from NVM. After taking the snapshot, *Espresso* will mark the whole heap as being garbage collected in the PJH metadata area so that a crash will trigger the recovery.

Nevertheless, the consistent snapshot is not enough for recovery. Consider the case shown in figure 12a where a live object x is being copied when a crash happens. The object x has two fields: a reference to another object y and a data field a storing an integer. x is half-copied in this case where *Espresso* fails to copy the data field before the crash. Since



(a) without timestamps **(b)** with timestamps

Figure 12. A case in GC which is only recoverable with timestamps

the snapshot only provides information about destination addresses, it is impossible for *Espresso* to find out the data field has not been copied, and the user data will be lost. The problem happens because *Espresso* don't know the crash state of live objects. If *Espresso* is aware that an object is copied, not copied, or half-copied, it can provide solutions for those scenarios respectively.

To this end, we have proposed a timestamp-based algorithm to infer the crash state of objects. It is implemented by reusing metadata bits from the Java object header. Those bits are only kept for *young space* and become useless once objects are copied out, so it is safe to reuse them. With those bits, *Espresso* will install a local timestamp in each object header, and maintain a global timestamp in PJH metadata area. Initially, the local timestamp should conform to the global one; but *Espresso* will increase the global timestamp before the compact phase starts, making all objects stale. The local timestamp of an object will not be synchronized to the global one until its whole content has been copied, updated and persisted. If a crash happens, *Espresso* can tell whether an object has been processed by simply inspecting the local timestamp. If the timestamp is stale, *Espresso* will copy and update references for the object with the help of the consistent snapshot. Combining timestamps with the snapshot, *Espresso* is capable of recovering from crash and continue the collection till its end.

Figure 12b illustrates how *Espresso* handles the previously mentioned case at presence of timestamps. Suppose the current global timestamp is 3, which invalidates the local one in x whose value is 2. Since the copy phase has not finished, the timestamp in x remains unchanged. If a crash occurs when x is half-copied, *Espresso* will find the local timestamp stale and redo the copy phase so that the inconsistent issue is solved.

4.3 Recovery

The recovery phase will be activated by the API *loadHeap* if the heap is marked as being garbage collected in the metadata area. The recovery mainly contains three steps. The first step is to reload the consistent snapshot (i.e. the mark bitmap) for future use. Afterwards, the summary phase should be redone by regenerating the region-based indices for destination address calculation. The last step is to find out all stale objects throughout the regions by reading the timestamps and process them respectively. Additionally, Invalid references will be nullified during the last step if the zeroing safety level is chosen. After recovery, *loadHeap* will return and the whole heap can be safely used by applications.

5 Persistent Java Object

PJH only guarantees crash consistency for heap-related metadata; application data may still be corrupted upon a crash. Providing high-level guarantee like ACID in language level would be very challenging: due to the semantic gap between Java programs and JVM, it is troublesome and inefficient for Java runtime to manage log space for programs as prior work [6, 8] does. On the contrary, while a *persistent layer* atop Java like JPA is helpful to provide a convenient persistence programming model, it incurs high overhead upon NVM. To this end, *Espresso* builds persistent Java object (PJO) atop the persistent Java heap (PJH) as an alternative for persistence programming.

PJH already allows applications like databases to store their data in NVM as normal Java objects. This offers opportunities to rethink about the persistent layer. PJO provides backward compatibility through reusing the interfaces and annotations provided by JPA. Yet, it reaps the benefits offered by PJH with better performance.

Figure 14 illustrates the modified architecture of database frameworks with PJO. The programmer can still use *em.persist(p)* to persist a *Person* object into NVM. However, when real persistent work begins, data in p will be directly shipped to the backend database. The PJO provider still helps manage the persistent objects, but the SQL transformation phase is removed.

Figure 13 provides a running example to persist user data in ACID fashion with PJO. The code should be wrapped with transaction-related instructions (line 2 and line 6), whose semantics are similar to the atomic blocks in NVHeaps [8] and Mnemosyne [42]. Since PJO hides the complexity of persistent data management from users, they can directly use *new* to create objects, but invoke *em.persist* to inform PJO that the object should be persisted. When a transaction is going to commit, PJO will locate all objects required to be persisted and convey them to the backend database.

```
1  // Start a transaction
2  em.getTransaction().begin();
3  Person p = new Person(...);
4  em.persist(p);
5  // Transaction commits
6  em.getTransaction().commit();
```

Figure 13. Programming in PJO with ACID semantic

Espresso provides a new lightweight abstraction called *DBPersistable* to support all objects actually stored in NVM. The *DBPerson* class in figure 14 is an example of *DBPersistable*. A *DBPersistable* object resembles the *Persistable* one except that the control fields related to PJO providers are stripped.

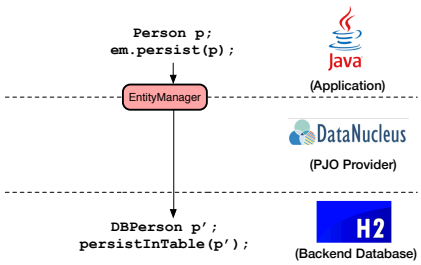
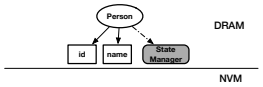


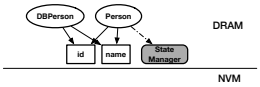
Figure 14. NVM-aware infrastructure of DataNucleus

Figure 15 shows how PJO exactly works for a persist operation on a *Person* object, whose data fields (*id* and *name*) are referenced by solid lines. The PJO provider (our modified *DataNucleus*) will enhance *Person* so that each object keeps a field named *StateManager* for metadata management and access control (referenced by a dash line). The *StateManager* field is transparent to applications. When persisting, a corresponding *DBPerson* object will be generated with all its data fields referenced to the *Person* object (figure 15b). The *DBPerson* object will be shipped to the backend database for data persistence. Now the database can directly persist it into NVM as illustrated in figure 15c.

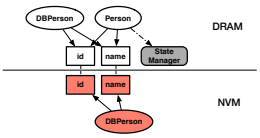
Once the data objects are persisted, the volatile copy left in DRAM becomes redundant. We have implemented a data deduplication optimization such that the data fields of objects will be redirected to the persistent data after a transaction commits. As illustrated in figure 15d, all data fields in the original *Person* object has been modified to point to the persisted data. Consequently, pervious volatile fields can be reclaimed or reused to save memory resource.



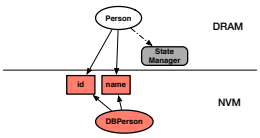
(a) An enhanced *Person* object stored in DRAM



(b) Creating a *DBPerson* object during persisting



(c) Persisting the *DBPerson* object



(d) Data deduplication for the original *Person* object

Figure 15. A detailed example to show how PJO exactly works

We have implemented a PJO provider by modifying *DataNucleus*. It provides the same APIs as JPA does such that no modification to applications is required. Programmers can leverage the APIs provided by the PJO provider to retrieve, process and update data in an object-oriented fashion. Similarly to JPA, PJO also supports various types such as inherited classes, collections and foreign-key-like references. The performance results for different types will be illustrated in section 6.

6 Evaluation

6.1 Experiment setup

We have implemented PJH on OpenJDK 8u102-b14, which comprises approximately 7,000 lines of C++ code and 300 lines of Java code. We have also modified DataNucleus to implement PJO with 1,500 lines of Java code. As for the backend database H2, it takes about 600 LoC to make it support both PJO (mainly for the *DBPersistable* interface) and PJH (mainly replacing *new* with *pnew*). The data structures for transaction control (like logging) remain intact. The modification is minor considering the whole code base of H2 (about 14K LoC). In contrast, a design like PCJ would require a thorough rewrite over the main data structure in H2 (an MVCC-based B+-tree) to fit NVM, which entails an estimate of 3K LoC.

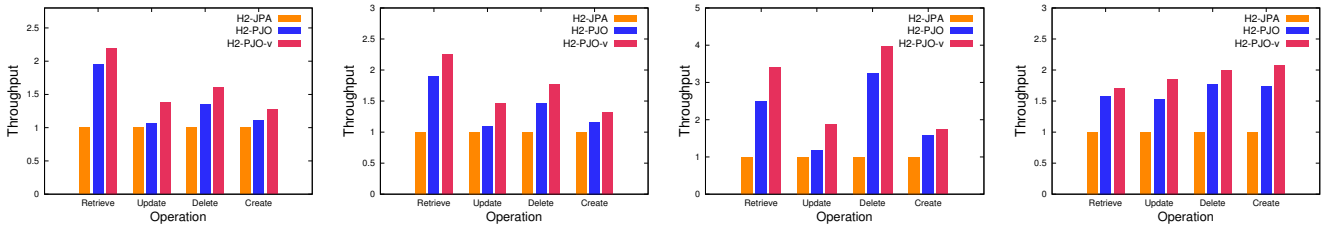
Our evaluation is conducted on a machine with dual Intel ®Xeon™ E5-2618L v3 CPUs (16 cores). It contains 64G DRAM and 64G Viking NVDIMM device. The operating system is Linux-4.9.6. We set the maximum Java heap size to 8G for evaluation. All baseline in this section are running on the stock JVM.

6.2 Comparison with PCJ

PCJ provides an independent type system against the original one in Java including tuples, generic arrays and hashmaps. We also implement similar data structures atop our PJH. Since PCJ provides ACID semantics for all operations, we also add ACID guarantee by providing a simple undo log to make a fair comparison. The microbenchmarks conduct millions of primitive operations (create/get/set) on those data types and then collect the execution time. The results are shown in figure 17.

Our PJH greatly outperforms against PCJ, and the speedup ranges from 6.0x to 256.3x. PJH performs much better in *set* and *create* operations in that PCJ stores data off heap and thus require a complicated metadata update for those operations. As for *get* operations, the improvement of PCJ drops due to less requirement for metadata management, but it still outperforms PCJ by at least 6.0x.

We have further broken down the evaluation to show where the performance improvement comes from by taking the set operation of *Tuple* as an example. After manually removing the GC management code in PCJ, the speedup drops from 256.3x to 102x. If we continue removing transaction-related invocations in PCJ, the performance will be on par with PJH. The result suggests that overhead in PCJ mainly comes from GC and transaction-related operations, which can be greatly mitigated in an on-heap design like PJH. The metadata updates mentioned in section 2.2 is trivial in this case, since type information memorization only happens when an PCJ object is created.



(a) BasicTest **(b)** ExtTest **(c)** CollectionTest **(d)** NodeTest

Figure 16. Evaluation for JPAB benchmark

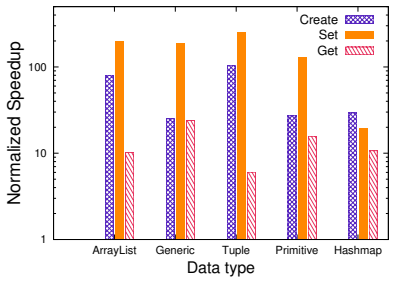


Figure 17. Normalized speedup for PJH compared to PCJ

6.3 Comparison with JPA

We use the JPA Benchmark (JPAB) [34] to compare PJO against JPA, whose detailed description is illustrated in table 2. JPAB contains normal CRUD[5] operations and tests over various features of a JPA framework, such as inheritance, collections and foreign keys. We use unmodified JPA and H2 running on NVDIMM as for the baseline (H2-JPA). The evaluation result in figure 16 indicates that PJO (H2-PJO) outperforms H2-JPA in all test cases and provides up to 3.24x speedup.

Name	Description
BasicTest	Testing over basic user-defined classes
ExtTest	Testing over classes with inheritance relationships
CollectionTest	Testing over classes containing collection members
NodeTest	Testing over classes with foreign-key-like references

Table 2. The description for each test cases in JPAB

To study the overhead brought by PJH, we have also implemented PJO on the stock JVM (H2-PJO-v in figure 16). The data in H2 will be stored within normal heap supported by DRAM. The result shows that PJO introduces 8%-34% overhead due to persistence guarantee and runtime issues such as more complex type checking. It is encouraging for users who want to port their original applications onto NVM with reasonable performance penalty.

We have also exploited *BasicTest* as an example to provide a detailed analysis. We break down the performance into three parts: execution in H2 database, transformation for SQL statements and others. As illustrated in figure 18, the transformation overhead is significantly reduced thanks to PJO. Furthermore, the execution time in H2 also decreases for most cases, which can be attributed to the interface changes from the JDBC interfaces to our *DBPersistable* abstractions.

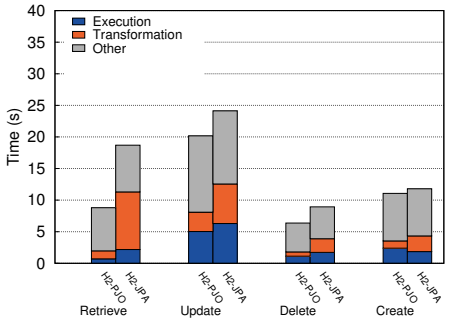


Figure 18. Breakdown analysis for BasicTest

6.4 Microbenchmark

Heap loading time. We test the heap loading time with a micro-benchmark which generates a large number of objects (from 0.2 million to 2 million) of 20 different Klasses. Furthermore, we evaluate heap loading with both user-guaranteed (UG) and zeroing (Zero) safety. As shown in figure 19, the heap loading time for user-guaranteed safety remains constant when the number of objects increases, as the loading time is dominated by the number of Klasses instead of objects (discussed in section 3.3). In contrast, the loading time grows linearly with the number of objects with zeroing safety since it requires a whole heap scan to validate all objects. When the number of objects reaches 2 million, the heap loading time is about 72.76ms, which is still trivial compared to the JVM warm-up time (at least several seconds) as shown in previous work [25].

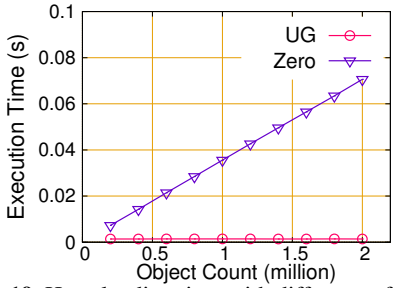


Figure 19. Heap loading time with different safety levels

Recoverable GC. We use a micro-benchmark to test our recoverable GC. The benchmark allocates a large array with

[5]CRUD means four basic operations of persistent storage: create, read, update and delete

millions of objects (from 10 million to 20 million) on PJH and one third of the references will be removed afterwards. It later invokes *System.gc()* to collect PJH by force.

We first evaluate the efficiency of our recoverable GC against the vanilla Parallel Scavenge collector (PSGC) in OpenJDK 8. Since the maximum working set is about 500MB, the heap size for both collectors is set to 1GB for this test. The result is illustrated in figure 20. Our enhanced collector performs much better than PSGC thanks to the single-generational heap organization. Compared to our design, PSGC maintains a young space and has to initiate young GC when it becomes full, which significantly increases the overall GC time. We have also enlarged the heap to 4GB for PSGC to avoid young GC and evaluate with the same benchmark. The result denoted as *PSGC-4G* in figure 20 suggests that our recoverable GC still has comparable performance against PSGC while additionally providing crash consistency.

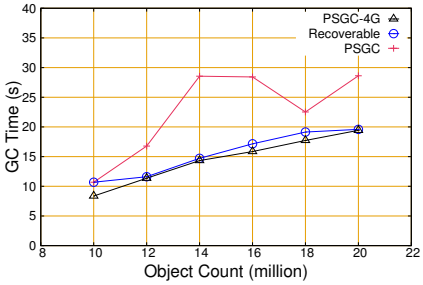


Figure 20. GC time for the microbenchmark

To evaluate recoverability, we inserted random crash events by sending SIGKILL signals during GC. The array size is fixed at 20 million to involve many regions for collection in parallel so that the heap is prone to inconsistency issues. The evaluation is repeated 50 times and the result shows that the recovery phase can always recover from crash and complete the whole collection. The average and maximum recovery time are 2.574s and 4.161s respectively, which is close to that for a normal old GC phase.

7 Related Work

Non-volatile Memory Heaps The invention of recoverable VM [39] once stimulates research on building persistent, recoverable and efficient heaps for user-defined objects [28, 35]. Orthogonally persistent Java [3, 19, 20] is proposed to provide a whole-system-persistent Java runtime with a non-volatile heap. However, it has to cope with tricky issues like the *System* class. Subsequent work turns to persistent object stores [26, 37, 43] and entity-relation mappings [9, 18] for practice. *Espresso* instead discards the requirement of whole-system persistence and provides both coarse-grained and fine-grained persistence atop NVM.

The topic on persistent memory heaps has been renewed recently due to the development of NVM technology. NV-Heaps [8] pioneers in specifying cross-heap pointers. It avoids potential memory leaks by directly disabling non-volatile-to-volatile pointers with a compile-time checker. However, this restriction precludes scenarios where applications leverage both DRAM and NVM, which are common in state-of-art NVM-based systems. Besides, NV-Heaps provides a simple reference-counting garbage collector and has to introduce weak references to manually avoid memory leak through cycles. Our PJH supports flexible pointers and integrates with the fully automatic garbage collector within JVM. Makalu [4] is a persistent memory allocator built on the programming model of Atlas [6]. It provides persistent and recoverable guarantee for the allocation metadata and leverages a recovery-time garbage collector. *Espresso* also considers the crash consistency for the heap but the garbage collection is online thanks to Java's GC service.

Java Runtime Optimization Improving the efficiency of Java runtime has drawn large attention due to its wide utilization in large-scale applications. HotTub [25] finds that class loading is an important source of inefficiency during JVM warm-up and introduces a pool with virtual machines whose classes have been loaded to mitigate the overhead. Our work shares similar wisdom of reducing loading time but in a different way through NVM.

Another line of work studies the performance of garbage collectors and leverage different ways to optimize them. NumaGiC [11, 12] finds that the old garbage collector in Java suffers from scalability issues due to NUMA-unawareness and comes up with a NUMA-friendly algorithm. Yu et al. [46] spot out a performance bottleneck upon destination address calculation in PS old GC and resolved it by caching previous results. Yak GC [33] tries to avoid unnecessary object copying with a region-based algorithm. These mechanisms are orthogonal but may further help optimize our recoverable GC.

Transactions on NVM-based Systems Transactions are a hot topic in building NVM-backed systems [5–7, 10, 14, 22, 27, 29, 42]. Mnemosyne [42] implements semantic-free raw word log (RAWL) in support of transactions. Atlas [5, 6] instead uses synchronization variables like locks and recovery code to provide transaction-like ACID properties, and NVThreads [14] tries to optimize it with a coarse-grained logging protocol. Other work [10, 22, 27, 29] points out that the persist operations (including clflush) should not be included in the critical path of transactions and provide various solutions to move them background. We also propose an abstraction named PJO to provide transaction interfaces.

8 Conclusions

This paper proposed *Espresso* to enable Java programmers to exploit NVM to ease persistence management. *Espresso* comprised Persistent Java Heap (PJH) and Persistent Java Object (PJO) atop PJH. Evaluation showed that the eased persistence management resulted in notable performance boost.

References

[1] DataNucleus. http://www.datanucleus.com/.

[2] Open source database engines in java. https://java-source.net/open-source/database-engines.

[3] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *ACM Sigmod Record*, 25(4):68–75, 1996.

[4] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 677–694. ACM, 2016.

[5] H.-J. Boehm and D. R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 55–67. ACM, 2016.

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[7] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan. Fast in-memory transaction processing using rdma and htm. *ACM Trans. Comput. Syst.*, 35(1):3:1–3:37, July 2017.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[9] L. DeMichiel and M. Keith. Java persistence api. *JSR*, 220, 2006.

[10] M. Dong and H. Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[11] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *ACM SIGPLAN Notices*, volume 48, pages 229–240. ACM, 2013.

[12] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: a garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 661–673. ACM, 2015.

[13] Hewlett Packard Enterprise. Managed data structures. https://github.com/HewlettPackard/mds.

[14] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482. ACM, 2017.

[15] INTEL. Persistent collections for java. https://github.com/pmem/pcj.

[16] INTEL. pmem.io: Persistent memory programming. http://pmem.io/.

[17] Intel and Micron. Intel and micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[18] D. Jordan and C. Russell. *Java data objects*. " O'Reilly Media, Inc.", 2003.

[19] M. Jordan. Early experiences with persistent java. In *Proceedings of the First International Workshop on Persistence and Java*, 2001.

[20] M. J. Jordan and M. P. Atkinson. Orthogonal persistence for javaâĂŤa mid-term report. *Morrison et al.[161]*, pages 335–352, 1999.

[21] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[22] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411. ACM, 2016.

[23] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. *Acm sigplan notices*, 33(10):36–44, 1998.

[24] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[25] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 383–400. USENIX Association, 2016.

[26] 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. *ACM SIGMOD Record*, 25(2):318–329, 1996.

[27] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–343. ACM, 2017.

[28] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 92–101. ACM, 1997.

[29] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *EuroSys*, pages 499–512, 2017.

[30] Micron. Nvdimm. https://www.micron.com/products/dram-modules/nvdimm/.

[31] T. Mueller. H2 database, 2012.

[32] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–148. ACM, 2017.

[33] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[34] ObjectDB Software Ltd. Jpa performance benchmark (jpab). http://www.jpab.org/.

[35] J. O'Toole, S. Nettles, and D. Gifford. Concurrent compacting garbage collection of a persistent heap. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 161–174. ACM, 1994.

[36] M. Paleczny, C. Vick, and C. Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.

[37] J. Paterson, S. Edlich, H. Hörning, and R. Hörning. db4o. 2006.

[38] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. *ACM SIGARCH Computer Architecture News*, 42(3):265–276, 2014.

[39] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems (TOCS)*, 12(1):33–57, 1994.

[40] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104. ACM, 2017.

[41] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[42] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*,

volume 39, pages 91–104. ACM, 2011.

[43] S. J. White and D. J. DeWitt. *QuickStore: A high performance mapped object store*, volume 23. ACM, 1994.

[44] F. Xia, D. Jiang, J. Xiong, and N. Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.

[45] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[46] Y. Yu, T. Lei, W. Zhang, H. Chen, and B. Zang. Performance analysis and optimization of full garbage collection in memory-hungry environments. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 123–130. ACM, 2016.