AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability

Thomas Shull
University of Illinois at Urbana-Champaign
shull1@illinois.edu

Jian Huang
University of Illinois at Urbana-Champaign
jianh@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

Abstract

Byte-addressable, non-volatile memory (NVM) is emerging as a revolutionary memory technology that provides persistency, near-DRAM performance, and scalable capacity. To facilitate its use, many NVM programming models have been proposed. However, most models require programmers to explicitly specify the data structures or objects that should reside in NVM. Such requirement increases the burden on programmers, complicates software development, and introduces opportunities for correctness and performance bugs.

We believe that requiring programmers to identify the data structures that should reside in NVM is untenable. Instead, programmers should only be required to identify *durable roots* – the entry points to the persistent data structures at recovery time. The NVM programming framework should then automatically ensure that all the data structures reachable from these roots are in NVM, and stores to these data structures are persistently completed in an intuitive order.

To this end, we present a new NVM programming framework, named *AutoPersist*, that only requires programmers to identify durable roots. AutoPersist then persists all the data structures that can be reached from the durable roots in an automated and transparent manner. We implement AutoPersist as a thread-safe extension to the Java language and perform experiments with a variety of applications running on Intel Optane DC persistent memory. We demonstrate that AutoPersist requires minimal code modifications, and significantly outperforms expert-marked Java NVM applications.

CCS Concepts • **Hardware → Non-volatile memory**; • **Software and its engineering → Just-in-time compilers**; **Source code generation**.

Keywords Java, Non-Volatile Memory, JIT Compilation

ACM Reference Format:
Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314608

1 Introduction

There have recently been significant technological advances towards providing fast, byte-addressable non-volatile memory (NVM), such as Intel 3D XPoint [37], Phase-Change Memory (PCM) [52], and Resistive RAM (ReRAM) [10]. These memory technologies promise near-DRAM performance, scalable memory capacity, and data durability, which offer great opportunities for software systems and applications.

To enable applications to take advantage of NVM, many NVM programming frameworks have been proposed, such as Intel PMDK [6], Mnemosyne [60], NVHeaps [21], Espresso [62], and others [20, 23, 25, 35, 48]. While the underlying model to ensure data consistency [20, 50] varies across frameworks, all of these frameworks share a common trait: they require the programmer to explicitly specify the data structures or objects that should reside in NVM. This limitation results in substantial effort from programmers, and introduces opportunities for correctness and performance bugs due to the increased programming complexity [53]. Moreover, it limits the ability of applications to use existing libraries.

We believe that requiring users to identify all the data structures or objects that reside in NVM is unreasonable. Instead, the user should only be required to identify the *durable roots*, which are the named entries into durable data structures at recovery time. Given this input, the NVM framework should then automatically ensure that all the data structures reachable from these durable roots are in NVM.

In this paper, we present a new NVM programming framework named *AutoPersist* that only requires programmers to identify the set of durable roots. While most NVM frameworks are implemented in C or C++, we choose to implement AutoPersist as an extension to the Java language. As is common for managed languages, Java already provides transparent support for object movement in memory, as well as high-level semantics for programmers.

AutoPersist shifts the burden of NVM programming to the Java runtime system. In an efficient and thread-safe manner, AutoPersist transparently and dynamically ensures that objects reachable from the specified durable roots reside in NVM, and stores to these data structures are persistently completed in an intuitive order. AutoPersist also provides support for failure-atomic regions. It provides these features by extending the behavior of several bytecodes, modifying the object layout, and introducing new runtime actions. Furthermore, AutoPersist leverages profiling information collected during application warm-up to make intelligent decisions on object placement in advance.

To evaluate the performance and programmability of AutoPersist, we use various applications, including the H2 relational database [2] and a key-value store [8]. We implement AutoPersist in the Maxine Java Virtual Machine [61], and evaluate its performance on a server with Intel Optane DC persistent memory. We find that, with minimal programmer effort, AutoPersist significantly outperforms expert-marked Java NVM applications. Overall, we make the following contributions:

- We propose AutoPersist, a new Java-based NVM framework that requires minimal programmer intervention, while ensuring memory persistency in a transparent manner.
- We extend a Java Virtual Machine with support for transparent, efficient, and thread-safe object access and movement across NVM and DRAM.
- We propose a profile-guided optimization that enables the optimizing compiler to eagerly allocate objects in NVM, reducing AutoPersist's overhead by avoiding unnecessary object movements at runtime.
- We evaluate AutoPersist on a server with Intel Optane DC persistent memory, and demonstrate that it significantly improves both usability and performance.

2 Background

2.1 Byte-Addressable Non-Volatile Memory

Recently, low-latency byte-addressable NVM (also known as NVRAM), has become realizable. In 2017, Intel released its first NVM device [3]. Since then, Intel and other companies have announced plans to introduce many more NVM products in the near future. NVM has a higher density than DRAM, allowing NVM to have larger capacities. Currently, the performance of NVM is slightly worse than DRAM. Hence, initial systems utilizing NVM are expected to be hybrid systems consisting of both NVM and DRAM. Hybrid systems typically have a unified address space, allowing applications to allocate data in both DRAM and NVM.

While NVM moves non-volatile storage a level closer to the processor, many levels of volatile cache still exist between the processor and the NVM. Hence, one needs to ensure that a processor write to NVM propagates its value beyond the levels of caches and into NVM. For this reason, x86-64 processors have introduced the `CLWB` instruction [9], which writes back a cache line to NVM, while retaining the line in the cache. Placing a CLWB instruction after a write ensures that the update will reach NVM.

Still, when we have multiple writes, each followed by a CLWB instruction, the order in which the updates reach NVM and are made persistent is not deterministic. The hardware may reorder the updates on their way to NVM. Hence, to guarantee a distinct ordering, fences must be inserted. Specifically, to guarantee that the CLWB instructions complete one after the other, one needs to place the x86-64 storage fence (`SFENCE`) instruction after every CLWB instruction.

2.2 Existing Frameworks for NVM

The Storage Networking Industry Association (SNIA) has been working to standardize the interactions with NVM. It has created a low-level programming model [4] meant to be followed by device driver programmers and low-level library designers. In addition, an open source project has been created to provide application developers with a high-level toolset that is compliant with SNIA's device-level model. This project has resulted in the development of the Persistent Memory Development Kit (PMDK) [6], a collection of libraries in C/C++ and Java that a developer can use to build durable applications on top of NVM.

PMDK requires that programmers explicitly label all the persistent data in their code with pragmas. As an alternative, PMDK also contains a few library data structures, such as a durable array and hashmap, with the necessary persistent pragmas already built into the library.

For persistently storing data, PMDK requires the programmer to either explicitly persist stores, or use demarcated failure-atomic regions. Failure-atomic regions enable many stores to persistent memory to appear to be persisted atomically. Recently, PMDK has also introduced C++ templates that allow some operations to be persistent without explicit user markings.

In addition to the industrial efforts, academia has also proposed several frameworks for NVM [20, 21, 23, 25, 35, 48, 60, 62]. The level of support provided by these frameworks varies. At best, they provide a similar level of abstraction as PMDK, with the user having to specify all durable objects and also providing some minimal failure-atomic region support.

As an example, Figure 1 shows how to append to a durable list of type E using pragmas representative of existing NVM frameworks. As shown in the figure, the user is expected to explicitly label all the objects to be allocated in NVM with `durable_new`. In addition, after each store, explicit cache line writebacks must be added to ensure that the values are written back to memory. Further, a memory fence is necessary to ensure that all persist operations have completed before the method returns. We discuss the limitations of these existing frameworks in the following section.

```
template <class E>
class DurableList{
  E *element; DurableList *next;
  DurableList append(E *element){
      DurableList *head =
          durable_new DurableList();
      head->element=element;
      CLWB(&head->element);
      head->next=this;
      CLWB(&head->next); SFENCE();
      return head;
  }
}
```

Figure 1. Example using NVM pragmas.

3 Removing the Programming Burden

3.1 Limitations of Existing NVM Solutions

All existing NVM frameworks require the application developer to either mark all the data that should reside in NVM, or to use specialized libraries where this marking has already been completed. This is both programmer intensive and prone to correctness and performance bugs. Correctness bugs occur when the user fails to mark all the data that must reside in NVM, preventing the data from being recovered in the event of a crash. Performance bugs occur when objects not needed to be recovered are marked, forcing these objects to unnecessarily use the slower NVM and be updated in a persistent manner.

Both types of bugs occur because of the difficulty of correctly identifying the transitive closure of all the objects reachable from a given object. Such analysis is necessary when deciding what objects are reachable at recovery time and hence must be placed in NVM. Determining this transitive closure is difficult for multiple reasons, including aliasing, the use of libraries, and the sheer size of modern application codebases. The use of libraries is especially problematic, as it may not be possible to have access to the source code to mark persistent objects within the libraries.

While requiring the programmer to label many objects is both user-intensive and bug-prone, it does match the traditional level of abstraction provided by lower-level languages such as C and C++. However, in higher-level managed languages such as Java, Scala, Python, and JavaScript, expecting the programmer to perform this type of low-level reasoning is unacceptable. In managed languages, the underlying object representation is hidden from the user, which allows the runtime to decide how to allocate and lay out objects. In addition, managed languages provide automatic memory management. The user does not need to reason about object lifetimes; instead, the runtime automatically detects and collects dead objects. Requiring developers of managed language applications to explicitly add markings for objects to reside in NVM does not match their expectations. Further, it exposes part of the allocation process to the user, violating the user-runtime boundary. In their paper, Shull et al. [56] describe in more detail the drawbacks of existing NVM frameworks and their misalignment with managed languages.

3.2 A New Framework for Managed Languages

We argue that, in managed languages, it should be the runtime's obligation to identify non-volatile objects, and to ensure that they are persisted correctly. As managed languages already transparently manage the underlying memory, it is a natural extension of the runtime support to automatically move objects to NVM as necessary when they need to be recoverable.

Consequently, we propose a new framework model for managed languages where the user must only identify *durable roots*. We call our new model and implementation *AutoPersist*. In AutoPersist, we define durable roots to be the named entry points into durable data structures at recovery time. Given the set of these durable roots, AutoPersist's runtime has two requirements:

Requirement 1. *All objects reachable from the durable root set must be in NVM.*

This requirement forces the runtime to dynamically check objects and potentially move them throughout the execution. To do so, the runtime must add additional checks to the code, and dynamically update pointers to moved objects.

Requirement 2. *All objects reachable from the durable root set must be automatically persisted when modified.*

This requirement forces the runtime to dynamically check the status of objects throughout execution. When objects reachable from the durable root set are updated, the updates need to be persisted in the correct order.

This type of support is natural to have in managed languages, as they commonly monitor objects, determine reachability from a root set, and move objects while performing garbage collection (GC). Put in another way, we are proposing to modify the behavior of mutator threads to actively move objects to NVM when stores make these objects reachable from a durable root. This behavior is very similar to concurrent GC. However, whereas the goal of concurrent GC is to free memory while not introducing pauses, in our framework the mutator threads move objects to NVM to ensure crash consistency. In addition, in our framework, after these stores, the mutator threads must insert the necessary cache line writebacks and fences to ensure the stores are persisted in the correct order.

3.3 An Example

We present an example of the required behavior of AutoPersist in Figure 2. Figure 2(a) shows the initial state of the heap, where objects *A*, *B*, *C*, *D*, and *E* are in volatile memory, and *F* and *G* are in NVM. Object *G* is pointed to by a durable root and, hence, must be in NVM. Object *F* is reachable from

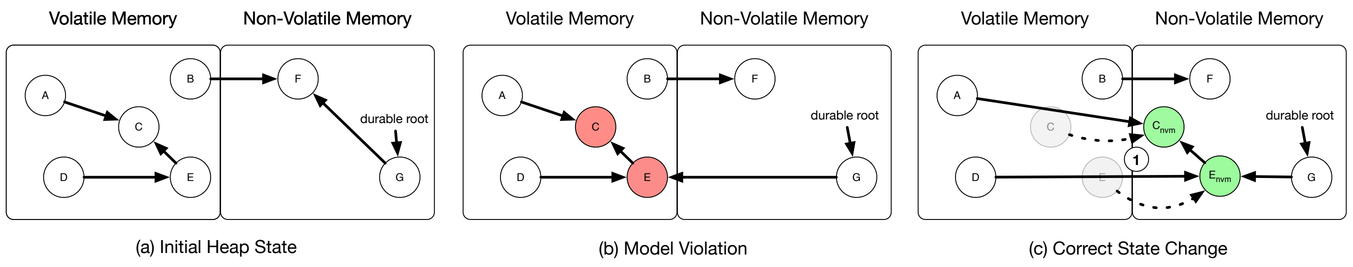

Figure 2. Changing heap state to meet the requirements of AutoPersist's model.

G and must also be in NVM. Because the other objects are not reachable from a durable root, they do not need to be in NVM.

The program changes the $G \rightarrow F$ pointer to $G \rightarrow E$ in Figure 2(b), which leaves the heap in an incorrect state. Objects E and C are now reachable from G but are still in volatile memory. They could not be recovered if a crash were to occur. To ensure that our framework's requirements are met, the runtime makes the changes shown in Figure 2(c). Specifically, before G's pointer changes, the runtime moves E and C from the volatile to the non-volatile heap (i.e., new objects E_{nvm} and C_{nvm}) (operation ①). Then, it adjusts all the pointers to the original E and C objects. Since F is not reachable from a durable root anymore, eventually it will be moved back to volatile memory.

4 AutoPersist Programming Model

4.1 Labeling Durable Roots

AutoPersist requires the programmer to declare the set of durable roots. Declaring a durable root consists of two parts: identifying the object and associating a name with it. We add a new annotation to Java [32], `@durable_root`, which is used to label fields containing objects. A field labeled with `@durable_root` indicates that the object pointed to by this field is a durable root.

Only static fields can be labeled with `@durable_root` in our model. Static fields have a unique name in the application environment, and hence can be easily identified at recovery time. While adding support in AutoPersist to allow dynamic fields to also be `@durable_roots` is trivial, we believe that the benefits that this additional feature would provide are outweighed by the opportunities for programmer mistakes that it would introduce. As multiple instances of the object could be created, it would be easy for the programmer to make mistakes when associating the durable root to a specific instance of the object.

4.2 Failure-Atomic Region Support

The default behavior of AutoPersist is to ensure that stores to objects reachable from a `@durable_root` are persisted in sequential order. However, in some situations, it may be necessary to provide the appearance of multiple stores completing *atomically* from the crash-consistency perspective. To allow this, our framework supports failure-atomic regions.

In AutoPersist, the user is expected to label the start and end of failure-atomic regions. Given these labels, the runtime ensures that all stores to objects reachable from a durable root within this region complete atomically from a crash-consistency perspective at the end of the region. There is no additional user involvement. AutoPersist uses a flattened nesting approach to ensure values are not made persistent prematurely. Like other implementations [6, 31, 42, 46], our failure-atomic region support is meant solely to provide all-or-nothing visibility to persistent data in the event of a crash. It does not detect data races or perform rollbacks like software transactional memory. Instead, the user is still expected to provide any synchronization needed to prevent data races in accordance with the Java memory model [45]. This type of concurrency model is known as an open transactional model [16]. Section 6.5 covers how we implement failure-atomic region support in AutoPersist.

4.3 Persistency Model

AutoPersist provides a simple and intuitive persistency model. Outside of failure-atomic regions, all writes to values reachable from a `@durable_root` are persisted in a sequential order. Inside of failure-atomic regions, no data is made persistent until the end of the region. At that point, all stores to data reachable from a `@durable_root` within the region are made persistent atomically.

To ensure sequential persistency outside of failure-atomic regions, AutoPersist detects the case when a value V is being stored into an object O that is reachable from a `@durable_root`. When this happens, the actions that AutoPersist takes depend on the state of the value V being stored. If V is either a primitive value or was previously reachable from a `@durable_root`, then AutoPersist ensures that the store to object O is done persistently by adding a CLWB and an SFENCE after the store.

However, if V is an object that was not previously reachable from a `@durable_root`, before AutoPersist can store V in O, AutoPersist must make V and its transitive closure persistent. Note that the order in which AutoPersist makes V and its transitive closure persistent does not affect the persistency model. This is because V will be unrecoverable until V is stored into O. It only matters that V and its transitive closure are made persistent before this store is performed.

While searching and potentially relocating *V* and its transitive closure, AutoPersist also inserts the necessary CLWBs to ensure their persistency. Before the store of *V* in *O*, AutoPersist inserts an SFENCE to ensure that all CLWBs have completed. After the store, AutoPersist inserts a CLWB and an SFENCE. In Section 6.1, we discuss how to update the objects that pointed to the old locations of *V* or its transitive closure.

Inside failure-atomic regions, before every store to an object reachable from a `@durable_root`, AutoPersist saves in a persistent undo log the value that will be overwritten. The undo log operation is followed by a CLWB and SFENCE to ensure that the log entry has been made persistent. After that, the store to the object is performed and a CLWB is added to write back the new update to NVM. At the end of the failure-atomic region, AutoPersist inserts an SFENCE to ensure that all the stored data has reached the NVM. Then, the undo log is discarded. With this design, stores to objects reachable from a `@durable_root` are allowed to be completed out of order, but they are all persisted at the end of the region. Moreover, if the atomic region fails to complete, the undo log in persistent memory is used to undo all of the updates in the region that were persisted. Such updates should not be part of the crash-consistent program state.

This persistency model only applies to data reachable from `@durable_roots`. None of the other data will be recovered in the event of a crash. Hence, it does not need to abide by our framework's persistency model. Such data can be reordered in accordance with the Java memory model standard.

4.4 Recovery API

In order to recover data from a `@durable_root` after a crash, we must have recovery code that allows the program to retrieve previous versions of an object as it starts-up. To allow this, we extend the Java Object class to include a new method, `recover(String image)`, which attempts to recover the value of the implicit object argument within a named image. In order to differentiate multiple executions running simultaneously, when initializing execution, the programmer is expected to provide an image name for the given execution. This image name is used to recover objects from the execution's non-volatile heap. The `recover` method is expected to be called from a `@durable_root`. If either the named image cannot be found or the object the method is invoked from is not a durable root, then `null` is returned.

Figure 3 shows a simple example of how to use this method. The example tries to recover a key-value store. If the key-value store cannot be recovered, then a new version of it is instantiated.

4.5 Introspection API

A strength of AutoPersist is that its simple abstraction frees the programmer from having to worry about many details. However, sometimes, such as when debugging, the user may want to extract more object information. For this reason, AutoPersist includes several method calls that allow for introspection. The method calls are: `isRecoverable()`, `inNVM()`, `isDurableRoot()`, `inFailureAtomicRegion(tid)`, and `failureAtomicRegionNestingLevel(tid)`.

```
@durable_root
public static KeyValueStore kv;
static{
    if((kv = kv.recover("image_name")) == null){
        kv = new KeyValueStore();
    }
}
```

Figure 3. Recovery API example.

The functionality of most of these calls is self-evident. `isRecoverable()`, `inNVM()`, and `isDurableRoot()` are called by an object and return a boolean of the requested information. On the other hand, `inFailureAtomicRegion(tid)` and `failureAtomicRegionNestingLevel(tid)` take a thread identifier as argument, and query it for the desired information.

4.6 *Unrecoverable* Keyword

In some situations, a programmer may decide that some data reachable from a `@durable_root` does not need to be recoverable across a crash. To provide this functionality, AutoPersist includes the `@unrecoverable` annotation, which can be applied to any dynamic object field. Any field labeled with this annotation will disable AutoPersist's requirements on stores to that field.

We anticipate that `@unrecoverable` may be used to limit the performance impact of persistency when objects can be recovered or recreated via other means. However, we strongly argue that the default behavior should be that all objects reachable from durable roots should be handled in a crash-consistent manner. This approach minimizes the likelihood of programmer mistakes.

5 Applying AutoPersist to the JVM

In this section, we describe how we change the Java Virtual Machine (JVM) to meet our NVM framework's requirements.

In AutoPersist, an object can be in one of three states: *Ordinary*, *Converted*, and *Recoverable*. The ordinary state means that the object will not be recovered in the event of a crash. The recoverable state indicates that the object is reachable from a durable root, and will be recovered in the event of a crash. The converted state means that the object is in the process of transitioning from the ordinary to the recoverable state. The object and its transitive closure may not yet be reachable from a durable root. However, the runtime is in the process of making them reachable. For brevity, an object that is in either the converted or recoverable state is said to be in the *ShouldPersist* state.

A programmer may choose to mark a field in an object as `@unrecoverable`. In such case, AutoPersist does not perform any persistency-related action on the field.

5.1 Modified Object Store Operations

Our framework alters the behavior of several JVM bytecodes. Below we highlight the main changes to storing to static and dynamic object fields, as well as to arrays.

5.1.1 Storing to Static Object Fields

Storing to a static field in Java is represented by the `putstatic (C,F,V)` bytecode. Normally, this instruction stores value V into field F of class C's static object representation. AutoPersist's new implementation of `putstatic` is shown in Algorithm 1.

In the `putStatic` procedure, first, if the value to be stored is an object, the algorithm finds the real current location of the object (Line 3). This is necessary because, as we will discuss in Section 6.1, when an object is moved to NVM, not all the pointers to it are immediately updated. Instead, AutoPersist leaves behind some temporary forwarding objects that point to the object's new location in NVM.

Next, if the field being stored to is a persistent root and the value being stored into the field is not recoverable, then the value is made recoverable (Lines 4-5). This is the only case that needs action for stores to static object fields.

After this, if the thread is in a failure-atomic region and the field is a persistent root, the old value is logged. Next, the value is written to the field. Finally, if the field is a persistent root, then the address of the object is stored in a global table (Line 13) that will be used to retrieve the object in a recovery.

5.1.2 Storing to Dynamic Object Fields and Arrays

Storing to a dynamic object field in Java is represented by the `putfield(H,F,V)` bytecode. Normally, this instruction stores value V into field F of dynamic object field holder H. Procedure `putField` in Algorithm 1 shows our new implementation. It is similar to `putStatic`, but has a few notable differences. First, the field being stored to cannot be a persistent root, so this condition does not need to be checked. Second, the holder object itself may now be in the ShouldPersist state. Therefore, for `putField`, the state of the holder object dictates whether the value to be stored must be made recoverable. Note that if the field is marked as `@unrecoverable`, no persistency action is taken. Line 20 reflects the appropriate check used to determine whether the value needs to be made recoverable.

After the object's field is updated (Line 27), the state of the holder determines what additional actions must be performed to satisfy our model. If the holder object is in the ShouldPersist state and the field stored to is not `@unrecoverable`, then the corresponding cache line is written back (Line 29). Further, if we are not in a failure-atomic region, a fence is inserted to guarantee completion of the writeback (Line 31).

Algorithm 1 Modified object store operations.

```
 1: procedure PUTSTATIC(class, field, value)
 2:     if typeof(value) is Object then
 3:         value = getCurrentLocation(value)
 4:         if isDurableRoot(field) and !isRecoverable(value) then
 5:             value = makeObjectRecoverable(value)
 6:         end if
 7:     end if
 8:     if inFailureAtomicRegion(tid) and isDurableRoot(field) then
 9:         logStore(class, field)
10:     end if
11:     writeField(class, field, value)
12:     if isDurableRoot(field) then
13:         RecordDurableLink(field, value)
14:     end if
15: end procedure

16: procedure PUTFIELD(holder, field, value)
17:     holder = getCurrentLocation(holder)
18:     if typeof(value) is Object then
19:         value = getCurrentLocation(value)
20:         if !isUnrecoverable(field) and isShouldPersist(holder) and !isRecoverable(value) then
21:             value = makeObjectRecoverable(value)
22:         end if
23:     end if
24:     if inFailureAtomicRegion(tid) and !isUnrecoverable(field) and isShouldPersist(holder) then
25:         logStore(holder, field)
26:     end if
27:     writeField(holder, field, value)
28:     if !isUnrecoverable(field) and isShouldPersist(holder) then
29:         cachelineWriteback(holder, field)
30:         if !inFailureAtomicRegion(tid) then
31:             persistFence()
32:         end if
33:     end if
34: end procedure

35: procedure ARRAYSTORE(holder, index, value)
36:     holder = getCurrentLocation(holder)
37:     if typeof(value) is Object then
38:         value = getCurrentLocation(value)
39:         if isShouldPersist(holder) and !isRecoverable(value) then
40:             value = makeObjectRecoverable(value)
41:         end if
42:     end if
43:     if inFailureAtomicRegion(tid) and isShouldPersist(holder) then
44:         logStore(holder, index)
45:     end if
46:     writeArray(holder, index, value)
47:     if isShouldPersist(holder) then
48:         cachelineWriteback(holder, index)
49:         if !inFailureAtomicRegion(tid) then
50:             persistFence()
51:         end if
52:     end if
53: end procedure
```

Stores to arrays (JVM's `{a,b,c,d,f,i,l,s}astore` bytecodes) are also modified in a way similar to `putfield`. Procedure `arrayStore` in Algorithm 1 shows the modifications.

5.2 Object Header

As the internal object representation is hidden from the user in Java, we modify the object layout to assist with our implementation. We add a 64-bit header word to each object, which we call the *NVM_Metadata* header. This header stores information about the state of the object relevant to AutoPersist. Figure 4 shows the fields in our object header word.

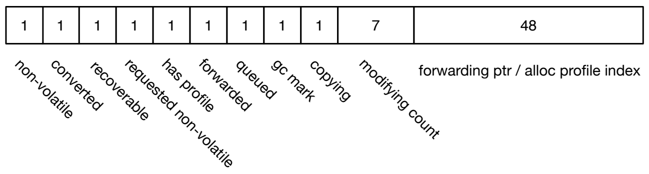

Figure 4. NVM_Metadata header contents.

In the header, the *converted* and *recoverable* bits denote the object state: converted objects have the converted bit set; recoverable ones have the recoverable bit set; ordinary objects have both bits clear. The rest of the bits are introduced in subsequent sections.

6 Advanced Implementation Aspects

This section describes transparently updating pointers, determining which objects to move to NVM, thread safety, garbage collection, and failure-atomic region support.

6.1 Transparently Updating Pointers

When an object is moved from volatile memory to NVM, all pointers to the original location of the object must be updated to reflect its new location (Figure 2). However, AutoPersist adjusts the pointers *lazily*; for performance, it temporarily inserts a level of indirection for some pointers until GC occurs.

For example, in Figure 2, when objects C and E are moved to NVM, the pointers from objects A, D, E, and G would also need to be updated. However, supporting the ability to change all of these pointers at the time of the move would have prohibitive performance overheads. Indeed, we would have to add a pointer table, and introduce a level of indirection to all pointer accesses. Alternatively, at the time of the move, we could search the entire heap to discover and update pointers to the moved objects. Either of these options would result in significant slowdowns.

Consequently, AutoPersist temporarily retains the original C and E objects and converts them into *forwarding* objects. Only the new pointers from the recoverable objects (G and E_{nvm}) point to the new recoverable copies of the objects (E_{nvm} and C_{nvm}). The other pointers are left pointing to the forwarding objects (i.e., A to C, and D to E) until a GC cycle is executed.

Note that this approach is correct, as it relies on the following key insight: if an object is in volatile memory, then all pointers to the object must be from objects not reachable from the durable root set. This is true by Requirement 1. Hence, if an object is moved, its original location can be used as a temporary forwarding pointer for pointers from objects in volatile memory. The only objects that cannot use this forwarding pointer in volatile memory are the objects that were in NVM or have been moved to NVM. The pointers from these objects are updated during the moving process.

In AutoPersist, the NVM_Metadata header of forwarding objects is set as follows: the *forwarded* bit is set, and the 48-bit *forwarding ptr* field points to the object's real location in NVM (Figure 4). In addition, some JVM bytecodes are adjusted to check for forwarding objects.

Algorithm 2 shows how bytecodes must be altered. First, procedure `getCurrentLocation` retrieves the current location of an object. It checks the object's forwarded bit in the NVM_Metadata header to see if the object currently pointed to is a forwarding object (Line 2). If so, the procedure reads the real location of the object from the forwarding ptr field in the header (Line 3).

Algorithm 2 Modified object load operation.

```
 1: procedure GETCURRENTLOCATION($obj$)
 2:     if isForwarded(obj) then
 3:         return getForwardingPtr(obj)
 4:     end if
 5:     return obj
 6: end procedure

 7: procedure GETFIELD(holder, field)
 8:     holder = getCurrentLocation(holder)
 9:     value = readField(holder, field)
10:     if typeof(value) is Object then
11:         newValue = getCurrentLocation(value)
12:     end if
13:     return newValue
14: end procedure
```

The second procedure, `getField(H,F)`, shows how the JVM bytecode `getfield` must be modified. Originally, this instruction loads the value stored in field F of dynamic object field holder H onto the JVM stack. Now, we call `getCurrentLocation` to ensure that the correct pointers are being used (Lines 8 and 11). Many of the procedures shown in Algorithm 1 must also perform this same check. Similar modifications are made to other JVM bytecodes that load and store values, namely, `getstatic`, `if_acmpeq`, `if_acmpne`, `monitorenter`, `monitorexit`, and the various array load bytecodes.

During GC, pointers to forwarding objects are updated to point to the real objects, and the forwarding objects are removed. As GC already must adjust pointers, it is natural for AutoPersist to perform this operation during GC.

6.2 Movement of Objects

In AutoPersist, it is the responsibility of the runtime to move objects to NVM when necessary during execution, to ensure all objects reachable from the durable root set are in NVM.

This means that the runtime must potentially trace the transitive closure of an object to ensure that all reachable objects are persistent.

Algorithm 3 shows the various procedures used for this operation. Procedure `makeObjectRecoverable` manages the phases of the operation. First, the initial object (i.e., the one that initiates the transitive search) is passed to procedure `addToQueueIfNotCoverted` (Line 2) to be added to a thread-local *work queue*. This queue holds the objects that need to be processed to ensure that the transitive closure is in NVM. To ensure that a given object is not placed twice in the work queue, we add the *queued* bit in the NVM_Metadata header of each object (Figure 4). If an object is in the work queue, the queued bit is set. Inter-thread dependencies are also detected at this point (Line 18). Note that multiple threads may be performing this action simultaneously. Hence, we use a CAS operation to set the queued bit (Line 22). Once the queued bit is set, the object is placed in the local work queue without synchronization.

Next, procedure `convertObjects` is called. This procedure processes the objects in the work queue (Line 28). For each object, we first check whether it is already allocated in NVM. The *non-volatile* bit in the NVM_Metadata header (Figure 4) is set if the object is in NVM. If the bit is not set, the object is moved to NVM (Line 31). In either case, cache line writebacks must be inserted to guarantee that the entire contents of the object are persistent (Line 33). Since AutoPersist can precisely determine an object's layout, the runtime is able to insert the minimal number of CLWBs necessary to ensure that the entire object has been written back. Next, the converted bit of the NVM_Metadata header is set. After this, we search all the objects that are reachable by pointers from the current object and, if necessary, add them to the work queue (Line 36). Note that fields annotated with the `@unrecoverable` marking are not searched.

While doing this, the algorithm also checks each of the pointers to see if they will need to be updated. Pointers will need to be updated if the object they point to will be moved to NVM while executing this algorithm. Such pointers are placed in another queue, the *ptr queue*, for later processing (Line 38). Recall that these updates are necessary to prevent persistent objects from pointing to volatile forwarding objects.

Finally, if the object has moved, we want the work queue to point to the new location of the object (Line 41).

When the `convertObjects` procedure returns, the thread must ensure that other objects reachable from the initial object and that are being persisted by other threads are already persisted. This is done by monitoring a global table and checking whether the other threads have finished their work (Line 4). If they have not, the thread waits until they do. In practice, we observe very little wait time.

The next step is to call procedure `updatePtrLocations` to update all pointer locations within the ptr queue (Lines 47

Algorithm 3 Transitive persist.

```
 1: procedure MAKEOBJECTRECOVERABLE(object)
 2:     addToQueueIfNotConverted(object)
 3:     convertObjects()
 4:     wait for other threads to complete phase
 5:     updatePtrLocations()
 6:     wait for other threads to complete phase
 7:     markRecoverable()
 8:     return getCurrentLocation(object)
 9: end procedure

10: procedure ADDTOQUEUEIFNOTCOVERTED(obj)
11:     do
12:         obj = getCurrentObject(obj)
13:         oldHeader = readPersistentHeader(obj)
14:         if isRecoverable(obj) then
15:             return
16:         end if
17:         if isConverted(obj) or isQueued(obj) then
18:             detect any inter-thread dependency
19:             return
20:         end if
21:         newHeader = setIsQueued(oldHeader)
22:     while !CAS(obj, oldHeader, newHeader)
23:     workQueue.add(obj)
24:     return
25: end procedure

26: procedure CONVERTOBJECTS
27:     idx = 0
28:     while idx != workQueue.size() do
29:         obj = workQueue[idx]
30:         if !isNonVolatile(obj) then
31:             obj = moveToNonVolatileMem(obj)
32:         end if
33:         write back entire object to NVM
34:         setIsConverted(obj)
35:         for (ref, offset) in nonUnrecoverableReferences(obj) do
36:             addToQueueIfNotConverted(ref)
37:             if !isNonVolatile(ref) then
38:                 ptrQueue.add(obj, offset, ref)
39:             end if
40:         end for
41:         workQueue[idx] = obj
42:         idx += 1
43:     end while
44: end procedure

45: procedure UPDATEPTRLOCATIONS
46:     while ptrQueue.size() != 0 do
47:         (obj, offset, ref) = ptrQueue.pop()
48:         ref = getCurrentLocation(ref)
49:         writeOffset(obj, offset, ref)
50:     end while
51: end procedure

52: procedure MARKRECOVERABLE
53:     idx = 0
54:     while !workQueue.isEmpty() do
55:         obj = workQueue.pop()
56:         setRecoverable(obj)
57:     end while
58: end procedure
```

to 49). Afterwards, once again in rare cases, the thread pauses for other threads to complete their work (Line 6).

The last step of this algorithm is to call the `markRecoverable` procedure to set the recoverable flag of all objects modified by this thread (Line 7). Recall that when recoverable is set for an object, it means that all objects reachable from this object are also persistent. This is stronger than the converted flag, which is a transition state. Finally, the process returns the object's current location (Line 8).

Mapping our three object states to traditional tri-color GC terms [27], the ordinary state is the white color, the converted state is the gray color, and the recoverable state is the black color. In other words, if a mutator thread encounters a converted object while performing a store, then it must proactively make the object's new transitive closure recoverable, even though the object is not yet reachable from a durable root. This is necessary to ensure that a crash-consistent state is maintained in the presence of concurrent mutations.

6.3 Thread Safety

Since Java is multithreaded, it is possible for a thread to try to access an object as the object is being moved to NVM. Without precautions, this can create a race condition that creates an execution state not possible in the Java memory model. To prevent this, caution must be taken in two places: when moving objects to NVM, and when storing to objects. This is because, without synchronization, it may be possible for these two events to race and for stores to be lost.

To prevent this race from occurring, we introduce two new fields in the NVM_Metadata header: *copying* and *modifying count* (Figure 4). The copying flag is set while the object is being copied over to NVM. The modifying count field indicates the number of threads that are currently in the process of modifying the object. Both fields are updated using CAS operations.

Algorithm 4 shows `moveToNonVolatileMem`, the thread-safe procedure to move an object to NVM. A thread is only allowed to copy an object to NVM when no other thread is in the process of modifying the object. Hence, the procedure checks the object's modifying count and waits to perform the copy until the modifying count is zero (Line 6).

To improve performance, we perform two optimizations. First, while an object is being copied, we still allow another thread to modify the object. To modify the object, a thread clears the copying flag before performing the modification. Hence, if the copying thread detects that the copying flag has been cleared during the copying (Line 14), then the copy must be performed again. Otherwise, the operation has been successful, and the thread resets the copying flag (Line 18).

The second optimization is not to increment the modifying count unless necessary. Incrementing the count is only necessary if the modifying thread detects that the object may have moved while it was performing the modification. The thread can check this by reading the object's NVM_Metadata header state and the object's address before and after it performs the write. Note that we need to place a fence between the write and subsequent reads to ensure that the write has completed by the time the reads are issued. If a change is detected, the write is repeated, this time incrementing the modifying count. This code is not shown due to lack of space.

Algorithm 4 Moving object to NVM.

```
1: procedure MOVETONONVOLATILEMEM(obj)
2:     newObj = allocateNVM(sizeof(obj))
3:     while true do
4:         do
5:             oldHeader = readPersistentHeader(obj)
6:             if getModifyingCount(oldHeader) > 0 then
7:                 continue
8:             end if
9:             newHeader = setIsCopying(oldHeader)
10:        while !CAS(obj, oldHeader, newHeader)
11:        copyMem(obj, newObj, sizeof(obj))
12:        do
13:            oldHeader = readPersistentHeader(obj)
14:            if !isCopying(oldHeader) then
15:                continue
16:            end if
17:            newHeader = unsetIsCopying(oldHeader)
18:        while !CAS(obj, oldHeader, newHeader)
19:        return newObj
20:    end while
21: end procedure
```

6.4 Allocation and Garbage Collection

Since there are now volatile and non-volatile portions of the heap, our runtime's allocator and garbage collector must be adjusted to account for this expansion, and to ensure that objects are placed in the correct portion of the heap. For allocation, thread local allocation buffers (TLABs) are used. Each thread has both a volatile and a non-volatile TLAB, which it can use to bump-allocate objects.

For GC, our implementation uses a stop-the-world copying collector for both parts of the heap. During a collection, if a forwarding object is encountered, all pointers to that object are adjusted to point to the object's new location, and the forwarding object is reaped.

Normally, during GC, objects are copied to either volatile memory or NVM based on their original location. One optimization we add to our GC implementation is to detect if an object is no longer reachable from a durable root and, if so, move the object back to volatile memory. To implement this optimization, we use two new flags in the NVM_Metadata header: *gc mark* and *requested non-volatile* (Figure 4). The gc mark flag is used during the GC cycle to identify which objects are reachable from a durable root. Before a GC cycle starts, our collector walks the heap and sets the gc mark flag for all objects that are reachable from a durable root. These are the objects that must stay in NVM. The requested non-volatile flag indicates to the collector that this object should remain in NVM even if it is not reachable by any durable root. We use this flag so as not to interfere with the optimization we propose in Section 7.

Note that since some objects are moved back from NVM to volatile memory during a GC cycle, it is possible for some objects not reachable from the durable root set to be still in NVM when an application crashes. To ensure consistency, at recovery time, when an existing image is loaded, a GC cycle is performed on the NVM to free all the objects not reachable from the durable root set.

6.5 Failure-Atomic Region Support

As described in Section 4.2, AutoPersist supports failure-atomic regions. Given the semantics of AutoPersist, we are given much flexibility in choosing how to design our implementation. Currently, AutoPersist uses per-thread undo logs with write-ahead logging. As shown in Algorithm 1 (Lines 9, 25, and 44), inside a failure-atomic region, any value within a durable object that will be overwritten is first logged ahead of the store. This involves copying the original value, a pointer to the object, and the value's offset within the object's internal layout to a thread-local log. Logging this information ensures that the object can be correctly restored in the event of a crash.

For each JVM thread, AutoPersist adds a counter indicating the current failure-atomic region nesting level, and a pointer to its thread-local undo log. The undo log is also considered a durable root, to ensure that all objects pointed to by the log continue to be persisted correctly. At the end of the failure-atomic region, the thread's undo log is cleared, allowing any dead objects to be reclaimed.

Many previous works have tried to optimize the performance of logging within failure-atomic regions [28, 30, 31, 40, 42, 46, 55, 60]. We believe that many of their optimizations can also be applied to AutoPersist. Since the implementation of our failure-atomic region support is transparent to the user, our runtime is free to internally change its implementation. We leave this as future work.

7 Optimizing Object Allocation

Modern Java implementations support tiered compilation. When a method is first invoked, it is compiled by a compiler that completes quickly but does not generate very optimized code. Later, if the method is deemed important, it is recompiled using an optimizing compiler that produces higher-quality code. In addition, the initial compiler tier typically inserts profiling information into the code, which is later used by the optimizing compiler to generate better code.

In AutoPersist, we modify the initial compiler to produce profiling information that is used by the optimizing compiler to reduce object handling overhead. Specifically, a source of overhead in our implementation is when an object is moved to NVM because it becomes reachable from a durable root. AutoPersist reduces this overhead by predicting that an object will eventually be moved to NVM, and eagerly allocating it in NVM in the first place. To support this optimization, AutoPersist inserts profiling information to identify which allocation sites often create objects that later need to be moved to NVM. Once these sites are identified, AutoPersist eagerly allocates objects from these sites in NVM. Note that determining such information statically is hard due to the presence of numerous control-flow paths and aliasing, and the need for inter-procedural analysis.

AutoPersist's profiling information is implemented as follows. First, each profiled allocation site is given an entry in a global table called *allocProfile*. The entry contains a count of the number of objects allocated from this site that are later moved to NVM. During execution, as objects are instantiated, two new fields in their NVM_Metadata header (Figure 4) are set as follows: the *has profile* flag is set, and the *alloc profile index* field is set to the index of the entry within the *allocProfile* table corresponding to its allocation site. If the object is later moved to NVM, the entry within *allocProfile* corresponding to the object's allocation site is incremented. To access the correct entry within *allocProfile*, the object's alloc profile index field is read. Note that it is fine for both the forwarding ptr and the alloc profile index to share the same field in the NVM_Metadata header, as they are not needed at the same time.

The compiler also retrieves profiling information on the number of method invocations and branch behavior. Via this information, the compiler is able to accurately estimate the total number of objects allocated from a site. Later, when the optimizing compiler recompiles a method, for each of its allocation sites, it checks the total number of objects allocated and the *allocProfile* count. Based on these values, it decides on whether the site should either continue to allocate objects in volatile memory or switch to eagerly allocating in NVM. To prevent the GC from moving objects eagerly allocated in NVM back to volatile memory, these objects set the requested non-volatile flag (Section 6.4) in their NVM_Metadata header.

Note that deciding which memory to use for initial object allocation is a performance issue and not a correctness one. AutoPersist guarantees that the necessary objects will be moved to NVM to meet our model's requirements. This profiling information simply helps to attain higher performance.

8 Evaluation Environment

Compiler Platform. We implement the AutoPersist framework within the Maxine JVM [61]. Maxine is an open-source research JVM that enables the fast prototyping of new features while achieving competitive performance. We use Maxine 2.0.5, and modify both its initial tier compiler (T1X) and its optimizing compiler (Graal). In addition, we modify its object layout to integrate our NVM_Metadata header (Figure 4), add new NVM heap regions, extend its GC (Section 6.4), and implement failure-atomic regions (Section 6.5).

Server Configuration. We use a server with multiple 128GB Intel Optane DC persistent memory modules and 384GB of

DDR4 DRAM. The server contains two 24-core Intel® second generation Xeon® Scalable processors (codenamed Cascade Lake), and runs Fedora 27 on Linux 4.15. In all of our experiments, we set up AutoPersist to reserve 20GB for each of the volatile and non-volatile heap spaces. To create the non-volatile heap, we use `libpmem` [6] to map a portion of the application's virtual address space to NVM. After that, via the Direct Access (DAX) protocol, applications can directly interact with the Intel Optane DC persistent memory. We use cache line writebacks (`CLWB`) and store memory fences (`SFENCE`) to persist values.

8.1 Applications

To evaluate AutoPersist, we perform experiments on two real-world applications, namely, a key-value store and the H2 relational database, and several kernels.

Key-Value Store. We implement a persistent version of a key-value store using AutoPersist. Specifically, we modify QuickCached [8], a pure Java implementation of Memcached to use persistent data structures internally for its key-value storage. The different backends that we compare are:

- **IntelKV.** This is Intel's `pmemkv` library [7], along with its Java bindings. This backend uses its *kvtree3* configuration, which consists of a hybrid b+ tree written in C++ using the PMDK library version 1.5. Similar to existing works [49], in this implementation, only the leaf nodes are in persistent memory. Note that the *IntelKV* backend does not use AutoPersist. Hence, it runs on an unmodified JVM.
- **Func.** This backend uses the PCollection library [5] and is implemented in Java. We create two versions: one in AutoPersist and one using the Espresso framework [62]. Espresso requires the user to add markings identifying objects to allocate in NVM, to mark stores that must be flushed to NVM, and to insert memory fences.
- **JavaKV.** This backend uses the same B+ tree structure as *IntelKV* and is implemented in Java. Like *Func*, we create two versions: one in AutoPersist and one in Espresso.

H2 Database. We modify the H2 relational database [2] to use AutoPersist. H2 is a popular SQL database written in Java. Currently, H2 has two persistent storage engines. One is MVStore, which is a log structured store and is currently H2's default storage engine. The other is PageStore, which is H2's legacy backend. We modify MVStore to use AutoPersist to persist the database's internal data structures instead of writing them out to files. In the evaluation, we compare our modified storage engine against both MVStore and PageStore. For a fair comparison, we direct MVStore and PageStore to use NVM as storage, as opposed to SSDs, to ensure their file operations execute as efficiently as possible.

Database Driver. To evaluate the performance of both the key-value store and H2 database, we use the Yahoo! Cloud Serving Benchmark (YCSB) [24]. This is a benchmark suite commonly used to evaluate the performance of cloud storage services. We run its A, B, C, D, and F workloads after loading the databases with one million records. Each record is 1KB by default. For each workload, we perform 500,000 operations.

Kernels. To characterize our framework, we combine into a benchmark several kernels that perform a random collection of reads, writes, inserts, and deletes to five persistent data structures: MArray, MList, FARArray, FArray, and FList. We list them in Table 1. We hand-wrote MArray, MList, and FARArray to ensure correct persistent operation. FArray and FList are functional data structures from the PCollections library [5], and inherently use persistent-safe structures.

Table 1. Persistent data structure description.

Data Structure & Description
Mutable ArrayList (MArray): ArrayList using copying to maintain persistency for inserts and deletes. Updates are in place.
Mutable LinkedList (MList): Doubly-linked list.
Failure-Atomic Region ArrayList (FARArray): ArrayList using failure-atomic regions to allow in-place insertions and deletions.
Functional ArrayList (FArray): Functional data structure that uses copying for data structure writes. Uses PCollection's PTreeVector class.
Functional LinkedList (FList): Functional data structure that uses copying for writes to the structure. Uses PCollection's ConsPStack class.

Table 2 shows the different AutoPersist-based NVM frameworks we use in this evaluation. *NoProfile* is AutoPersist without the profiling optimization described in Section 7. *T1X* is *NoProfile* but only using the initial tier compiler (T1X). *T1XProfile* is *T1X* plus collecting the profiling information described in Section 7. In other words, both *T1X* and *T1XProfile* are not using the optimizing compiler (Graal). *AutoPersist* is the full AutoPersist framework with all of its optimizations.

Table 2. Frameworks evaluated.

Framework	Description
NoProfile	AutoPersist without the profiling opt. of Section 7
T1X	*NoProfile* but only using the initial tier compiler (T1X)
T1XProfile	*T1X* plus collecting the profiling info of Section 7
AutoPersist	Complete AutoPersist
*Espresso**	Our implementation of Espresso [62]

We also created our own implementation of Espresso [62], which we call *Espresso**. *Espresso** requires the user to add markings identifying objects to allocate in NVM, to mark stores that must be flushed to NVM, and to insert memory fences. We have tried to faithfully implement *Espresso** in the most optimal way possible, including creating new compiler intrinsics and developing new JVM built-in calls to ensure that the *Espresso** markings execute as efficiently as possible.

9 Evaluation

9.1 Model Usability

A key benefit of AutoPersist is that it requires a developer to add only minimal markings in their program to ensure crash consistency. Specifically, the markings are: identifying the durable root set, inserting failure-atomic region entry and exit points, and marking unrecoverable fields for higher

Table 3. Number of markings for memory persistency.

Framework	Applications			Kernels					Total Markings without H2
	Func	JavaKV	H2	MArray	MList	FARArray	FArray	FList	
AutoPersist	4	6	6	1	1	5	1	1	19
Espresso*	55	45	N/A	49	48	63	47	14	321

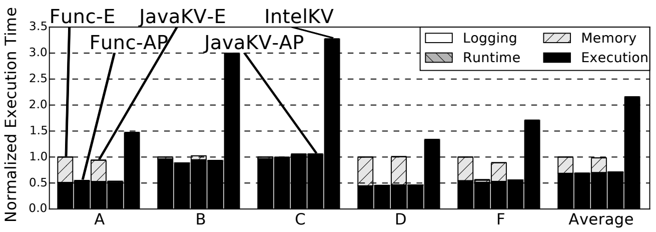

Figure 5. Key-value store execution time.

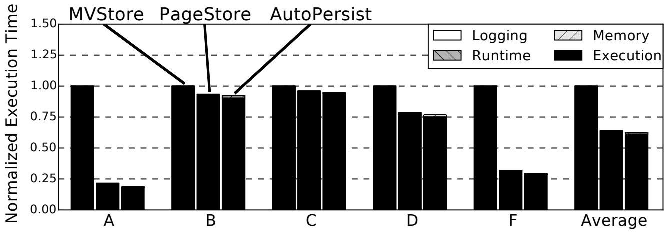

Figure 6. H2 execution time.

performance. This is in contrast to *Espresso**, which needs explicit markings for each persistent object allocation, cache line writeback to NVM, and fence [62].

Table 3 shows the number of markings added for each application when using AutoPersist and *Espresso**. Table 3 shows that AutoPersist only needs 25 markings in total (or 19 markings without H2). In contrast, when using *Espresso**, the programmer needs to add 321 markings in the programs to ensure crash consistency. Note that we did not implement a persistent version of H2 in *Espresso** due to the difficulty of implementing it correctly. In their paper, Wu et al. [62] claim to need to modify around 600 LoC to implement a persistent version of H2 in their framework, built on top of DataNucleus [1]. They also estimated that, if they had implemented their system directly on top of H2, they would have needed to modify over 3,000 LoC.

There is a significant difference in the number of markings required in the two frameworks. However, in our experience, even this difference does not do justice to the fact that we found it *much more difficult* to create a correct crash-consistent application in *Espresso**. Overall, using AutoPersist greatly reduces programmer effort and the likelihood of introducing performance or correctness bugs.

9.2 Analysis of the Key-Value Store

Figure 5 shows the execution time of different persistent key-value store backends while running YCSB. In the figure, the different versions of the Func and JavaKV backends are named as *{backend}-{framework}*, where framework can be *E* for Espresso* and *AP* for AutoPersist. We also show a bar for *IntelKV*. The execution time is normalized to *Func-E*. We break down the execution time into four categories which, from top to bottom, are: *Logging*, *Runtime*, *Memory*, and *Execution*. *Logging* is the time spent performing logging in failure-atomic regions. Note that it does not include the time spent executing CLWB or SFENCE instructions while performing this logging. *Runtime* is the time spent by the AutoPersist runtime ensuring that the transitive closure of the durable root set resides in NVM, and moving objects to NVM as necessary. It corresponds to the execution of the *makeObjectRecoverable* method (Algorithm 3). *Memory* is the overhead of executing CLWB and SFENCE instructions. Finally, *Execution* is the remaining execution time. Note that *Logging* and *Runtime* only apply to AutoPersist backends. Also, *IntelKV* cannot be broken down because it uses a C++ library that we cannot instrument; all its time is *Execution*.

Looking at the Average bars, we see that the execution time of *IntelKV* is 116% and 119% higher than of *Func-E* and *JavaKV-E*, respectively, which correspond to a previously proposed system. More importantly, the execution times of our *Func-AP* and *JavaKV-AP* backends are 31% and 28% lower than of *Func-E* and *JavaKV-E*, respectively.

The reason why *IntelKV* is substantially slower than the others is that, since the QuickCached application is written in Java and the pmemkv library in C++, the data objects must be serialized in order to pass them from QuickCached to the pmemkv library. For the backends implemented in pure Java, the data does not need to be serialized, as the non-volatile portion of the heap provides crash consistency.

AutoPersist significantly outperforms Espresso* due to having a practically negligible *Memory* time. This is because AutoPersist's runtime is able to limit the number of CWLBs when objects become reachable from the durable root set. Specifically, as AutoPersist is built into the JVM, it has precise knowledge of the address and layout of the objects. Hence, when objects become recoverable, it emits a single CLWB per cache line, reducing the total number of CLWBs. On the other hand, since *Espresso** adds cache line writebacks at the source code level, it does not have any information about the object's layout or alignment within cache lines. Hence, it must insert a CLWB for every object field to ensure that the object is entirely persistent. This is an inherent limitation of performing markings at the Java source code level. It is a strong argument for why, in managed languages such as Java, it is best to let the runtime decide when to emit cache line writebacks.

How much AutoPersist outperforms Espresso* is directly proportional to the number of insert and update operations

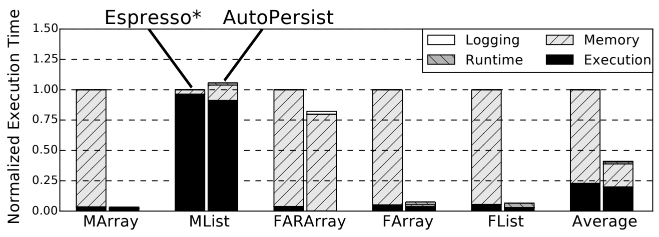

Figure 7. *Espresso** and *AutoPersist* kernel execution time.

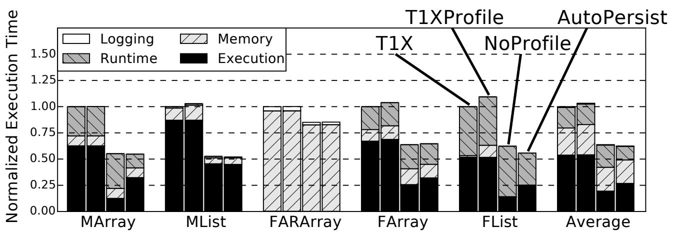

Figure 8. Kernel execution time with AutoPersist configs.

within the given YCSB benchmark. For instance, in the read-only *C* workload and read-mostly *B* workload, Espresso* performs about the same as AutoPersist. However, for workloads with writes, such as *A*, *D*, and *F*, AutoPersist is able to significantly outperform Espresso*.

Figure 5 also shows that the *Logging* and *Runtime* times in AutoPersist are negligible. Importantly, the *Runtime* overhead is negligible because of the efficiency of our algorithms. Finally, when using the same framework, the performance difference between *Func* and *JavaKV* is minimal. This is because both data structures are tree-based and have similar branching factors.

9.3 Analysis of the H2 Database

Figure 6 shows the execution time of the H2 database with the MVStore, PageStore, and AutoPersist storage backends when running the YCSB workloads. Recall that MVStore and PageStore are file-based backends, which we direct to use NVM as storage. MVStore and PageStore do not have *Memory* time because, rather than using CLWBs and SFENCEs, they persist data via file operations.

On average, the execution time using AutoPersist is 38% and 3% lower than using MVStore and PageStore, respectively. Similar to the key-value store applications, the execution time reductions of AutoPersist are higher when executing write-heavy workloads. Surprisingly, the PageStore storage engine significantly outperforms MVStore. As mentioned in Section 8, currently our AutoPersist H2 backend is based on the MVStore engine. We anticipate that we can achieve greater speedups by implementing an AutoPersist version of the PageStore backend.

9.4 Analysis of the Kernels

9.4.1 Execution Time

Figure 7 shows the kernel execution times for *Espresso** and *AutoPersist*. For each kernel, the bars are normalized to *Espresso**. The bars are broken down into the usual categories. We see that, on average, *AutoPersist* reduces the execution time by 59% over *Espresso**. The *AutoPersist* gains largely come from a large reduction in *Memory* time. This is because, as discussed in Section 9.2, AutoPersist inserts the minimal number of CLWBs necessary to ensure that objects reachable from the durable root set are persistent.

We see that the *AutoPersist* configuration of *FARArray* does not reduce the *Memory* time much. This is because, in this kernel, many CLWBs and SFENCEs are executed while performing logging. AutoPersist cannot easily reduce the number of such CLWBs and SFENCEs because a given log entry must be persisted before its program store can execute. *MList* has little *Memory* time because it does not need to perform many writes. AutoPersist increases the *Memory* time because it supports sequential persistency and, therefore, introduces more SFENCEs.

To highlight the benefits of optimizations within AutoPersist, Figure 8 compares the execution time of the kernels in the different AutoPersist frameworks: *T1X*, *T1XProfile*, *NoProfile*, and *AutoPersist* (Table 2). The bars are normalized to *T1X* and are broken down into the usual categories.

As shown in Figure 8, we find that, on average, *NoProfile* and *AutoPersist* reduce the execution time by 36% and 38% over *T1X*, respectively. This reduction is due to using the optimizing compiler, which reduces the *Execution* time. We also see that *T1XProfile* takes only a bit longer to execute than *T1X*, which shows that the overhead of our profiling in the baseline compiler is minimal.

Comparing *NoProfile* and *AutoPersist* shows the performance impact of our profiling pass. We see that, by eagerly allocating in NVM objects anticipated to become persistent, our pass reduces the *Runtime* by an average of 39%. However, the total execution time decreases by an average of only 2%. Nevertheless, we believe that, as NVM technologies improve, the amount of time needed to perform CLWBs and SFENCEs will decrease. Hence, it will be important to ensure that other bottlenecks, like runtime overhead, are minimized. Therefore, we believe that our profiling optimization will become more important.

9.4.2 Runtime Events

To further understand the behavior of AutoPersist, we profile the *NoProfile* and *AutoPersist* frameworks while running each kernel (Table 4). For *NoProfile*, Column 1 shows the number of objects allocated during execution; Column 2 shows the number of objects copied to NVM; and Column 3 shows the number of pointers updated as a result of the copies.

The rest of the columns show the impact of our profiling optimization in AutoPersist. Specifically, Column 4 shows the number of objects that are eagerly allocated in NVM. We see that our optimization allocates a large fraction of the objects eagerly. Columns 5 and 6 show the data corresponding to Columns 2 and 3. We see that our profiling optimization significantly reduces the number of objects to be copied and the number of pointers to be updated. Note that the FArray and FList kernels still perform many copies and updates. This is because some of their methods do not get recompiled by the optimizing compiler.

Table 4. *NoProfile* and *AutoPersist* event counts.

Kernel	*NoProfile*			*AutoPersist*		
	Obj Alloc (K)	Obj Copy (K)	Ptr Update (K)	NVM Alloc (K)	Obj Copy (K)	Ptr Update (K)
MArray	29.9	29.9	7.4	29.9	0	0
MList	22.5	22.5	7.4	22.5	0	0
FARArray	15.1	15.1	0	15.1	0	0
FArray	468.4	304.4	281.9	225.9	170.8	170.8
FList	11447.6	11440.1	11417.6	7548.4	3891.7	3884.1

It can be shown that the number of allocation sites in the source code that are profiled by our profiling pass ranges from 208 to 279 sites per kernel. Of those, only a small number are converted to eagerly allocate objects in NVM. Specifically, only 4 to 43 sites per kernel (on average, 15 sites per kernel) are converted. However, we believe that identifying such sites manually would be hard.

9.5 AutoPersist Runtime Overheads

The changes proposed in AutoPersist introduce two new overheads over normal Java execution: its augmented JVM bytecodes must perform extra actions, and the *NVM_Metadata* header adds extra memory overhead. The execution overhead of the extra actions is modest, in part thanks to the application of the biasing techniques described in QuickCheck [57] and implemented here. Such overhead appears within the *Execution* category of AutoPersist in Figures 5 to 8. Overall, it was shown in QuickCheck [57] that the resulting overhead of this effect is, on average, less than 10% of the total execution time.

The memory overhead of the larger header increases the memory consumption of the key-value store and the H2 database by an average of 9.4% and 1.6%, respectively. The overhead is greater for the key-value store than for H2 due to the relatively low branching factor within the B+tree nodes used in both the Func and JavaKV backends of the key-value store. Fortunately, this overhead is tolerable due to the large memory capacity that NVM can provide.

10 Related Work

Many NVM frameworks have been proposed [6, 20–23, 25, 35, 41, 48, 60, 62]. We describe their limitations in Section 3.1. While most existing frameworks are written in C or C++, Wu et al. [62] proposed a framework in Java called Espresso. In Espresso, like in previous frameworks, the user has to explicitly identify all non-volatile allocations, and perform the persist operations necessary for crash consistency. Section 9 compares AutoPersist and Espresso.

Many previous works propose both software [31, 36, 42, 46, 60] and hardware [28, 40, 55] techniques to limit the overhead of logging in atomic regions and transactions. We implement a simple undo log and leave more advanced implementations as future work.

Forwarding pointers and write barriers are used in concurrent and generational garbage collectors [15, 26, 29, 51, 59]. To our knowledge, this the first time that such runtime techniques have been applied to help create crash-consistent applications in NVM. This use case is different than GC, since the behavior is dictated by static user markers instead of the temporal behavior of GC. However, AutoPersist's forwarding pointers and barriers can be shared with GC when used in conjunction with advanced GC implementations.

Analysis techniques have been proposed to anticipate long living objects [17], to prevent wearout [11], and to predict when object resources can be reclaimed [33, 63]. We expect to further improve AutoPersist with such techniques.

Previous papers have proposed many persistency models [19, 31, 38, 41, 44, 50], which define how stores to NVM can be reordered both in software and hardware. AutoPersist's persistency model is described in Section 4.3. Currently, AutoPersist uses sequential persistency outside of failure-atomic regions and epoch persistency inside failure-atomic regions. However, more relaxed persistency models can also leverage our runtime reachability analysis.

Many persistent programming languages [12, 18, 34, 39, 54, 58] and implementations [13, 43, 47] were proposed before the introduction of byte-addressable NVM. These languages focus on attaining the *orthogonal persistency* defined by Atkinson and Morrison [14], where the persistency of an application is orthogonal to its design. AutoPersist's model is different in that it does not seek to attain complete orthogonal persistency. Furthermore, previous papers on orthogonal persistency describe approaches to optimize for a two-level storage model with orders of magnitude differences in performance, whereas AutoPersist targets byte-addressable NVM.

11 Conclusion

In this paper, we introduced *AutoPersist*, a new NVM framework where the programmer only needs to identify durable roots. We described its implementation as an extension of Java, including its support for transparent, efficient, and thread-safe object access and movement across NVM and DRAM. To evaluate the performance and programmability of AutoPersist, we used various applications running on Intel Optane DC persistent memory. We demonstrated that, with minimal programmer effort, AutoPersist significantly outperforms expert-marked Java NVM applications.

Acknowledgments

We thank our anonymous reviewers for their helpful comments. We also thank Sanjay Kumar and Intel for giving us early access to a server with Intel Optane DC persistent memory. This work was supported in part by the NSF under grants CCF-1527223 and CNS-1850317.

References

[1] Data Nucleus. http://www.datanucleus.org/

[2] H2 Database Engine. https://www.h2database.com

[3] Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[4] NVM Programming Model v1.2. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf

[5] PCollections. https://pcollections.org/

[6] Persistent Memory Development Kit. http://pmem.io/pmdk/

[7] Pmemkv: Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv

[8] QuickCached. https://github.com/QuickServerLab/QuickCached

[9] Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-\instruction-set-reference-manual-325383.pdf.

[10] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

[11] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 62–77. https://doi.org/10.1145/3192366.3192392

[12] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. 1982. PS-algol: An Algol with a Persistent Heap. *SIGPLAN Not.* 17, 7 (July 1982), 24–31. https://doi.org/10.1145/988376.988378

[13] Malcolm Atkinson and Mick Jordan. 2000. *A Review of the Rationale and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform.* Technical Report. Mountain View, CA, USA.

[14] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *The VLDB Journal* 4, 3 (July 1995), 319–402. http://dl.acm.org/citation.cfm?id=615224.615226

[15] David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops, OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES, WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003, Proceedings.* 466–478. https://doi.org/10.1007/978-3-540-39962-9_52

[16] Stephen Blackburn and John N. Zigman. 1999. Concurrency - The Fly in the Ointment?. 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.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 250–258. http://dl.acm.org/citation.cfm?id=648123.747394

[17] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 342–352. https://doi.org/10.1145/504282.504307

[18] Luc Bläser. 2007. Persistent Oberon: A Programming Language with Integrated Persistence. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–85.

[19] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

[20] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[21] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[22] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages,* Vol. 2, OOPSLA (2018), 153:1–153:22. https://doi.org/10.1145/3276523

[23] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[25] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/2907294.2907303

[26] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[27] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966–975. https://doi.org/10.1145/359642.359655

[28] K. Doshi, E. Giles, and P. Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 77–89. https://doi.org/10.1109/HPCA.2016.7446055

[29] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. ACM, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[30] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware Transactional Persistent Memory. arXiv:cs.DC/1806.01108

[31] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

[32] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[33] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 364–375. https://doi.org/10.1145/1133981.1134024

[34] Antony L. Hosking and Jiawan Chen. 1999. PM3: An Orthogonal Persistent Systems Programming Language - Design, Implementation,

Performance. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 587–598. http://dl.acm.org/citation.cfm?id=645925.671503

[35] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[36] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[37] Intel. 3D XPoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[38] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

[39] Mick Jordan and Malcolm Atkinson. 2000. *Orthogonal Persistence for the Java[Tm] Platform: Specification and Rationale.* Technical Report. Mountain View, CA, USA.

[40] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. https://doi.org/10.1109/HPCA.2017.50

[41] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[42] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[43] Brian Lewis, Bernd Mathiske, and Neal M. Gafter. 2001. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. In *Revised Papers from the 9th International Workshop on Persistent Object Systems (POS-9)*. Springer-Verlag, London, UK, UK, 18–33. http://dl.acm.org/citation.cfm?id=648124.747405

[44] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[45] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[46] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent Memory Transactions. arXiv:cs.DC/1804.00701

[47] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. 2001. Implementing Orthogonally Persistent Java. In *Revised Papers from the 9th International Workshop on Persistent Object Systems (POS-9)*. Springer-Verlag, London, UK, UK, 247–261. http://dl.acm.org/citation.cfm?id=648124.747395

[48] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[49] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[50] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[51] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. 2007. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007.* 159–172. https://doi.org/10.1145/1296907.1296927

[52] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (July 2008), 465–479. https://doi.org/10.1147/rd.524.0465

[53] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. ACM, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3124680.3124729

[54] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. 1993. The Design of the E Programming Language. *ACM Trans. Program. Lang. Syst.* 15, 3 (July 1993), 494–534. https://doi.org/10.1145/169683.174157

[55] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/3123939.3124539

[56] Thomas Shull, Jian Huang, and Josep Torrellas. 2018. Defining a High-level Programming Model for Emerging NVRAM Technologies. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. ACM, New York, NY, USA, Article 11, 7 pages. https://doi.org/10.1145/3237009.3237027

[57] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*. https://doi.org/10.1145/3313808.3313822

[58] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. 1993. Texas: An Efficient, Portable Persistent Store. In *Persistent Object Systems*, Antonio Albano and Ron Morrison (Eds.). Springer London, London, 11–33.

[59] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[60] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[61] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

[62] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 70–83. https://doi.org/10.1145/3173162.3173201

[63] Guoqing Xu. 2013. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 111–130. https://doi.org/10.1145/2509136.2509512