QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks

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 technology that provides near-DRAM performance and scalable memory capacity. To facilitate the usability of NVM, new programming frameworks have been proposed to automatically or semi-automatically maintain crash-consistent data structures, relieving much of the burden of developing persistent applications from programmers.

While these new frameworks greatly improve programmer productivity, they also require many runtime checks for correct execution on persistent objects, which significantly affect the application performance. With a characterization study of various workloads, we find that the overhead of these persistence checks in these programmer-friendly NVM frameworks can be substantial and reach up to 214%. Furthermore, we find that programs nearly always access exclusively either a persistent or a non-persistent object at a given site, making the behavior of these checks highly predictable.

In this paper, we propose *QuickCheck*, a technique that *biases* persistence checks based on their expected behavior, and exploits speculative optimizations to further reduce the overheads of these persistence checks. We evaluate QuickCheck with a variety of data intensive applications such as a key-value store. Our experiments show that QuickCheck improves the performance of a persistent Java framework on average by 48.2% for applications that do not require data persistence, and by 8.0% for a persistent memcached implementation running YCSB.

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

Keywords Java, Non-Volatile Memory, JIT Compilation

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313822

ACM Reference Format:
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), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3313808.3313822

1 Introduction

Recently, there have been massive technological advances towards providing fast and byte addressable Non-Volatile Memory (NVM), such as Intel 3D XPoint [38], Phase-Change Memory (PCM) [42, 52], and Resistive RAM (ReRAM) [5]. These memory technologies promise near-DRAM performance, scalable memory capacity, and data durability, which provides opportunities for boosting the performance of software systems and applications [19, 34, 64].

To enable applications to take advantage of NVM while also improving their programmability, many frameworks for NVM have been proposed, including those for C/C++ [3, 15, 16, 19, 20, 22, 24, 29, 33, 40, 47, 62] and Java [3, 55, 56, 64] applications. A trend in this research area is to identify and persist data structures automatically or semi-automatically, to relieve the burden on developers. Instead of having to either use custom persistent libraries or explicitly mark all actions needed for crash consistency, developers mark only a few core objects. Then, the runtime ensures that the transitive-closure of these core objects is stored in NVM and operated on in a crash-consistent manner.

While such an approach greatly reduces the load on programmers, it requires a large number of software checks, which we call *persistence checks*, to be inserted alongside object accesses. These checks guard runtime actions necessary for correctly handling persistent objects during execution. A persistence check's action may be *activated*, meaning that extra actions need to be performed, or *bypassed*, meaning that no extra work is needed. The overhead of these persistence checks can be substantial. This is because these checks must be inserted for both persistent and non-persistent objects, as the compiler does not know which objects will be persistent until execution time, and these checks must be conservative. Through our profiling with various Java-based workloads, we find that the overhead of persistence checks is 55.4% on average, and can be as high as 214%, for applications that do not require data persistence.

However, we also find that whether the check at a given code location is activated or bypassed is highly predictable. Specifically, we find that, over 99% of the time, a given site accesses exclusively either persistent or non-persistent objects. Given this insight, we propose to *bias* each persistence check site to match its expected behavior. We leverage the multi-tier execution of a Java Virtual Machine (JVM) to profile and categorize each persistence check into one of the four categories: *likely*, *unbiased*, *unlikely*, and *very_unlikely*, based on how likely its action will be activated. According to the persistence check state, the compiler decides what optimized code to generate. Options range from generating code to favor the likely persistence check execution path, to performing speculative optimizations that eliminate the check activation path entirely.

We develop our persistence check biasing technique named *QuickCheck* in a real JVM framework running on a server system with Intel Optane DC persistent memory [35]. We evaluate QuickCheck with a variety of applications. Our experimental results demonstrate that QuickCheck improves the performance of the persistent Java framework on average by 48.2% for applications that do not require data persistence, and by 8.0% for a persistent memcached implementation running the Yahoo! Cloud Serving Benchmarks (YCSB). Overall, we make the following contributions in this paper:

- We conduct a thorough study on the checking of persistent objects in NVM frameworks, and identify that the overhead of persistence checks is substantial.
- We provide a detailed characterization of the behavior of persistence checking on a diverse set of applications that require data persistence, and show that persistence checking is highly predictable.
- We propose QuickCheck, a persistence check biasing technique that uses profiling and speculation to reduce the overhead of persistence checks.

2 Background

2.1 Java Application Execution

Java programs are executed on top of a Java Virtual Machine (JVM) [44]. The JVM receives as input `.class` files consisting of JVM bytecodes and metadata about the location of methods and other classes. From this input, the JVM then generates machine code customized for the current execution environment's architecture.

Instead of generating machine code ahead-of-time before execution, most JVM implementations *dynamically* generate code throughout execution using Just-in-Time (JIT) compilation techniques. By performing JIT compilation, non-executed code paths do not need to be compiled, and the generated code can be tailored to be optimized for the application's current behavior.

To fully leverage the potential of JIT compilation, advanced JVM compilers are organized into multiple tiers,

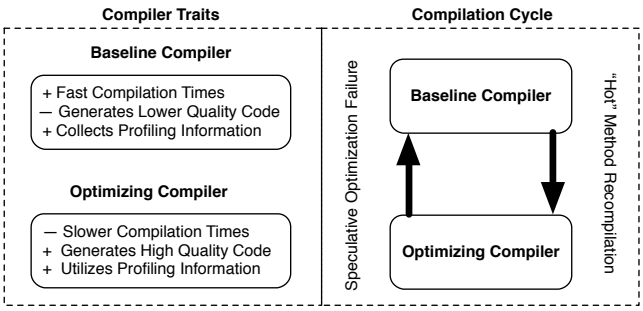


Figure 1. JVM compilation overview.

where each tier offers a different tradeoff between the time to generate code and the quality of code generated. Figure 1 provides a high-level overview of JVM compilation. The initial compiler, commonly known as the *baseline compiler*, generates code quickly. However, the code is not efficient. The "hot" methods in which most of the execution time is spent are later recompiled by a more advanced compiler, which in the figure we call the *optimizing compiler*. The optimizing compiler produces high-quality code. However, the code takes longer to generate.

An optimization commonly performed during multi-tiered compilation is to perform *speculative optimizations* within the optimizing compiler. Instead of generating code which can correctly handle all corner cases, the optimizing compiler chooses to speculate on which behaviors will be encountered during runtime, and only generates code that covers these behaviors. By performing this speculation, the optimizing compiler is able to better optimize the code paths. However, if an unexpected behavior is encountered, i.e., a *mispeculation* occurs, the optimized code will not be able to handle it correctly. In this case, the execution is transferred back to the code generated by the baseline compiler, which handles all corner cases. This process of transferring code execution back to a more conservative version of generated code is known as *on-stack-replacement* [31] fallback.

Performing an on-stack-replacement fallback on a mispeculation can be an expensive operation. Therefore, the optimizing compiler must be careful in choosing what speculative optimizations to perform. To assist with this process, multi-tiered compilers collect profiling information during the initial execution phase, which the optimizing compiler can later use to guide its speculation choices. The baseline compiler instruments its generated code with profiling metrics which are dynamically updated during execution. After that, when code is recompiled by the optimizing compiler, this collected profiling information is utilized. Profiling information has been shown to accurately predict the behavior of later executions and help the optimizing compiler improve the application performance [17, 18, 26, 30, 48].

2.2 Non-Volatile Random-Access Memory

Byte addressable NVM has performance characteristics that are roughly similar to DRAM, but has a much higher density,

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 memory system consisting of both types of memory. Hybrid systems typically have a unified virtual address space, allowing applications to allocate data to either volatile or non-volatile memory based on the needs of applications.

While NVM provides non-volatility, volatile caches still exist between the processor and NVM. This means that care must be taken to ensure that a store from the processor becomes persistent. In other words, one must ensure that the stored value is propagated to NVM, rather than remaining dirty within the cache hierarchy.

Without special support, the order in which stores reach the NVM and are made persistent depends on the order in which they are evicted from the cache hierarchy. To ensure that a store reaches NVM deterministically, values must be flushed from the cache via explicit instructions. In addition, fences must be inserted to guarantee the correct ordering. Recently, x86-64 processors have added a new cacheline-writeback instruction (*CLWB*) [37] to support writing back a cache line to NVM without flushing the cache line. However, multiple such cacheline writebacks are allowed to be reordered by the processor unless fences are placed in the code. Specifically, to guarantee completion of a *CLWB*, the x86-64 store fence (*SFENCE*) instruction must be used.

2.3 Industrial NVM Frameworks

The Storage Networking Industry Association (SNIA) has been working to standardize interactions with NVM. They have created a low-level programming model [58] meant to be followed by device driver programmers and low-level library designers. In addition, an open source project led by Intel has been created to provide application developers a higher-level toolset that is compliant with this device-level model. This project has resulted in the development of the Persistent Memory Development Kit (PMDK) [3], 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 durable data in their code with pragmas. As an alternative, PMDK also contains a few library data structures, such as a durable primitive array and map, with the necessary persistent pragmas built already into the code.

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

2.4 Academic NVM Frameworks

In addition to the industrial efforts, academia has also proposed several frameworks for NVM [15, 16, 19, 20, 22, 24, 29, 33, 40, 47, 55, 56, 62, 64]. The level of support provided by these frameworks varies. Some frameworks [19, 47] provide a similar level of abstraction as PMDK and expect the user to specify all persistent objects, while others try to offload much of the overhead of creating persistent applications off the programmer. One strategy to simplify the programming model is to require the user to only identify persistent epochs [15, 16, 22, 29, 33, 62]. Given these epochs, the framework will include the runtime support necessary to ensure crash-consistency at region granularity. This simplifies the programming model from the user perspective, since when using these frameworks, users do not have to explicitly perform cacheline flushes or place memory fences to ensure crash consistency.

Another strategy to improve programmability is to forgo libraries and build NVM support directly into the language [15, 24, 40, 55, 56, 64]. These approaches modify both the compiler and runtime to include support for persistent features. With language-level support, these frameworks provide better portability and performance. Also, since the compiler is now aware of persistent features, it can perform advanced optimizations to limit the overhead of persistency.

With runtime support, it is also possible for users to not have to identify all persistent objects [15, 55, 56]. Instead, the runtime can dynamically detect which objects must be stored in NVM for recovery purposes. Leveraging the runtime to identify which objects must be made persistent even further reduces the amount of persistent markings needed by the user, which both improves programmability and reduces the likelihood of having programmer bugs.

2.5 Programmer-Friendly NVM Framework

As discussed, a recent trend for NVM frameworks is to relieve the developer burden by identifying persistent data structures automatically or semi-automatically. As programmer productivity becomes a first-class metric, we believe this trend will continue and more programmer-friendly NVM frameworks will be introduced. An ideal programmer-friendly NVM framework should minimize both the programmer effort needed to create a persistent program and likelihood of having programmer bugs. To highlight the features of these frameworks and their implementations, we outline the compiler changes and runtime support needed by the *AutoPersist* framework [56] as follows. However, any programmer-friendly framework must undergo similar changes. In the rest of the paper, we collectively refer to these programmer-friendly frameworks as friendly NVMs (fNVMs).

2.5.1 Managing Persistent Objects

A key goal of fNVMs is to limit the amount of objects that must be manually identified as needing special treatment for crash-consistency. Instead of requiring the user to identify all objects that should reside in NVM, these frameworks should require the user to label only a few core objects which are

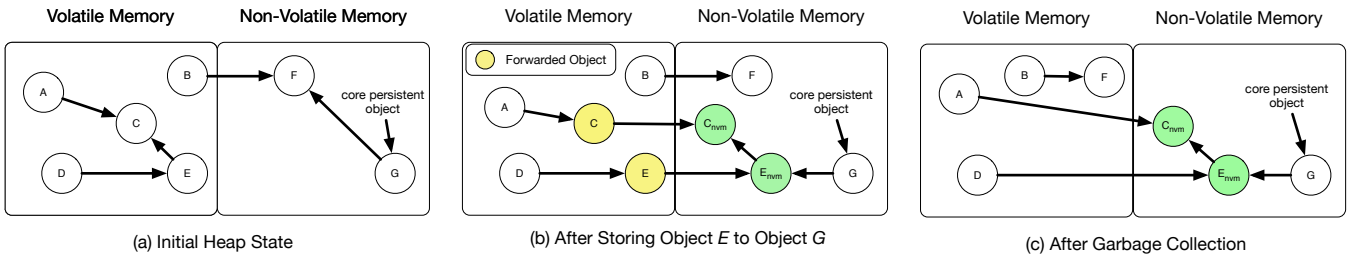


Figure 2. Changing heap state to ensure correct persistent behavior.

then used as the basis for recovery. We call these core objects the *core persistent object set*.

Given the core persistent object set, AutoPersist's runtime ensures that the transitive closure of the core persistent set is also placed in NVM. AutoPersist inserts runtime checks to dynamically uphold this mandate.

Figures 2 (a)-(c) show how the heap changes as an application executes to ensure crash-consistency. 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 within the core persistent object set. Hence, it must be in NVM. Object *F* is reachable from *G* and must also be in NVM. Because *B* is not reachable from the core persistent object set, it does not need to be in NVM.

In Figure 2(b), the program has changed the $G \rightarrow F$ pointer to $G \rightarrow E$. This causes the runtime to intervene and move objects *E* and *C* to NVM, since they are now reachable from the core persistent object set. The new versions of *E* and *C* are colored green in the figure. Because eagerly updating all pointers to *E* and *C* has significant overheads, AutoPersist uses the original *E* and *C* objects as *forwarding objects* and lazily updates pointers in the volatile heap. In the figure, the forwarding objects are colored yellow.

Finally, during a garbage collection (GC) cycle in AutoPersist, all pointers are updated and the forwarding objects are collected. Figure 2(c) shows the state of the heap after a GC cycle has occurred.

In addition to ensuring all objects reachable from the core persistent object set reside in NVM, fNVMs' runtimes must also guarantee that stores to these objects are made persistent in the proper order without requiring explicit user markings. In AutoPersist, this is accomplished by adding additional behaviors onto the end of store operations to persistent objects. Since objects become persistent dynamically throughout execution, it is unknown which stores must include the extra persistent behavior at compilation time. Therefore, AutoPersist includes a check for whether an object is persistent or not to guard the persist operations.

2.5.2 Failure-Atomic Region Support

fNVMs also add support for failure-atomic regions. Within a failure-atomic region, all stores to objects reachable from the core persistent object set are not part of the recoverable program state until the region has ended. When a failure-atomic region is exited, all stores within the region are atomically added to the recoverable program state. In AutoPersist, the runtime tracks stores to objects reachable from the core persistent object set within a failure-atomic region. It ensures that they become a part of the crash-recoverable state only at the region end, and in an atomic manner.

3 Limitations of Programmer-Friendly NVM Frameworks

As described in Section 2.5, fNVMs have the potential to dramatically ease the programming burden of creating a persistent application. However, to facilitate the widespread acceptance of these frameworks, we need to ensure their performance is close to other manually optimized NVM programs. One significant overhead is the persistence checks required by fNVMs around accesses to both persistent and non-persistent objects. Before we discuss the proposed solution to reduce the persistence checking overhead, we first describe how these frameworks use checks to guard much of the runtime behavior for correct persistent execution.

3.1 Store Field Persistence Checks

As described in Sections 2.5.1 and 2.5.2, many persistence runtime actions in fNVMs are dependent on whether an object is reachable from the core persistent object set or not. Since these actions can be very expensive, it is beneficial to ensure that they are only enabled when absolutely necessary. Because of this, fNVMs traditionally include many checks, which we call *persistence checks*, to guard runtime actions needed for correct persistent execution. Persistence checks query the object's state to determine whether the action is needed and must be *activated*, or if the action can be *bypassed*.

We show how stores to fields are performed in AutoPersistin Algorithm 1. Without persistence support, the function `storeField(holder,field,value)` writes value *V* into field *F* of the holder object *H* (line 18). Now, in AutoPersist, both before and after writing *V* into field *F*, persistence checks must be included to ensure the runtime performs all actions needed when handling persistent objects.

The persistence check code before the write (lines 2 to 17) first checks whether the guarded action should be activated or not. This is done by including a persistence check (line 3) to determine if *H* is either forwarded or persistent. If *H* is a forwarded object, its forwarding address will point to a persistent object. As explained in Section 2.5.1, forwarded

Algorithm 1 store operation with persistence checks.

```
1:  procedure storeField(holder, field, value)
    [Start Persistence Check Code]
2:      [Persistence Check]
3:      if isPersistent(holder) or isForwarded(holder) then
4:          [Persistence Check's Guarded Action]
5:          if isForwarded(holder) then
6:              holder = getForwardedAddr(holder)
7:          end if
8:          if isForwarded(value) then
9:              value = getForwardedAddr(value)
10:         end if
11:         if !isPersistent(value) then
12:             move value to NVM
13:         end if
14:         if inFailureAtomicRegion(tid) then
15:             logStore(holder, field)
16:         end if
17:     end if
    [End Persistence Check Code]
18:     writeField(holder, field, value)
    [Start Persistence Check Code]
19:     [Persistence Check]
20:     if isPersistent(holder) then
21:         [Persistence Check's Guarded Action]
22:         cachelineWriteback(holder, field)
23:         if !inFailureAtomicRegion(tid) then
24:             persistFence()
25:         end if
26:     end if
    [End Persistence Check Code]
27: end procedure
```

objects act as temporary pointers to persistent objects before all pointers are updated during a GC cycle. If the persistence check is true, then the action it is guarding will be executed (lines 4 to 17). This code first retrieves the current location of H. This is performed by checking whether the holder object has been forwarded (line 5) or not. If so, the current holder object's location must be retrieved (line 6). AutoPersist has an extra object header word to allow for the fast retrieval of persistent state information and storing forwarding addresses. Next, the runtime must ensure that V is also persistent. As with object H, this check must first retrieve V's current location (lines 8 to 10), and then afterwards must check whether V is persistent (line 11). If V is not persistent, then the runtime must move V and its transitive closure to NVM. Lines 14 to 16 perform the logging needed to maintain the appearance of atomicity within failure-atomic regions.

The persistence check's guarded action after the original write (lines 21 to 26) ensures that the proper persistency measures are taken. If H is persistent, the field must be written back to NVM using a *CLWB* (line 22). In addition, if execution is currently not within a failure-atomic region, then a *SFENCE* must be inserted to ensure data consistency.

Note that while there are many checks within a persistence check's guarded action, we only consider the checks on lines 3 and 20 to be *persistence checks*, as these are the checks which guard runtime actions needed when interacting with persistent objects.

Algorithm 2 load operation with a persistence check

```
1:  procedure loadField(holder, field)
    [Start Persistence Check Code]
2:      [Persistence Check]
3:      if isForwarded(holder) then
4:          [Persistence Check's Guarded Action]
5:          holder = getForwardedAddr(holder)
6:      end if
    [End Persistence Check Code]
7:      value = readField(holder, field)
8:      return newValue
9:  end procedure
```

3.2 Load Field Persistence Checks

In addition to modifying object store procedures, fNVMs must also add a persistence check to load procedures to ensure pointers do not refer to forwarded objects. Algorithm 2 shows how field loads are performed in AutoPersist.

Without persistence support, `loadField(holder,field)` loads the value V from field F of holder object H and returns V. In AutoPersist, before this load can occur, a persistence check is inserted (line 3) to guard retrieving the current location of H (line 5). As described in Section 3.1, only objects which are persistent can have forwarded objects.

4 Characterizing Persistence Checks

As shown in Section 3, fNVMs require many persistence checks to guard runtime actions. In this section, we first evaluate the overhead of these persistence checks. Afterwards, we profile the activation behavior of persistence checks across various persistent applications.

4.1 Overhead of Persistence Checks

As shown in Algorithms 1 and 2, persistence checks are used frequently in fNVMs. The actions guarded by these checks are only activated if the holder object being accessed is reachable from the core persistent object set. Otherwise, for non-persistent holder objects, the actions are bypassed. To evaluate the cost of persistence checks guarding bypassed actions, we run the DaCapo [12] and Scala DaCapo Benchmark Suites [54]. As these benchmarks do not have persistent markings, all objects will be non-persistent by default. Hence, they provide an excellent opportunity to evaluate the overheads of persistence checks with bypassed actions.

We conduct the experiments on AutoPersist [56]. More details about the implementation are in Section 7.1. We evaluate two configurations: *NoChecks*, which does not have persistence checks, and *WChecks*, which includes the persistence checks described in Section 3. We run each benchmark multiple times to warm up the system before measuring the execution time.

Figures 3 and 4 show the performance of the two configurations. In the worst performing benchmark, Scala DaCapo's

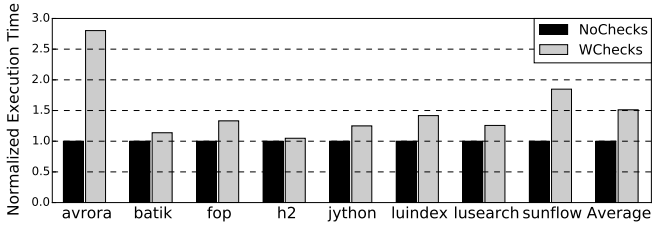


Figure 3. DaCapo persistence check overhead.

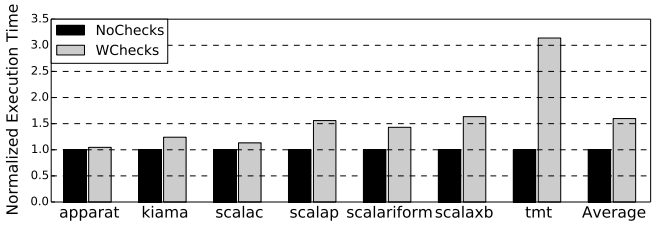


Figure 4. Scala DaCapo persistence check overhead.

	Dynamic Check Behavior			Classification of Check Sites		
Data Structure	# Checks Seen (M)	# Checks w/ Actions Activated (M)	% Checks w/ Actions Activated	% Sites w/ Actions Always Activated	% Sites w/ Actions Sometimes Activated	% Sites w/ Actions Never Activated
FHMap	2708.4	6.4	0.24	0.03	0.05	99.86
PBTree	2729.2	10.9	0.40	0.27	0.08	99.57
HBTree	2730.9	5.6	0.20	0.09	0.02	99.87

Table 1. Persistence check statistics when running YCSB workloads.

`tmt` benchmark, *WChecks* is 214% slower than *NoChecks*. On average, the *WChecks* configuration is 51.1% and 59.7% slower than the *NoChecks* configuration for the DaCapo and Scala DaCapo benchmarks, respectively. This overhead is significant because no useful work is performed, since all persistence checks guard bypassed actions. While fNVMs can significantly simplify the process of creating persistent programs, their overhead must be minimal, compared to other NVM frameworks. Clearly, the overheads of persistence checks must be drastically reduced before fNVMs can gain widespread acceptance.

4.2 Persistence Check Activation Behavior

As described in Sections 3.1 and 3.2, in AutoPersist, the actions guarded by persistence checks are activated only when the holder object is either a persistent object or a forwarded object. While it is possible in theory for each persistence check's action to vary between being activated and bypassed throughout program execution, it is well known that many program characteristics are highly consistent throughout execution in Java and other languages. They include branching behaviors, virtual call dispatch targets, and dynamic object property lookup offsets.

To confirm whether the behavior of the actions guarded by persistence checks are as consistent as other program features, we monitor the behavior of persistence checks across three persistent applications. Our persistent applications are three versions of a persistent key-value store. Each version is based on QuickCached [4], a pure Java implementation of memcached [1], and uses a different persistent data structure internally for its key-value storage. Specifically, *Functional HashMap (FHMap)* uses a functional hash map as its backend; *Persistent B+Tree (PBTree)* uses an entirely persistent B+ tree as its backend; and *Hybrid B+Tree (HBTree)* uses a B+ tree where only the leaf nodes are persistent. Section 7.2 contains more details about the persistent applications.

We run each persistent application with the Yahoo! Cloud Serving Benchmarks [23] and show the aggregated results. For each run, we monitor the dynamic number of persistence checks encountered, and the number of persistence checks guarding activated actions. In addition, we categorize each persistence check site into one of three categories based on its behavior: check sites with actions that are always activated, check sites with actions that are sometimes activated, and check sites with actions that are never activated.

We show the profiling results in Table 1. From the results we see two main traits. First, persistence checks have predictable behavior; very few checks guard actions which are only sometimes activated. Second, and most noticeably, actions guarded by persistence checks are activated very rarely. In each application, we find that over 99% of the checks guard bypassed actions. One reason for this is that these actions are never activated for non-persistent objects. However, another important reason why the persistence check activation bypass rate is so high is that the checks described in Section 3.2, which are placed before object loads, are rarely activated, regardless of whether handling persistent or non-persistent objects. This is because these actions are only activated if the pointer is to a forwarded object. As explained in Section 2.5.1, forwarded objects exist only temporarily before a garbage collection cycle occurs. The results shown Table 1 indicate that forwarded pointers are seldomly used.

Overall, our results confirm that the behavior of persistence checks is highly predictable and suggest that, in persistent applications, the actions guarded by persistence checks are rarely activated.

5 Main Idea

In the previous section, we discovered two traits of persistence checks: they impose significant overheads and the activation behavior at a given check site is highly predictable. Based on these two traits, we propose to transform individual

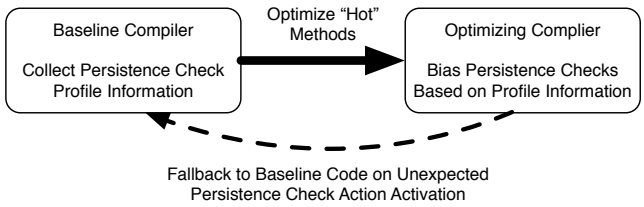


Figure 5. Overview of QuickCheck.

persistence checks dynamically to optimize for the common case expected at each check site. In addition, for persistence checks which guard actions that are predicted to never be activated, we propose to speculatively remove the actions guarded by the check. We name our solution QuickCheck. In the following subsections, we describe its technical details.

5.1 Overview of QuickCheck

To minimize the overhead of persistence checks, we propose to *bias* each check to be specialized for its common case. Specifically, we propose to have two phases of execution for each method, namely, a profiling phase where profiling information is collected for each persistence check, and a biasing phase where each check is transformed to be optimized for its expected common case.

Given that modern JVM implementations already compile "hot" code regions multiple times as discussed in Section 2.1, QuickCheck can use these multiple code generation phases as its profiling and biasing phases. Figure 5 shows how QuickCheck can be added to the traditional JVM compilation phases. Initially, when code is generated by the baseline compiler, we augment the compiler to also generate profile information about each persistence check site. For each persistence check site, we record the number of times the check's action is activated and bypassed.

After that, when the profiled method is recompiled by the optimizing compiler, the optimizing compiler uses the persistence check profile information to generate optimized code. Specifically, QuickCheck directs the compiler to perform simple optimizations such as deciding if the action guarded by the check should be on the main execution path or should be sunk to the end of the method. To further improve performance, QuickCheck can also direct the compiler to perform *speculative optimizations* to further reduce the overhead of highly predictable checks. For persistence checks guarding actions which are extremely unlikely to be activated, our modified compiler generates code which assumes the action guarded by the check will never be activated. If this assumption is false, then QuickCheck triggers a deoptimization to the baseline compiler when this action is activated. This fallback to the baseline compiler on an unexpected activation is represented by the dashed line in Figure 5.

5.2 Persistence Check Biasing Strategies

As discussed, the optimizing compiler can use the persistence check profile information collected during warm-up to generate better code. In this section, we describe how to bias the generated code. We divide the likelihood of a check's guarded action being activated into two categories: *likely* and *unlikely*. Figures 6 (a) and (b) show how QuickCheck generates code in these two cases. Figure 6 (a) shows the code for persistence checks whose guarded actions are likely to be activated. On line 1, the persistence check is performed. Line 2 contains a conditional branch to the post-action code. The branch is taken if the persistence check guards a bypassed action. Otherwise, if the branch is not taken, execution falls through to line 3 and performs the guarded action. Finally, the post-action code is placed after the guarded action routine (line 5). Given that it is likely that the persistence check guards an activated action, it is desirable to have the routine on the fall-through path of the conditional branch, as processors initially predict forward branches as not-taken [37].

Figure 6 (b) shows how QuickCheck generates code for persistence checks whose guarded actions are unlikely to be activated. On line 2, the conditional branch jumps to the guarded action routine on line 6 if the action is activated. The fall-through code is the code after the guarded action. Since the guarded action is unlikely to be activated, it is beneficial to move the routine off of the main execution path down to the end of the method. This improves instruction cache performance by ensuring that code unlikely to execute will not interrupt the spatial locality of code likely to execute.

5.3 Speculatively Removing Action Routines

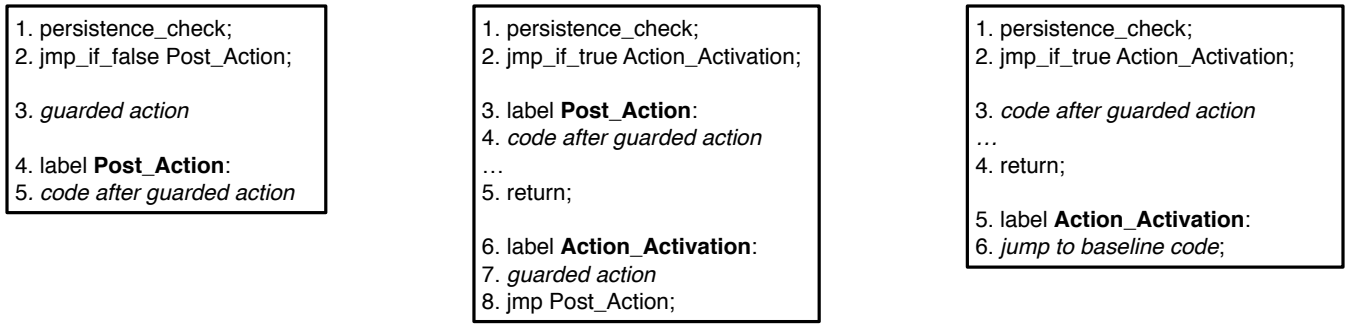
As shown in Section 4.2, over 99% of persistence checks guard actions that are *never* activated. To further reduce overhead, QuickCheck introduces the *very_unlikely* category for actions extremely unlikely to be activated.

In this case, instead of simply moving the guarded action routine off the main execution path, QuickCheck *speculatively removes* it. In the rare case that a check categorized as *very_unlikely* needs to be followed by the execution of the guarded action, a deoptimization occurs and execution is transferred to the baseline compiler.

Figure 6 (c) shows the code generated by QuickCheck. On line 2, there is a conditional branch which is taken when the persistence check's guarded action is activated. However, instead of the guarded action routine being the destination of the conditional branch, now, as shown on line 6, execution is transferred to the code generated by the baseline compiler.

Because of the substantial overhead of transferring execution to a different code tier, performing this speculative optimization is only beneficial if it is rarely incorrect. Hence, QuickCheck only performs it if the profile predicts that the persistence check's guarded action will never be executed.

This speculative optimization provides further cache performance improvements over Figure 6 (b), by not only improving spatial locality, but also eliminating the code bloat of routines that are unlikely to execute. More importantly, this speculative optimization helps improve the efficiency of compiler optimization passes by eliminating code that can



(a) Likely activation of action guarded by persistence check

(b) Unlikely activation of action guarded by persistence check

(c) Very unlikely activation of action guarded by persistence check

Figure 6. Code generation strategies for persistence checks in QuickCheck.

interfere with the optimizations. As shown in Algorithms 1 and 2, the code guarded by persistence checks includes many memory accesses, calls to the runtime, and even memory fences. By removing this code, the compiler is now able to perform more aggressive code reordering. In addition, since the compiler's aliasing information is now unobscured by runtime calls, many additional optimizations, such as common subexpression elimination, are also more effective.

6 QuickCheck Implementation

In this section, we explain how we apply QuickCheck to AutoPersist. We first describe how our baseline compiler is modified to collect persistence check profiling information. Afterwards, we discuss how this profiling information is used to determine the activation bias of each persistence check and guide code generation in the optimizing compiler.

6.1 Persistence Check Profile Collection

As shown in Figure 5, QuickCheck needs the baseline compiler to collect profiling information which is used in later phases to determine the activation bias of persistence checks. To accomplish this, we modify the baseline compiler to record profiling information at each persistence check.

To store our profiling information, we expand the per-method profiling present in the baseline compiler to include two new counters for each persistence check site: an `activated` counter and a `bypassed` counter. We increment the proper counter depending on whether the check's guarded action is activated or bypassed.

While two persistence checks surround each write operation (see Section 3.1), we only need to profile the persistence check before the write. This is because if the pre-write persistence check's guarded action is executed, then the post-write check's guarded action is also executed, as can be seen in Algorithm 1. While the check at line 3 determines if the object is either forwarded or persistent, and the check at line 20 only queries if the object is persistent, note that these are equivalent. This is because if the pre-write check's guarded action is executed and the object is forwarded, then the persistent object will be retrieved from its forwarding address (line 6). Hence, the object will be persistent in the check at line 20. Having only one profile site per store helps limit the memory and computation overhead of recording the persistence check profiling information.

6.2 Compiler Optimization Pass

6.2.1 Calculating Persistence Check Bias

Once a method is "hot" enough to be recompiled, the optimizing compiler reads each persistence check profile's `activated` and `bypassed` counters to determine the activation rate of its guarded action. Based on this activation rate, we then categorize checks into four states for biasing:

- ***likely:*** Action activated over 95% of the time.
- ***unbiased:*** Action activated 5%–95% of the time.
- ***unlikely:*** Action activated less than 5% of the time.
- ***very_unlikely:*** Action not activated during profiling.

Once the biasing state of each persistence check is determined, the optimizing compiler can generate code which is optimized for the expected behavior. In the following, we describe how the persistence checks added to both load and store operations are optimized based on their biasing state.

6.2.2 Load Operation Persistence Check Generation

As shown in Section 3.2, all operations that load a value from the heap must have a persistence check beforehand to check for forwarding pointers. Based on the check's biasing state, the optimizing compiler biases the persistence check branch shown in line 3 of Algorithm 2. Specifically, for persistence checks in the *likely*, *unbiased*, and *unlikely* states, QuickCheck adjusts the weight of the true branch being executed to 95%, 50%, and 5%, respectively.

AutoPersist uses Graal as its optimizing compiler [21]. Within Graal, each branch is represented as a node containing a configurable parameter denoting its true branch weight. Subsequent compiler optimization passes then use this branch weight to determine whether the true branch should be present or not in the main execution path. Note

Algorithm 3 load with *very_unlikely* biased check.

```
1: procedure LOADFIELD(holder, field)
   [Start Persistence Check Code]
2:     if isForwarded(holder) then
3:         [Misprediction Deoptimization]
4:     end if
   [End Persistence Check Code]
5:     value = readField(holder, field)
6:     return newValue
7: end procedure
```

that for other popular compilers such as LLVM, HotSpot C2, and GCC, a similar mechanism also exists.

For persistence checks in the *very_unlikely* state, QuickCheck sets the weight of the true branch being executed to 1% and also changes the true branch execution path to trigger a deoptimization. Algorithm 3 shows a simplified version of this persistence check. If line 3 is reached, then a signal is raised. We register this point has a *deoptimization point*, where execution transfers to the baseline compiler. To catch raised signals, we modify the runtime to install a handler which performs an on-stack-replacement [31] to transfer execution to the baseline compiler in the event of a raised signal.

6.2.3 Store Operation Persistence Check Generation

As shown in Algorithm 1, stores originally have persistence checks both before and after the write operation. For checks in the *likely*, *unbiased*, and *unlikely* state, we adjust the weight of the true branches like in Section 6.2.2. However, instead of biasing all the branches within the guarded action code, we only bias the branches on lines 3 and 20 of the algorithm. This is because these are the branches which determine whether the persistence check's action is activated or not. We choose not to profile the behavior of internal branches in the guarded action code due to the code's low activation rate, as shown in Section 4.2.

For *very_unlikely* biased persistence checks, we only generate a persistence check before the write. This is because the post-write check's action can only be activated if the pre-write check's action is activated. However, if the pre-write check's action is activated, then execution will be transferred to the baseline code and the subsequent code generated by the optimizing compiler will not execute.

Algorithm 4 shows the store operation when the persistence check is biased to the *very_unlikely* state. Now, on line 2, the persistence check guards a deoptimization point. The compiler sets the weight of the true path to be 1% so line 3 is removed from the main execution path. Also, as with *very_unlikely* load operation, line 3 is registered as a deoptimization point.

7 Environmental Setup

7.1 System Platform

Compiler Platform. We implement QuickCheck on top of AutoPersist [56]. AutoPersist uses a modified version of the

Algorithm 4 store with *very_unlikely* biased check.

```
1: procedure STOREFIELD(holder, field, value)
   [Start Persistence Check Code]
2:     if isPersistent(holder) or isForwarded(holder) then
3:         [Misprediction Deoptimization]
4:     end if
   [End Persistence Check Code]
5:     writeField(holder, field, value)
6: end procedure
```

Maxine Java Virtual Machine (JVM) [63] version 2.0.5. Maxine is an open-sourced research JVM which facilitates the fast prototyping of new features while achieving competitive performance. We modify Maxine's baseline compiler (T1X) to collect the profiling information described in Section 6.1 and modify Maxine's optimizing compiler (Graal) [21] to use our persistence check biasing techniques. In addition, we augment Maxine to include the handlers needed to handle the mispeculation of persistence checks biased to the *very_unlikely* state.

Server Configuration. We use a development platform with 128GB of Intel Optane DC persistent memory [35] and 324GB of DDR4 DRAM. The server contains two 24-core Intel® second generation Xeon® Scalable processors (codenamed Cascade Lake) and runs Fedora 27 on Linux kernel 4.15. In all of our experiments, we set up our framework to reserve 20GB for each of the volatile and non-volatile heap spaces. To create the non-volatile heap, we use `libpmem` [3] to map a portion of the persistent address space to the application's virtual address space. After that, via the Direct Access (DAX) protocol, applications can directly interact with the Intel Optane DC persistent memory. We use cacheline writebacks (`CLWB`) and store memory fences (`SFENCE`) to persist values.

7.2 Applications

Java Benchmark Suites. To evaluate the effectiveness of QuickCheck in reducing the overhead of persistence checks guarding bypassed actions, we run most of the DaCapo [12] and Scala DaCapo [54] benchmark suites on our infrastructure. Both benchmarks suites are commonly used to evaluate the performance of JVM implementations. To measure the optimal performance of each benchmark, we run each benchmark several times (the same warm-up counts as used in [27]) before measuring the execution time.

Real-World Applications. To evaluate the effectiveness of QuickCheck on real persistent applications, we implement a persistent version of memcached using our framework. Specifically, we modify QuickCached [4], a pure Java implementation of memcached, to use persistent data structures internally for its key-value storage and compare with the state-of-the-art solutions. The different storage engines are described below:

- **IntelKV.** To test against the state-of-the-art design, we use Intel's `pmemkv` library [36] along with its Java bindings

Data Structure & Description
Mutable ArrayList (MArray): ArrayList using copying to maintain persistence 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 all writes to the structure. It uses PCollections' PTreeVector class.
Functional LinkedList (FList): Functional data structure that uses copying for all writes to the structure. It uses PCollections' ConsPStack class.

Table 2. Description of persistent data structures.

as one of our backends, which we call *IntelKV*. This backend uses pmemkv's *kvtree3* storage engine, which consists of a hybrid B+ tree written in C++ using the PMDK library version 1.5. Similar to existing work [49], in their implementation, only the leaf nodes are in persistent memory. Note that the *IntelKV* backend does not need the support of our custom JVM implementation and hence runs on an unmodified JVM.

- **Functional Hash Map (FHMap).** It uses the HashMap implementation from PCollections library [2], and is annotated with the persistent markings required for our modified JVM backend.
- **Persistent B+ Tree (PBTree).** This backend uses a pure Java version of the IntelKV B+ tree, along with the persistent markings required for our modified JVM backend.
- **Hybrid B+ Tree (HBTree).** This backend uses the same data structure as PBTree. However, like IntelKV, it only persists the leaf nodes instead of the entire data structure.

To evaluate the performance of these backends, we use the Yahoo! Cloud Serving Benchmark (YCSB) [23], a benchmark suite commonly used to evaluate the performance of cloud storage services. We run its A, B, C, D, and F workloads after populating the key-value store with one million key-value pairs (each pair is 1KB by default). For each workload, we perform five hundred thousand operations.

Kernel Applications. To isolate the behavior of our framework from large applications, we also create a benchmark which performs a random collection of reads, writes, inserts, and deletes against five typical persistent data structures. We list them in Table 2. We hand-wrote MArray, MList, and FARArray to ensure correct persistent operation. FArray and FList are functional data structures from the PCollections library [2], and inherently use persistent-safe structures.

7.3 Configurations

To evaluate the performance of QuickCheck, we use several configurations in our evaluation. The different configurations are shown in Table 3. The *Clean* configuration is the unmodified Maxine JVM, which does not have persistent support. The *Unbiased* configuration is the AutoPersist implementation. It does not perform persistence check biasing. The *Likely* and *Unlikely* configurations bias each persistence check site to the *likely* and *unlikely* state, respectively. The *Deopt* configuration biases all persistence checks to the *very_unlikely* state. Finally, the *QuickCheck* configuration uses our proposed profiling and biasing techniques.

Configuration & Description
Clean: Unmodified Maxine JVM. No persistent support.
Unbiased: Original AutoPersist implementation. No persistence check biasing performed.
Likely: Bias all persistence checks to the *likely* state and represent them as likely taken branches as shown in Figure 6(a).
Unlikely: Bias all persistence checks to the *unlikely* state and represent them as unlikely taken branches as shown in Figure 6(b).
Deopt: Bias all persistence checks to the *very_unlikely* state and represent them as shown in Figure 6(c).
QuickCheck: Use QuickCheck to dynamically predict persistence check activation behavior and bias them accordingly.

Table 3. Configurations evaluated.

8 Evaluation

In this section, we evaluate the performance of QuickCheck. We first evaluate its performance on non-persistent applications. Next, we analyze QuickCheck's performance on persistent applications. Finally, we evaluate the accuracy of QuickCheck's profiling.

8.1 Performance in Non-Persistent Applications

Figures 7 and 8 show the execution time of the DaCapo and Scala DaCapo benchmarks when using the configurations described in Table 3. On average, for the DaCapo benchmarks, the execution time of *Likely*, *Unbiased*, *Unlikely*, *Deopt*, and *QuickCheck* is 61.0%, 51.1%, 19.1%, 8.8%, and 8.8% higher than *Clean*, respectively. It is expected that the *Likely* configuration should have the worst performance. This is because, in this configuration, the persistence checks are biased towards activating the guarded actions, while throughout execution the actions are always bypassed. It is also expected that *Unlikely* outperforms *Unbiased* and *Deopt* outperforms *Unlikely*, as stronger biases towards the expected behavior should improve performance. Finally, the *QuickCheck* configuration performs the same as *Deopt*, which shows that our infrastructure is able to successfully profile persistence check sites.

The Scala DaCapo benchmark results are very similar to the DaCapo results. On average, the execution time of *Likely*, *Unbiased*, *Unlikely*, *Deopt*, and *QuickCheck* is 62.3%, 59.7%, 38.5%, 5.7%, and 5.7% higher than *Clean*, respectively. As with the DaCapo benchmarks, each configuration's performance corresponds to its bias towards the persistence checks' guarded actions being bypassed. Overall, these results show that the biasing strategies employed by QuickCheck are effective in reducing the overhead of bypassed actions guarded by persistence checks.

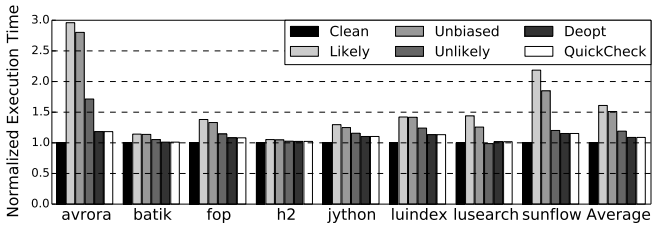


Figure 7. DaCapo execution time.

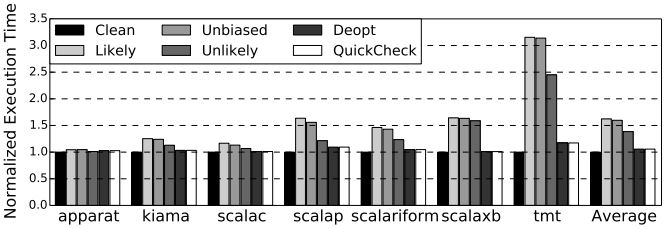


Figure 8. Scala DaCapo execution time.

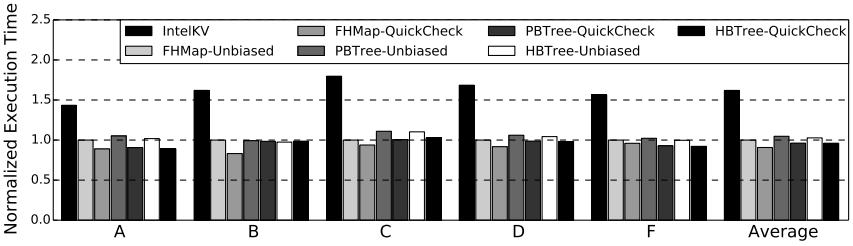


Figure 9. Persistent YCSB execution time.

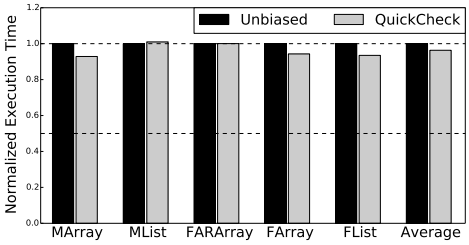


Figure 10. Persistent kernel execution time.

Overall, *QuickCheck* reduces the execution time of the DaCapo and Scala DaCapo benchmarks by 42.3% and 54.0% relative to *Unbiased*, which is the original implementation of AutoPersist. This is an average 48.2% reduction.

In *QuickCheck* the remaining overhead of persistence checks over *Clean* is largely due to needing to read each object's header word. The value of the header word is used to determine if the object as been forwarded or is persistent. While *QuickCheck* needs many additional reads compared to *Clean*, *QuickCheck*'s performance is very similar. To better understand why *QuickCheck*'s overheads are so low, we profiled every object header read in our framework. We recorded the memory address accessed and compared it to the memory address of the object field accessed next. We found that most header accesses read addresses that are very close to the addresses of subsequent memory access. In particular, on average, 88.0% and 87.6% of header accesses are within 2 cache lines of the subsequent memory access for the DaCapo and Scala DaCapo benchmarks, respectively. This indicates that the header word reads exhibit very good spatial locality and, hence, the data prefetching hardware in current processors is likely to ensure they add minimal overhead.

8.2 Performance in Persistent Applications

Figure 9 shows the execution time of the persistent memcached data structures running the YCSB benchmark suite under different configurations. A configuration corresponds to a persistent storage engine and biasing strategy, as identified with "{storage engine}-{biasing strategy}". All results are normalized to the configuration of FHMap with *Unbiased*. In the figure, we notice two traits. First, all storage engines in our framework outperform IntelKV. On average, the execution time of IntelKV is 62.0%, 54.7%, and 57.7% higher than the *Unbiased* configurations of FHMap, PBTree, and HBTree, respectively. This is because, when using IntelKV, objects must be serialized and transferred to IntelKV's C++ backend. In our framework, Java objects do not need to be serialized since they are directly stored into the non-volatile heap.

The second trait is that, for each persistent storage engine, using our biasing techniques improves performance. On average, the *QuickCheck* configurations of FHMap, PBTree, and HBTree reduce the execution time by 9.3%, 8.2%, and 6.5% compared to their respective *Unbiased* configurations. On average, this is an 8.0% reduction. Notice that the improvements are much smaller than when running the non-persistent applications. This is because, since persistent objects are being used, much of the execution time is spent executing CLWBs and SFENCEs in both configurations. However, this performance improvement is still significant, as it is provided without any new hardware or user involvement.

Figure 10 shows the execution time of the kernel applications when using *Unbiased* and *QuickCheck*. On average, *QuickCheck* outperforms *Unbiased* by 3.7%. Note that these kernels heavily write to persistent data structures and hence require many CLWBs and SFENCEs. Overall, our results show that *QuickCheck* is beneficial not only when running applications that do not require data persistence, but also for persistent programs.

8.3 Persistence Check Access Patterns

Table 4 presents the number of dynamic persistence checks encountered in the persistent memcached and kernel applications biased towards the Likely, Unbiased, Unlikely, and Very Unlikely states. The data is shown as percentages. As shown in the table, in all applications, no check is Unbiased. Also, similar to Table 1, the vast majority of the persistence checks are biased to either the Unlikely or Very Unlikely states. In the persistent memcached applications, a large percentage of the checks are in the Unlikely state. This is because these checks are in methods which are compiled by the optimizing compiler before their `activated` and `bypassed`

Structures	Likely	Unbiased	Unlikely	Very Unlikely
FHMap	0.07%	0.00%	33.08%	66.84%
PBTree	0.15%	0.00%	71.96%	27.88%
HBTree	0.06%	0.00%	71.96%	27.98%
MArray	0.18%	0.00%	0.00%	99.82%
MList	0.38%	0.00%	0.00%	99.62%
FARArray	22.12%	0.00%	0.00%	77.88%
FArray	0.40%	0.00%	0.00%	99.60%
FList	0.03%	0.00%	0.00%	99.97%

Table 4. Dynamic persistence check statistics while running YCSB and kernel workloads.

counters have enough samples. Hence, the compiler cannot confidently conclude that the persistence check's action will never be activated. Within the kernel applications, only FARArray has a large portion of checks being in the Likely state. This is because this application uses failure-atomic regions for crash consistency. As a result, many more persistence check sites encounter persistent objects.

Overall, these results further demonstrate that persistence checks have a very consistent behavior. Furthermore, the results show that QuickCheck is able to exploit the bias of persistence checks to minimize their performance impact.

9 Related Works

To simplify the process of using NVM to create persistent applications, many NVM frameworks have been proposed [3, 16, 19, 20, 22, 24, 33, 40, 47, 55, 56, 62, 64]. The amount of programmer effort involved in creating a persistent application is inversely correlated to the amount of support provided by the runtime. To enable automatic or semi-automatic data persistence, fNVMs have to insert a significant number of persistence checks at runtime, which increases the software overhead of using NVMs. Our paper is the first work that quantifies the persistence check overhead and proposes the solution of biasing persistence checks to fix this overhead.

Before the introduction of byte addressable NVM, many persistent programming languages [8, 14, 32, 39, 53, 57] and implementations [9, 43, 45] were proposed. These languages focus on attaining the *orthogonal persistence* defined by Atkinson and Morrison [10], where the persistency of an application is orthogonal to its design. In this model, all data is persistent. In addition, applications attain persistency by performing snapshots of their volatile heap. However, in current systems, where the performance of NVM is significantly lower than DRAM, it is important to minimize the amount of program state that must be persistently stored in NVM. Furthermore, performing whole-heap snapshots is expensive and does not take advantage of the cacheline-level persistent granularity provided by NVM.

Garbage Collection (GC) barriers have been studied extensively [11, 13, 25, 28, 46, 50, 51, 60, 61, 65]. Barriers are needed for concurrent, semi-concurrent, and generational garbage collectors. Persistence checks exhibit different behavior from the GC barriers. This is because whether GC barriers are activated or not is dependent on transient characteristics, such as whether a GC phase is running and the age of the object. In contrast, we show that persistence checks have highly predictable characteristics throughout the entire execution. The biasing techniques we propose in this paper are not beneficial for GC barriers due to their variable behavior.

Many languages perform inline caching [6, 7, 17, 18, 26, 41, 48] to reduce the overheads of field accesses and method invocations. For inline caching, one records the offset of field accesses and the target address of method invocations at a site-level granularity. Later, one regenerates code to optimize for commonly observed patterns, with the assumption that those patterns will not change. While the profiling technique for inline caching is similar, QuickCheck's optimization strategies for code generation are different from optimizing field accesses and method invocations.

The strategy of raising a signal in the event of a misspeculation is not unique to QuickCheck. HotSpot [41] and other Java compilers [6, 7, 59] use this strategy for tasks such as performing implicit *null* checks and for unexpected run time initialization of classes. However, to our knowledge, this paper is the first to use this strategy to reduce the persistence check overhead in programmer-friendly NVM frameworks.

10 Conclusion

In this paper, we identified a major source of overhead that limits the performance of programmer-friendly Java NVM frameworks: runtime checks guarding actions necessary for correct persistent execution. By profiling a variety of workloads, we observed that the behavior of each check is highly predictable. Based on these insights, we proposed QuickCheck, a technique that biases persistence checks based on their expected behavior, and uses speculative optimizations to further reduce persistence check overheads. We evaluated QuickCheck on a variety of data intensive application and showed that QuickCheck improves the performance of a Java-based programmer-friendly NVM framework. As we move towards NVM programming frameworks that aim to automatically identify and manage persistent objects, our work can significantly reduce the overhead of their persistence checks.

Acknowledgments

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

References

[1] Memcached: A distributed memory object caching system. https://memcached.org/
[2] PCollections. https://pcollections.org/
[3] Persistent Memory Development Kit. http://pmem.io/pmdk/

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

[5] 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

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño Virtual Machine. *IBM Syst. J.* 39, 1 (Jan. 2000), 211–238. https://doi.org/10.1147/sj.391.0211

[7] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. 2005. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.* 44, 2 (Jan. 2005), 399–417. https://doi.org/10.1147/sj.442.0399

[8] 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

[9] 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.

[10] 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

[11] 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

[12] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[13] Stephen M. Blackburn and Antony L. Hosking. 2004. Barriers: Friend or Foe?. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 143–151. https://doi.org/10.1145/1029873.1029891

[14] 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.

[15] 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

[16] 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

[17] C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF, a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

[18] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2754169.2754181

[19] 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

[20] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented recovery for non-volatile memory. *PACMPL* 2, OOPSLA (2018), 153:1–153:22. https://doi.org/10.1145/3276523

[21] OpenJDK Community. Graal Project. http://openjdk.java.net/projects/graal/.

[22] 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

[23] 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

[24] 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

[25] 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

[26] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Proc. of POPL*.

[27] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-based Register Allocation in a JIT Compiler. 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 14, 11 pages. https://doi.org/10.1145/2972206.2972211

[28] 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

[29] 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

[30] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. 1995. Profile-guided Receiver Class Prediction. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '95)*. ACM, New York, NY, USA, 108–123. https://doi.org/10.1145/217838.217848

[31] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. of PLDI*. 32–43.

[32] 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

[33] 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

[34] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. 2015. NVRAM-Aware Logging in Transaction Systems. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB'15)*.

[35] Intel. Intel Optane DC Persistent Memory. www.intel.com/optanedcpersistentmemory.

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

[37] Intel. 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.

[38] 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.

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

[40] 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

[41] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[42] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan 2010), 143–143. https://doi.org/10.1109/MM.2010.24

[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] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[45] 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

[46] Matthias Meyer. 2006. A True Hardware Read Barrier. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/1133956.1133959

[47] 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

[48] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 151–165. https://doi.org/10.1145/3192366.3192374

[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] Pekka P. Pirinen. 1998. Barrier Techniques for Incremental Tracing. In *Proceedings of the 1st International Symposium on Memory Management (ISMM '98)*. ACM, New York, NY, USA, 20–25. https://doi.org/10.1145/286860.286863

[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] 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

[54] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 657–676.

[55] 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

[56] 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)*. https://doi.org/10.1145/3314221.3314608

[57] 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.

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

[59] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. 2000. Overview of the IBM Java Just-in-time Compiler. *IBM Syst. J.* 39, 1 (Jan. 2000), 175–193. https://doi.org/10.1147/sj.391.0175

[60] 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

[61] Martin T. Vechev and David F. Bacon. 2004. Write Barrier Elision for Concurrent Garbage Collectors. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY,

USA, 13–24. https://doi.org/10.1145/1029873.1029876

[62] 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

[63] 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

[64] 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

[65] Benjamin Zorn. 1990. *Barrier Methods for Garbage Collection.* Technical Report.