MAKING NON-VOLATILE MEMORY PROGRAMMABLE

BY

THOMAS SHULL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Abstract

Byte-addressable, non-volatile memory (NVM) is emerging as a revolutionary memory technology that provides persistence, near-DRAM performance, and scalable capacity. By using NVM, applications can directly create and manipulate durable data in place without the need for serialization out to SSDs.

Ideally, through NVM, persistent applications will be able to maintain crash-consistency at a minimal cost. However, before this is possible, improvements must be made at both the hardware and software level to support persistent applications. Currently, software support for NVM places too high of a burden on the developer, introducing many opportunities for mistakes while also being too rigid for compiler optimizations. Likewise, at the hardware level, too little information is passed to the processor about the instruction-level ordering requirements of persistent applications; this forces the hardware to require the use of coarse fences, which significantly slow down execution.

To help realize the promise of NVM, this thesis proposes both *new software and hardware support that make NVM programmable*. From the software side, this thesis proposes a new NVM programming model which relieves the programmer from performing much of the accounting work in persistent applications, instead relying on the runtime to perform error-prone tasks. Specifically, within the proposed model, the user only needs to provide minimal markings to identify the persistent data set and to ensure data is updated in a crash-consistent manner.

Given this new NVM programming model, this thesis next presents an implementation of the model in Java. I call my implementation *AutoPersist* and build my support into the Maxine research Java Virtual Machine (JVM). In this thesis I describe how the JVM can be changed to support the proposed NVM programming model, including adding new Java libraries, adding new JVM runtime features, and augmenting the behavior of existing Java bytecodes.

In addition to being easy-to-use, another advantage of the proposed model is that it is amenable to compiler optimizations. In this thesis I highlight two profile-guided optimizations: eagerly allocating objects directly into NVM and speculatively pruning control flow to only include expected-to-be taken paths. I also describe how to apply these optimizations to AutoPersist and show they have a substantial performance impact.

While designing AutoPersist, I often observed that dependency information known by the compiler cannot be passed down to the underlying hardware; instead, the compiler must insert coarse-grain fences to enforce needed dependencies. This is because current instruction set architectures (ISA) cannot describe arbitrary instruction-level execution ordering constraints. To fix this limitation, I introduce the *Execution Dependency Extension (EDE)*, and describe how EDE can be added to an existing ISA as well as be implemented in current processor pipelines.

Overall, emerging NVM technologies can deliver programmer-friendly high performance. However, for this to happen, both software and hardware improvements are necessary. This thesis takes steps to address current the software and hardware gaps: I propose new software support to assist in the development of persistent applications and also introduce new instructions which allow for arbitrary instruction-level dependencies to be conveyed and enforced by the underlying hardware. With these improvements, hopefully the dream of programmable high-performance NVM is one step closer to being realized.

To my family and friends.

Acknowledgments

My Ph.D. has been a very memorable journey. While there have been many experiences, by far the most valuable takeaway has been the friends I have made and researchers I have met throughout the process. They have made my Ph.D. very enjoyable and something I will look back at with fond memories.

First, I would like to thank my advisor, Professor Josep Torrellas. Throughout my Ph.D., Josep has expertly balanced allowing me enough freedom to explore random topics while also guiding me to areas ripe for research. Josep has always been supportive and willing to spend time to help me concretize my ideas. His work ethic is unparalleled and serves as an inspiration to all.

I would also like to thank my committee for their help and guidance. I have worked with Jian extensively on designing a new non-volatile memory (NVM) programming environment. His expertise and insights on NVM systems has been invaluable throughout the project and has helped to increase its impact. Too, I have appreciated Jian's help in writing papers and his willingness to spend the time and effort needed to ensure everything is done right. Thank you also to the rest of my committee for your insightful questions throughout my exam and for your comments on my dissertation.

I would like to express my sincere gratitude to the various Computer Science Department support staff who have helped along the way. Thank you to Sherry Unkraut and Madeleine Garvey for helping with all of the i-acoma travel arrangements and scheduling. Likewise, I'd like to thank Kara MacGregor, Kathy Ann Runck, Mary Beth Kelly, Viveka Kudaligama, and Maggie Metzger Chappell for providing assistance with various program logistics.

I am also grateful for the camaraderie of the other i-acoma group members, both past and present. I joined the group along with three other students, Bhargava, Jiho, and Raghavendra, and have shared many of the Ph.D.'s important milestones alongside them. Wonsun served as my first mentor and helped introduce me to managed language virtual machines. Jiho and I worked together on many JavaScript projects. Throughout the years, I have enjoyed also getting to know Antonio, Apostolos, Dimitris, Mengjia, Serif, Wooil, and Yasser, among others.

Beyond the i-acoma group, I'd also like to thank the other friends I met throughout graduate school. Thank you for serving as an enjoyable escape from work while also expanding my horizons.

My family has been essential for me both in striving for, and completing my Ph.D. Throughout my entire academic career, my family has always been supportive and has always encouraged me to make the most of my potential. Thank you for helping to set me up for success in graduate school and beyond.

Finally, and most importantly, I would like to thank my wife, Sofi. I met Sofi here while in grad school, and she is by far and away its greatest reward. Thank you for helping me stay focused and always being supportive. I am grateful that we get to begin our new journey united, as we both travel through valleys and stand atop mountains together.

Table of Contents

Chapter 1: Introduction

1.1 THE PROMISE OF EMERGING NON-VOLATILE MEMORY (NVM) TECHNOLOGIES

Byte-addressable non-volatile memory (NVM) holds much promise within the computing world. Traditionally memory technologies have an inherent tradeoff where they can be either fast, small, and transient (volatile), or slow, large, and durable (non-volatile). Because of these tradeoffs, large hierarchies of memories are present in today's systems, allowing frequently queried data to reside in caches near the processor, while peripheral block-storage devices are used to store both inactive data and data which must not be lost when an application terminates.

NVM is interesting because it does not fit within the traditional memory dichotomy. Instead, NVM's traits bridge the gap between existing memory technologies. In particular, NVM is byte-addressable like DRAM, but is non-volatile like storage devices. In terms of performance, NVM is slower than DRAM but is faster SSDs. Likewise, NVM has higher capacity than DRAM but less than SSDs.

Recently, significant technological advances have been made towards manufacturing NVM, such as Intel 3D XPoint [1], Phase-Change Memory (PCM) [2], and Resistive RAM (ReRAM) [3]. Indeed, in April 2019 Intel began to sell 3D XPoint memory [1] to be used within servers and plans to also incorporate NVM into their consumer products.

Because of NVM's unique characteristics, it is poised to benefit many different domains. Due to its byte-addressability and larger capacity than DRAM, one immediate use case of NVM is to improve the performance of in-memory databases. In addition, its non-volatility can be leveraged to improve the performance of file systems [4] and allow for smaller batching sizes within real-time logging systems such as Java Flight Recorder [5].

However, many researchers are also optimistic that NVM can help fundamentally change how persistent applications are programmed. In particular, NVM provides the opportunity for an application's working data set and durable data set to be unified. This eliminates the act of copying data to/from durable storage and also devising a serialization scheme. Hence, in the future, NVM is poised to allow for faster application recovery, fine-grain persistent updates, and faster performance, all while also being simpler to program.

1.2 CURRENT SUPPORT FOR PERSISTENT APPLICATIONS

While non-volatile memory technologies have much promise, leveraging NVM in persistent applications requires both hardware and software support. On the hardware side, since volatile caches sit between the processor and NVM, instructions must be issued to ensure data is propagated to NVM. Intel's x86-64 processors have introduced the `CLWB` instruction [6], which writes back a cache line to NVM while also retaining the line in the cache, to help push data to NVM. Likewise, within Arm's AArch64 architecture, the `DC CVAP` instruction [7] has been introduced to accomplish the same function.

Another consequence of volatile caches is that even when data is forced to propagate to memory via `CLWB`s or `DC CVAP`s, the order in which data is persisted may not follow program order; this can cause an inconsistent program state to be present at the time of a crash. To prevent reorderings, currently fences must be added to the execution to ensure data is persisted in the desired order. On x86-64, the `SFENCE` instruction is used to block subsequent writes from executing until all prior stores and `CLWB`s have completed. On AArch64, the `DSB` instruction must be used to order stores and `DC CVAP`s. Note that `SFENCE` and `DSB` predate NVM, and have been retrofitted to also enforce NVM orderings. Furthermore, these instructions enforce an ordering on *all* stores and NVM writebacks, and therefore have significant performance overheads.

In addition to the hardware coordination described above, much software support is also needed to maintain crash-consistency. Such support includes the ability to relocate data to different virtual address regions, crash-consistent memory allocation and collection, and support for collectively having sets of writes atomically persist. To help assist with these tasks, many NVM programming frameworks have been proposed. Proposed frameworks include Intel's PMDK [8], Mnemosyne [9], NVHeaps [10], Espresso [11], and others [12, 13, 14, 15, 16, 17, 18].

While the specific details of the frameworks vary, at minimum these frameworks have support for persistent allocation and primitives to enable the persistent updates of data. Overall, however, I believe that current frameworks place too many burdens on the programmer. In particular, existing frameworks require the programmer to either explicitly identify all persistency characteristics of data structures and objects, or they require the programmer to use library data-structures which have already been correctly labeled. This limitation makes the process of creating persistent applications very time-intensive, and also introduces opportunities for correctness and performance bugs due to the increased programming complexity [19]. Moreover, it limits the ability of persistent applications to reuse existing libraries.

1.3 THESIS CONTRIBUTION AND OVERVIEW

With NVM devices available today, it is critical that the proper software tools are available to ensure programmers can leverage NVM effectively. In addition, it is important that hardware is able to efficiently execute code patterns commonly found within persistent applications.

I believe the NVM frameworks available today are too complicated to be widely adopted by programmers. In response, in this thesis I propose a new NVM application environment called *AutoPersist*. AutoPersist strives to achieve a better balance of programmability, correctness, and performance than prior NVM frameworks.

Likewise, I find that current instruction set architectures (ISA) are unable to effectively communicate to the hardware the ordering requirements of patterns commonly found within persistent applications. To fix this issue, in my thesis I propose to extend the ISA to allow for arbitrary instruction-level ordering dependencies to be defined within the ISA. I call my extension the *Execution Dependency Extension (EDE)* and describe both how it can be incorporated into Arm's AArch64 instruction set architecture as well as the underlying hardware representation.

1.3.1 Summary of Contributions

Designing a New Programmer-Friendy NVM Programming Model. Before designing a new NVM framework, I studied and characterized existing frameworks [20]. Based on my characterization, I discovered that existing NVM frameworks too closely match the underlying hardware. In particular, they require the user to explicitly identify all data which must reside in NVM and wrap stores that need to be performed persistently. In my opinion, these requirements place too much of a burden on programmers and create many opportunities to write buggy programs.

To make using NVM simpler, I propose a new NVM programming model which requires minimal markings to define persistent data. Specifically, I introduce the concept of *durable roots*, i.e., named objects that are used at recovery time as hooks into the persistent data structures. Once a user identifies the durable roots, my model requires the runtime to automatically make all objects reachable from durable roots recoverable and also requires the runtime to ensure stores to these objects are performed persistently. Such a model will be intuitive to programmers and foster the integration of NVM into applications.

AutoPersist: An Easy-To-Use Java NVM Framework. Based on my model, I propose a new NVM framework in Java which I call AutoPersist [21]. In Chapter 4, I define how such a programmer-friendly NVM programmer model could be applied to Java. In addition, I describe how the underlying Java Virtual Machine (JVM) must be modified to support this model. These changes include augmenting each object's internal representation, changing the heap layout and garbage collection process, and also extending the behavior of many JVM bytecodes.

I implement AutoPersist on top of the Maxine JVM [22] and run my framework on top of a system with NVM DIMMs. I find that AutoPersist is able to significantly simplify the task of creating persistent programs while also outperforming existing Java NVM frameworks. This is because since AutoPersist is integrated into the JVM, it is better at understanding the underlying object layout, performing dynamic optimizations, and customizing the execution to be tailored towards the observed common cases.

Improving AutoPersist's Performance Through Profile-Guided Optimizations. While in the baseline AutoPersist implementation significantly outperforms existing Java NVM framework offerings, I find that many opportunities still exist to improve AutoPersist's performance. In particular, I find two significant sources of overhead. First, I observe that the runtime often must move objects between the volatile and non-volatile heaps. Second, *persistence checks*, or checks introduced in AutoPersist to determine when runtime actions must be invoked, incur a significant execution overhead, even if the runtime action is bypassed.

To reduce these overheads, I propose two profile-guided optimizations. First, I propose to reduce the overhead of moving objects to NVM by identifying which objects are likely to become persistent and eagerly allocating these objects within NVM. Second, I propose QuickCheck, a technique to *bias* persistence checks towards their expected behavior [23]. QuickCheck divides the program execution into two phases, a profiling phase and an optimization phase. During the profiling phase, persistence checks are instrumented to record whether the guarded actions are bypassed or executed. Later, during the optimization phase, the code is recompiled and machine code representing the persistence checks and their guarded actions is generated to be biased towards the expected behavior. To further improve performance, guarded actions expected to be always bypassed are *speculatively* completely removed from the code.

ISA Support for Instruction-Level Execution Dependencies. Currently, manipulating persistent data structures requires the insertion of fences into the code to ensure writes propagate to NVM in a specific order. For instance, while performing undo logging, the undo log entry must be persisted before the original element can be updated.

In undo logging, an update has an *execution dependence* on its corresponding log entry. An execution dependence means that, for correctness, the execution dependences source operation must complete before the dependences sink can make any observable memory changes.

For performance, different log updates should be able to proceed in parallel. However, in current ISAs, these execution dependencies are not able to be conveyed between independent (i.e., no register or memory dependent) instructions. Therefore, programmers must instead use fences, which enforce an execution ordering between *all instructions* and serialize independent log updates.

To remedy this, I propose the *Execution Dependency Extension (EDE)*, which allows for fine-grain execution ordering dependencies to be represented within the ISA, and also describe how EDE can be added to Arm's AArch64 ISA.

I also present two hardware realizations of EDE. My hardware proposals enforce EDE's execution dependencies at different stages within the pipeline: one within the issue queue (*IQ*) and another at the write buffer (*WB*). Overall, I find that EDE is able to significantly improve persistent applications' performance.

1.4 THESIS ORGANIZATION

This thesis is organized as follows. Chapter 2 describes current NVM offerings, what instructions are needed to ensure data is persisted, and current techniques for creating persistent applications. Next, Chapters 3 through 6 present the design of AutoPersist. Chapter 3 describes the programming model used within AutoPersist; Chapter 4 describes how AutoPersist is implemented within a Java Virtual Machine (JVM); and Chapter 5 discusses several of AutoPersist's optimizations. Finally, Chapter 6 evaluates the performance of AutoPersist.

In Chapter 7, I introduce the EDE ISA extension, and describe how it can be implemented in hardware. Finally, Chapter 8 concludes by summarizing my contributions.

Chapter 2: Background & Related Work

2.1 CURRENT HARDWARE NON-VOLATILE MEMORY (NVM) SUPPORT

In recent years, technological advances have been made towards having byte-addressable non-volatile memory. Whereas traditionally durably storing data required the use of block-based storage devices such as Hard Disk Drives (HDDs) or Solid State Drives (SSDs), new device technologies such as Phase-change memory (PCM) [2, 24] and Resistive RAM (ReRAM) [3] that offer non-volatile memory with byte-level access granularity are being rapidly developed.

These new technologies are known collectively as non-volatile random-access memory (NVM). NVM offers substantial performance improvements over traditional storage devices; it has performance similar to current volatile dynamic random-access memory (DRAM), yet also has higher capacities and retains its values across system restarts. Intel has already released NVM products [25], including releasing NVM in DIMM form factor in April 2019.

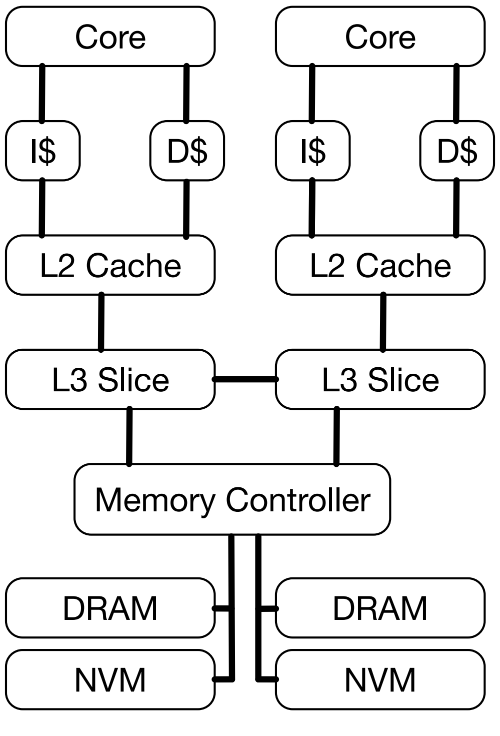


Figure 2.1: Hybrid DRAM+NVM hierarchy.

Currently, NVM is being used alongside DRAM in *hybrid* memory systems. An example hybrid memory hierarchy is shown in Figure 2.1. In such a setup, Intel systems present the user with two operation modes for NVM [26]: Memory Mode, where DRAM serves as a transparent cache for the NVM physical address space; and AppDirect Mode, in which NVM and DRAM are mapped to distinct physical addresses and can be directly accessed individually. Note that on Intel systems a given channel has the option to contain both NVM and DRAM DIMMs; this is to accelerate the Memory Mode's DRAM caching.

While NVM moves non-volatile storage a level closer to the processor, as shown in Figure 2.1, many levels of volatile cache still exist between the processor and NVM. Hence, one needs to ensure that persist writes propagate their values beyond the caches and reach NVM. For this reason, x86-64 processors have introduced the `CLWB` instruction [6], which writes back a cache line to NVM while also retaining the line in the cache.

In Armv8.2-A [27], Arm introduced new instructions to propagate writes to the persistence domain. Specifically, Arm has added the *Data or unified Cache line Clean by Virtual Address to Point-of-Persistence (`DC CVAP`)* [7] instruction to its AArch64 ISA. Like `CLWB`, `DC CVAP` ensures the value at the provided virtual address is sent to NVM.

In both x86-64 and AArch64, `CLWB` and `DC CVAP` follow a relaxed memory ordering. This means, while using these instructions, the order in which the updates reach NVM and are made persistent is not deterministic. Hence, to guarantee a given ordering, fences must be inserted. On x86-64, to guarantee that all prior `CLWB` instructions have completed before subsequent writes and `CLWB`s can execute, one needs to insert a storage fence (`SFENCE`) [6] instruction.

Note that on current Intel systems a store is considered complete once it reaches the on-DIMM memory controller; it does have to wait until it is actually stored on the non-volatile media. This is because the memory controller is part of the *Asynchronous DRAM Refresh (ADR)* [28] domain on Intel products. ADR expands the persistent domain to include additional components which are guaranteed to flush to NVM on shutdown. On Intel chips, the memory controller is added to the ADR by including enough capacitance to provide power to write all data within the memory controller to NVM in the event of a crash.

On AArch64, a system-wide data synchronization barrier (*`DSB`*) [7, 29] is needed to order `DC CVAP`s relative to other instructions. `DSB` imposes an ordering on *all* instructions – before any instruction after a `DSB` can execute, all prior instructions must finish. Note that while AArch64 also provides a data memory barrier (`DMB`), unfortunately currently this instruction does not order `DC CVAP`s.

2.2 USING NVM IN PERSISTENT APPLICATIONS

The Storage Networking Industry Association (SNIA) has been working to standardize interactions with NVM. It has created a low-level programming model [30] 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 toolkit compliant with SNIA's device-level model. This project has resulted in the development of the Persistent Memory Development Kit (PMDK) [8], a collection of libraries in C/C++ and Java that a developer can use to build persistent 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 provides a library of persistent 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.

```
POBJ_LAYOUT_BEGIN(list);
POBJ_LAYOUT_ROOT(list, struct durable_list);
POBJ_LAYOUT_END(list);
struct durable_list{
  int element;
  TOID(struct durable_list) next;
}
TOID(struct durable_list) head; \\initialized elsewhere
void insert(PMEMobjpool *pop, int element){
  TX_BEGIN(pop){
    TOID(struct durable_list) node =
        TX_NEW(struct durable_list);
    D_RW(node)->element = element;
    TX_ADD_FIELD(head, next);
    D_RW(node)->next = D_RO(head)->next;
    D_RW(head)->next = node;
  } TX_END
}
```

Figure 2.2: Example using NVM pragmas.

Figure 2.2 shows how to use PMDK's macros to create a persistent list and also the method to add needed elements to the head of the list. As shown in the figure, the user is expected to describe the layout of the list via `POBJ_LAYOUT_(*)` macros. In addition, each pointer access to persistent memory must be done through a `D_RW` or `D_RO`. The above macros are needed because PMDK uses location-independent pointers which allows for the objects to be assigned a different virtual address space across restarts. Finally, the list prepend must be performed within a transaction to ensure that either the action persistently completes or the newly allocated memory is recovered in the event of a crash. PMDK provides `TX_(*)` macros to denote the transactional region (`TX_BEGIN` and `TX_END`), transactional memory allocation (`TX_NEW`), and transactional memory modifications to persistent state (`TX_ADD_FIELD`).

In addition to the industrial efforts, academia has also proposed several frameworks for NVM [11, 10, 9, 12, 13, 14, 15, 16, 17, 18]. With the exception of Espresso [11], all of these languages target C/C++. Some frameworks leverage language-level support directly [17, 18], whereas others are built as libraries and provide a persistence API to the user.

Another differentiating factor between these frameworks is how both failure-atomic regions are identified and also race resolution occurs. Failure-atomic regions are portions of code which provide the appearance of having all persistent updates complete atomically at the end. Some frameworks, such as BPFS [12], propose to periodically create epochs to which all memory operations are aligned. Others, such as Atlas [13], tie failure-atomic regions to existing critical sections within the code. Finally, others, such as NVMReconstruction [17], expect the user to identify failure-atomic regions manually.

Once failure-atomic regions are identified, another responsibility of many frameworks is to ensure that the failure-atomic regions of multiple threads compose into a crash-consistent state. One approach to accomplish this is to use software transaction memory to rollback racy updates [9]. Another approach is to track happens-before dependencies between log entries and coordinate the rollback of log entries across multiple threads [13]. Note, however, that it is not necessary for the runtime to offer race detection and mitigation. Instead, many frameworks, including PMDK, expect the user to correctly synchronize their application to prevent races between failure-atomic regions.

Presently, the two most popular ways to durably store objects within Java [31] is by either using the Java Persistence API (JPA) or extending Java's `Serializable` interface. JPA is an API which allows applications to transparently interface with databases from multiple providers. Alternatively, extending the `Serializable` interface allows an application designer to directly write objects to durable storage. These existing techniques are designed for when there is a separation between the volatile main memory and non-volatile storage. New frameworks need to be designed for Java to fully leverage the capabilities of NVM.

2.3 IMPROVING NVM PERFORMANCE

Many proposals have been made to improve the performance of NVM. An important area of study has been to devise new data structures which optimally leverage NVM's traits. This includes leveraging NVM's byte-addressability in structures [32, 33] as well as storing in DRAM data, such as indexes, which can be recreated across crashes [34, 35].

As stated before, often it is necessary for a collection of writes to NVM to have the appearance of being persistently completed atomically. To do so requires the use of logging to enable the recovery of a consistent state in the event of a crash. Therefore, logging is a key feature of NVM application and hence has been studied extensively. Many works have tried to optimize the performance of logging in software [36, 37, 38, 39, 40]. These works try to limit the amount of fences which must be added to the problem by either redoing work after a crash or by leveraging Intel's intra-cacheline ordering guarantees.

In addition to the software logging optimization techniques cited above, many researchers have also proposed hardware techniques to accelerate logging [41, 42, 43, 44, 45, 46]. In these works, the performance of logging is increased by the introduction of advanced memory controllers, persistent caches, and/or additional persistent storage buffers. Unfortunately, these hardware approaches are very brittle, have unclear software interoperability, and are tied to a specific logging approach. I believe at most hardware should only offer primitives for logging and that the majority, if not all of the logging support, should be implemented in software.

Similar to how a processor's consistency model dictates when stores and loads become visible to other threads, *persistency models* have been proposed [47, 48, 49, 50, 51, 52, 53] to dictate how loads to and stores from non-volatile memory can be reordered by the hardware. The proposed persistency models allow different amounts of reordering, with more relaxed models potentially having better performance with the tradeoff of possibly creating very counterintuitive data states in the non-volatile memory. Based on the hardware's underlying persistency model, the software must decide where fences are needed within the application to ensure a specific ordering.

Another way to improve the performance of NVM is to allow the crash-consistent state to lag behind the program's visible state. This can be done by adding buffers within the hardware to store pending NVM writes and to monitor the system's coherence messages to determine the order in which data must be written back to NVM. This concept, known as *buffered persistency*, has been proposed in multiple works [15, 54, 55, 52]. While buffered persistency can significantly improve performance, I believe the cost of monitoring coherence messages is prohibitive. Instead, a simpler and more effective solution is to add additional capacitance to chips which can be used to flush volatile cache state in the event of a shutdown. This approach, known as *eADR* (enhanced asynchronous DRAM refresh), allows for persistent fences to be removed from programs without the need to monitor coherence messages.

2.4 JAVA VIRTUAL MACHINE (JVM) BEHAVIOR

Java programs are executed on top of a Java Virtual Machine (JVM) [56]. 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 generates machine code customized for the current execution environment's architecture.

Some of Java's key features are its object transparency and automatic memory management. In Java, users do not have direct pointers to memory. Instead, they have reference handles to *objects*. These reference handles hide an object's underlying representation, allowing the runtime to store metadata hidden from the user alongside user-declared object fields. This metadata, also known has *header fields*, helps to improve the performance of runtime features such as synchronization and garbage collection.

In addition, since users can only access objects through references, the runtime is free to move objects around throughout execution. Object movement is done both for performance reasons and also to collect garbage. Garbage collection is a key feature of Java. Instead of the user having to explicitly free memory, the runtime is able automatically to identify unreachable objects and to reclaim this memory. This helps to improve programmer productivity and eliminate common bugs such as dangling pointers.

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.

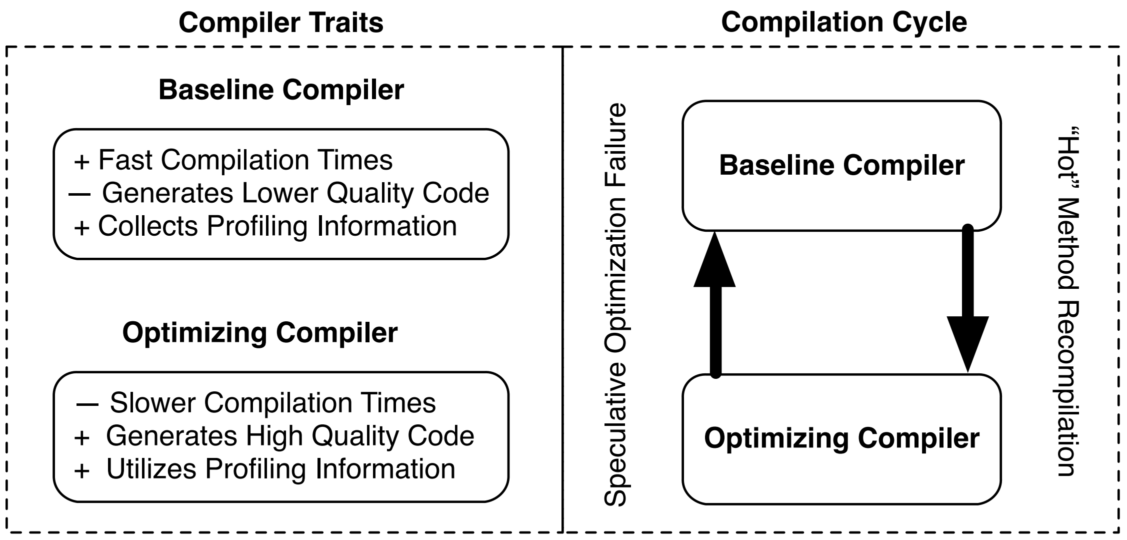

Compiler Traits
Compilation Cycle
Baseline Compiler
+ Fast Compilation Times
— Generates Lower Quality Code
+ Collects Profiling Information
Optimizing Compiler
— Slower Compilation Times
+ Generates High Quality Code
+ Utilizes Profiling Information
Speculative Optimization Failure
Baseline Compiler
Optimizing Compiler
"Hot" Method Recompilation

Figure 2.3: JVM compilation overview.

To fully leverage the potential of JIT compilation, advanced JVM compilers are organized into multiple tiers, where each tier offers a different tradeoff between the time to generate code and the quality of code generated. Figure 2.3 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 I 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 an *on-stack-replacement* [57] fallback, and the points at which they can occur are known as *deoptimization points*.

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 [58, 59, 60, 61, 62, 63].

Chapter 3: Designing a New Programmer-Friendy NVM Programming Model

3.1 INTRODUCTION

By using NVM, persistent applications have the potential to improve in both simplicity and performance. Unfortunately, current NVM frameworks leave much to be desired. In particular, they require the programmer to label all persistent data, identify which stores need to be persistently handled, and disallow preexisting built-ins and libraries to be used in a persistent manner. This increases the adoption difficulty and presents many opportunities for the programmer to make mistakes.

In addition, current frameworks' low-level abstraction is a mismatch for today's managed languages, such as Java. For instance, in managed languages, objects' underlying representation and location are hidden from the user; therefore, it is incongruent for users to consider the persistency implications for each object and write.

Finally, since current NVM frameworks are implemented as libraries, as opposed to having full language- and compiler-level integration, the implementations are rigid and performance is left on the table. This is because the compiler is unable to understand the programmer's intentions and optimize the code accordingly.

This chapter explains the limitations of existing frameworks and proposes a new programmer-friendly NVM programming model. In particular, the proposed model minimizes the amount of markings an application developer must add to the program, allows for existing code to be handled persistently, and is amenable to full language- and compiler-level integration.

To minimize the amount of user markings required and enable the reuse of preexisting code, my model leverages *persistence by reachability*. Persistence by reachability expects the runtime to ensure that all objects reachable from an object labeled as persistent to also be handled persistently. Via persistence by reachability, a user must only label handles in data structures requiring persistency in order to ensure the entire data structure is made persistent. Furthermore, via persistence by reachability, if a handle points to a built-in or library object, then this object must also be handled persistently. Therefore, via persistence by reachability, the user needs to add very few markings into their code and also existing code can be handled persistently.

By default, in my model it is the requirement of the underlying implementation to persist all durable objects automatically. In this way, the user does not have to explicitly identify persistent stores. In addition to its default behavior, my model also enables the user to identify larger regions which will abide by *failure-atomic* semantics (i.e., either all or none of the region's updates become part of the persistent state).

Finally, since my model requires minimal user intervention and instead places requirements on the underlying runtime, there are many opportunities for optimizations. This is because the compiler is free to optimize the implementation based on both the observed application runtime behavior and also the features of the underlying hardware.

3.2 LIMITATION OF EXISTING NVM FRAMEWORKS

Existing frameworks with support for programming NVM ask programmers to make many concessions. A programmer must correctly mark all memory which should be durable and ensure that data is persisted properly either through explicit failure-atomic regions or persists. This is an error-prone process requiring many markings in code and prohibiting the use of preexisting libraries. As highlighted by Ren et al. [19], programmers have many difficulties correctly adapting code to be compliant with existing NVM frameworks.

3.2.1 Abstraction Level Mismatch

Current frameworks are incongruent with the current trend towards managed languages. Managed languages, such as Java, try to lower the programmer burden and increase both safety and productivity.

Instead of targeting a specific machine, Java applications target the Java Virtual Machine (JVM) [56]. By doing so, the underlying representation of objects is hidden from the user. This allows the runtime to change the objects' representation by performing such optimizations as colocation [64] and scalar replacement [65]. Another key feature of Java is garbage collection (GC). Because the user is unable to directly manipulate pointers, the runtime is able to move objects to both collect dead objects as well as remove fragmentation.

Unfortunately, existing NVM frameworks are tied to the underlying hardware. Currently the framework's features closely match the current hardware primitives. In particular, the user must make static decisions about what objects should be placed in NVM as well as identifying which writes are to persist data. This forces the user to consider object layout decisions which are typically abstracted away. Furthermore, in managed languages where objects are already moved around to GC, whether an object is a persistent or not should be a dynamic trait. However, current NVM frameworks do not allow for dynamic traits to be defined.

Another tenet of managed languages such as Java is to ensure safe execution. Java performs many runtime checks to detect incorrect programs early before lasting damage is done. For instance, Java automatically checks array accesses to ensure the element being accessed remains within bounds and triggers an exception as soon as an out of bounds access occurs, preventing unintentionally buggy programs or malicious entities from continuing to execute and potentially leaking or corrupting memory. Contrary to Java, existing NVM frameworks present many opportunities for unchecked or silent errors to occur, such as if non-volatile memory points to volatile memory or if a consistent program state is not persisted before a crash occurs.

3.2.2 Incompatibility with Existing Code

Managed languages typically rely on a large central set of libraries and utilities included by default with their distributions. Programmers appreciate that via this built-in functionality there is a de facto standard application programmer interface (API) for many data structures and template for accomplishing most tasks. For instance, in Java, programmers rely on a large set of `List`, `Map`, `Tree`, and `Set` data structures to hold data.

Unfortunately, since existing NVM frameworks require each durable object to be marked, existing built-in libraries cannot be used, as they will not have the proper durable markings and persists in place. This is detrimental in at least two ways. First, since existing code cannot be reused, a substantial amount of work must be performed to recreate existing infrastructure. This reduces how likely developers will switch to using fine-grain persistency within their applications. The greater issue is that increasing the amount of code one has to create increases the likelihood of bugs being introduced into the code. A key feature of libraries is that they have been verified for functional correctness; newly written code is much more likely to have correctness, performance, and persistency bugs.

3.2.3 Limited Optimization Potential

Java and managed languages in general try to provide the programmer with simple intuitive models which are easy for the user to adhere to. However, while the models provided may be high-level, users still expect competitive performance. Indeed, users expect Java code to execute efficiently and have minimal overheads. To accomplish this, most Java/JVM implementations employ Just-In-Time (JIT) compilers with speculative optimizations to attain maximal performance. JIT compilation allows for the generated code to be optimized for the common, or “hot,” paths seen during execution. Furthermore, since the models Java provides are high-level, the compiler has much freedom to perform optimizations which may benefit the current execution.

Unfortunately, low-level frameworks, such as what exists for NVM currently, have limited optimization potential. This is because they are overspecified – by the framework features being closely tied to existing hardware. As a result, the high-level intentions of the programmer are lost, making it hard to for a compiler to be effective.

For instance, in many frameworks the user must manually perform persist operations and design the logging necessary for failure-atomic regions. This ties the application to a specific implementation of failure-atomic region support. Furthermore, if a user manually emits persist operations, they may be unnecessary or suboptimally placed. Unfortunately, the compiler will struggle to optimize around and remove them, as explicit persist operations can have barriers which limit the compiler's ability to perform optimizations.

Overall, these existing frameworks impose many restrictions on application programmers which will limit their integration into managed languages. Clearly, a new NVM programming model is needed to match the expectations of managed language programmers.

3.3 NEW PROGRAMMER-FRIENDLY NVM PROGRAMMING MODEL

In the previous section, I highlighted the main deficiencies of existing NVM frameworks; namely, that many of their features are not aligned with the philosophy of Java and other managed languages. In this section, I provide a high-level specification of a new NVM programming model tailored to managed languages.

3.3.1 Model Goals

An ideal model for programming NVM is very intuitive for a programmer to use, not overspecified, and also is decoupled from the underlying hardware. This allows for the model to remain unchanged as hardware improves, enables the compiler to make aggressive optimizations, and minimizes the chances for the programmer to write incorrect persistent applications. Below the main goals a new model should attain are described.

Goal 1. As few objects as possible should require durable markings.

Current models require programmers to mark many objects as durable. This is because they want to ensure only objects which must necessarily be durable incur the performance overheads of residing in non-volatile memory. However, this is very error-prone and requires the programmer to mark many objects. Contrary to this, I believe the number of durable markings should be minimal; a user should only have to mark objects immediately visible during the crash recovery process. Instead, the runtime should then automatically make all objects reachable from these few objects durable as well.

Goal 2. Libraries and other pre-existing codes should not need to be changed to work correctly in a durable program.

As described in Chapter 3.2, existing NVM frameworks cannot be used with current unmodified standard libraries. I believe this is unacceptable – users should not be forced to rewrite large swaths of code, potentially introducing bugs, to create durable applications.

Goal 3. The user should not need to explicitly persist durable objects.

Many current NVM frameworks require the user to explicitly persist objects to ensure a value reaches NVM. This limits the amount of optimizations the compiler can perform and potentially enables the user to either add an excessive or insufficient number of persist operations. I believe that instead a framework should automatically persist durable objects as necessary without user involvement.

Goal 4. A clear and simple persistency model should be provided.

As described in Chapter 2.1, the order of operations to NVM may not be in program order unless measures are taken, due to caches in between the processor and NVM. This can result in program state at recovery time that does not correspond to a sequential execution of the application. Hence, a persistency model must be established and enforced by the framework that is intuitive to the user and simplifies recovery.

Goal 5. Failure-atomic region support should be provided and need only minimal markings.

In many cases, it is necessary for a region of code to appear to execute atomically in case of failure, with either all or none of the operations in the region being persisted. I believe support for failure-atomic regions must be provided, and that it should be intuitive for programmers to use. Namely, the user should not need to differentiate between durable and volatile objects within the region and the mechanisms for achieving this atomicity should be transparent. In addition, the user should not need to specify how the atomic support should be implemented; instead, the framework should perform logging in the most efficient way for the specific use case.

3.3.2 Establishing a New NVM Programming Model

With the above goals in mind, I now establish a new NVM programming model for managed languages. My model consists of three requirements that the *runtime* must ensure hold true. The requirements I create fall into two categories: determining which objects must be placed in non-volatile memory and ensuring that the order in which stores are persisted is intuitive to programmers.

Placing Objects in Non-Volatile Memory.

NVM Model Requirement 1. All objects reachable from the durable root set must be recoverable and in non-volatile memory.

I define the *durable root set* as the set of pointers which are named entries into durable structures. At recovery time, the programmer can directly access these roots by name. Since these roots are visible across executions, by necessity they must be named and marked; otherwise, they cannot be recovered if a crash were to occur.

This requirement helps to meet Goals 1 and 2. This requirement helps to meet Goal 1 because it only requires that the durable root set have markings; since it ensures all of the objects reachable from this set to be stored in non-volatile memory, it is unnecessary to mark them. Note that all objects which should be durable must be reachable from a durable root; since they are unnamed, otherwise it would be impossible to access them across executions.

This requirement also helps meet Goal 2, as it implies that if a library data structure is reachable from a durable root, then it will automatically be made durable. This prevents the libraries from having to be modified in any way. Specifically, built-in classes' fields do not need durable markings as is necessary in existing NVM frameworks.

To meet this requirement, the runtime may need to move objects to non-volatile memory when it detects they are reachable from a durable root. Note that managed languages already move objects throughout execution while performing garbage collection. However, how the runtime chooses to adhere to this requirement is implementation specific.

Controlling Persistent Atomicity Granularity.

NVM Model Requirement 2. Support for failure-atomic regions must be provided. All stores to durable objects within a failure-atomic region should appear to have been performed atomically and persistently at the end of the region.

This requirement is intended to satisfy Goal 5. Namely, it ensures support for atomic regions is provided, users do not have to explicitly mark objects within atomic regions, and that failure-atomic regions' behavior is as expected.

While this requirement ensures that users have the support for failure-atomic regions of arbitrary size they expect, it also does not place unnecessary limitations on the language runtime and compiler. The runtime is free to perform any logging strategy and the compiler is free to reorder operations to both volatile and non-volatile memories as long as the model requirements are met.

NVM Model Requirement 3. Outside of explicit failure-atomic regions, each store to memory reachable from a durable root should be persistently completed before a new store to non-volatile memory can proceed.

This requirement helps to meet Goals 3 and 4. First, it ensures stores to durable objects must be persistently performed without explicit user instructions. Second, for a single-thread, this requirement enforces a specific ordering of stores to NVM. This allows the user to clearly reason about what values will be persisted at a given point in the execution.

To meet this requirement the runtime is responsible for inserting persist operations and fences as necessary. Like NVM Model Requirement 1, how the runtime chooses to achieve this should be implementation specific.

3.4 EXAMPLE OF RUNTIME RESPONSIBILITIES

Given programmer-labeled durable roots, I present an example of the required behavior of my model in Figure 3.1. Figure 3.1(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 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 3.1(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 my framework's requirements are met, the runtime makes the changes shown in Figure 3.1(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.

3.5 DISCUSSION

3.5.1 Model Implementation Approaches

Given the model proposed in this Chapter, many implementation choices exist. The most straightforward implementation option is to implement this model via middleware and source-to-source compilation. Via this approach, at compile time the middleware would rewrite the program's outputted bytecode to contain all of the runtime checks and actions needed by my model.

Note, however, this approach would have significant performance overheads. This is because since the language compiler is unaware of the model's semantics, the optimization potential is limited.

Another option is to directly support the model within the language compiler. Via this approach, the compiler itself will incorporate the model's semantics into its internal representation (IR), compiler optimization passes, and generated output. By doing so, more performant code can be generated. In particular, within managed language virtual machines, profile-guided optimizations can be added to minimize the amount of runtime checks and code movement needed. In Chapter 5, I discuss many compiler techniques which can be used to improve the model's performance.

3.5.2 Applicability to Statically Compiled Languages

Note that my model is designed to work optimally with managed languages, such as Java, Scala, and Kotlin, as opposed to statically compiled languages such as C and C++. A key feature of my model is to have the runtime dynamically move objects to NVM throughout execution. Unfortunately, due to casting and the pointer arithmetic semantics of the C and C++ languages, having such support is problematic. Hence, to meet Requirement 1, either (1) pointer casting and arithmetic must be banned, or (2) all objects which may be reachable from persistent memory at any point in their lifetime must initially be placed in NVM and also always abide by Rules 2 and 3.

Note that option one enables precise object pointer graphs to be constructed. This is the approach taken by some prior frameworks [17, 18]. Beyond allowing objects to be moved throughout runtime, precise object pointer graphs also allow for objects to be relocated to different virtual addresses across new executions, as highlighted by Cohen et al. [17].

Option two forgoes runtime analysis and pessimistically assumes all objects may be reachable from a durable root at all times. This has significant performance implications, since NVM is slower than DRAM, and also because all updates must be persistent. Also, this approach does not enable objects to be assigned different virtual address spaces across restarts.

3.5.3 Differentiating User-Annotated Code & Pre-existing Code

In my model the user is expected to identify durable roots and failure-atomic regions. Another key tenet of my model is that pre-existing codes are allowed to be used in persistent applications. However, one cannot expect pre-existing code to contain any persistent markings. Therefore, it is important to ensure measures are in place to handle pre-existing code correctly when it is unmarked.

Via the reachability features of my model, it is straightforward to ensure objects within pre-existing code can become persistent. Likewise, stores within the code will be persistently completed. However, in many cases it may not make sense for sequential persistency to be the default model. Therefore, I propose to automatically begin a failure-atomic region once the execution enters pre-existing code, which then continues until the execution returns to the annotated application code.

To distinguish between pre-existing and user-annotate code, the user can explicitly specify which classpaths they have annotated. Note that for Java execution the application classpaths are already expected to be provided by the user, so the additional effort needed to identify annotated classes is minimal. Given this information, unless the user has specified otherwise, by default within user-annotated code each store will be persisted in order and a failure-atomic region begins whenever a pre-existing class is entered.

3.5.4 Limitations

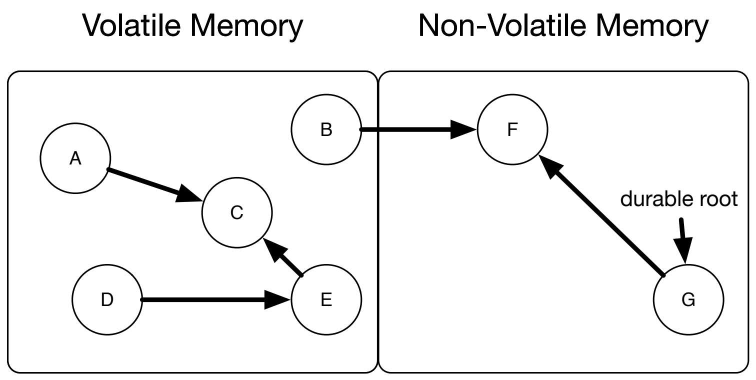
While my model substantially reduces the effort of creating persistent applications, limitations still exist. One of my model's limitations is that failure-atomic regions are not ordered between threads. This means that if two threads concurrently execute failure-atomic regions which access the same data or have causal relationships and a crash happens, the application may be recovered to an inconsistent state. To prevent this, the user must insert synchronization primitives to ensure isolation.

Another limitation of my model is handling situations when threads and other inherently transient data, such as socket connections, are reachable from a durable root. In these scenarios, either the pointer to the transient information can be set to its default value (`NULL`) or the transient process can be restarted. Ideally, a model should provide a overwritable default action which allows the user to initiate any recovery processes. This is the approach taken by Cohen et. al [17].

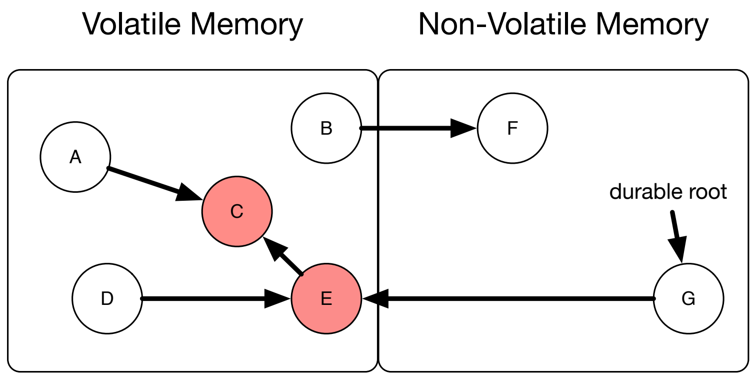
3.5.5 Alternative Persistency Models

Given the benefit of hindsight, I believe that a snapshot-based persistency model is more appropriate for a user-friendly programming model. In a snapshot-based persistency model, the user is expected to request a snapshot of the entire persistent state as needed. Taking a snapshot either requires all threads to reach a barrier or for the data to be partitioned into independent sections; however, in a snapshot model, the races described in Chapter 3.5.4 will not occur.

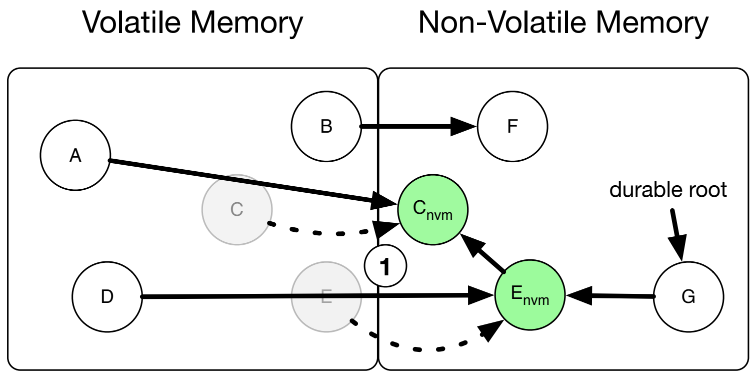
Note that snapshot-based persistency is orthogonal to how the persistent data is identified, and therefore works very well with persistence by reachability. Furthermore, since how the snapshots are created are left up to the underlying runtime, in snapshot-based persistency many opportunities for optimizations exist.



(a) Initial Heap State



(b) Model Violation



(c) Correct State Change

Figure 3.1: Changing heap state to meet the model's requirements.

Chapter 4: AutoPersist: An Easy-To-Use Java NVM Framework

4.1 INTRODUCTION

The programming model defined in Chapter 3 aims to make the process of creating persistent applications programmer-friendly by offloading much of the responsibility of NVM programs to the runtime. Because of this, runtime support is crucial to the realization of my NVM programming model.

In this chapter, I describe how to implement my NVM programming model in a Java Virtual Machine (JVM). I chose to implement my model in Java, as opposed to another language, for many reasons. First, as discussed in the prior Chapter, my model is designed for managed languages, not statically compiled languages like C/C++. Given this, Java and the JVM is the natural target; Java is the most popular managed language for enterprise applications.

Another reason for targeting the JVM for my programming model is that many other languages beyond Java also target the JVM. As will be discussed in Chapter 4.5.1, since I add persistence support at the JVM level, it is also straightforward to incorporate my model into all JVM-based languages.

I implement my NVM programming model within the Maxine JVM [22] and call my implementation *AutoPersist*, in reference to the persistence by reachability trait of my model. Within Maxine, many changes are made to support creating persistent applications. Primarily, this includes introducing a new NVM heap space and adding the runtime support to both move objects to NVM as necessary and ensure updates to objects within NVM follow the persistency model.

This chapter describes many implementation details, including how the semantics of many JVM bytecodes must be augmented to account for correct persistent updates and the movement of objects between the volatile and non-volatile heaps. To help assist the runtime in identifying an object's current state, a new header word has be added to each object to contain this information. In addition, as will be described in Chapter 5, this header word can be used to contain information needed by optimization passes.

Because Java is a multi-threaded language, care must be taken to ensure threads do not race while moving data. Therefore, this chapter also describes how multi-threaded coordination of object movement in performed within AutoPersist.

Finally, in this chapter I also explain how my model is exposed within the Java language. To meet my model's requirements, it is only necessary to enable the user to identify durable roots and failure-atomic regions. However, for debugging purposes, I also add an introspection API to allow users to query the state of a given object. In addition, this chapter also describes the recovery process in the event of a crash.

4.2 APPLYING THE PROGRAMMING MODEL TO JAVA

4.2.1 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. I add a new annotation to Java [31], `@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 AutoPersist. 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, I 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.

4.2.2 Failure-Atomic Regions

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, AutoPersist 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 [8, 66, 67, 68], AutoPersist's 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 [69]. This type of concurrency model is known as an open transactional model [70]. Chapter 4.4.5 covers how we implement failure-atomic region support in AutoPersist.

4.2.3 Persistency Model

As described in Chapter 3.3.2, 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 Chapter 4.4.1, I 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 AutoPersist's persistency model. Such data can be reordered in accordance with the Java memory model.

4.2.4 Recovery API

In order to recover data from a `@durable_root` after a crash, AutoPersist must have recovery code that allows the program to retrieve previous versions of an object as it starts-up. To allow this, AutoPersist extends 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 4.1 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.

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

Figure 4.1: Recovery API example.

4.2.5 Introspection API

A strength of my programming model 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)`.

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.

4.2.6 *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.

`@unrecoverable` may be used to limit the performance impact of persistency when objects can be recovered or recreated via other means. However, I 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.

4.3 IMPLEMENTING AUTOPERSIST WITHIN THE JVM

In this section, I describe how I implement AutoPersist the Java Virtual Machine (JVM). 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 any object as `@unrecoverable`. In such case, AutoPersist does not perform any persistency-related action on the field.

4.3.1 Modified Object Store Operations

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

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 Algorithms 4.1 and 4.2.

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 discussed in Chapter 4.4.1, 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.

Algorithm 4.1 Modified object store operations. (1/2)

```
 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 isShouldPersist(holder) and !isUnrecoverable(field) then
29:         cachelineWriteback(holder, field)
30:         if !inFailureAtomicRegion(tid) then
31:             persistFence()
32:         end if
33:     end if
34: end procedure
```

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.

Algorithm 4.2 Modified object store operations. (2/2)

```
 1: procedure ARRAYSTORE(holder, index, value)
 2:     holder = getCurrentLocation(holder)
 3:     if typeof(value) is Object then
 4:         value = getCurrentLocation(value)
 5:         if isShouldPersist(holder) and !isRecoverable(value) then
 6:             value = makeObjectRecoverable(value)
 7:         end if
 8:     end if
 9:     if inFailureAtomicRegion(tid) and isShouldPersist(holder) then
10:         logStore(holder, index)
11:     end if
12:     writeArray(holder, index, value)
13:     if isShouldPersist(holder) then
14:         cachelineWriteback(holder, index)
15:         if !inFailureAtomicRegion(tid) then
16:             persistFence()
17:         end if
18:     end if
19: end procedure
```

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 4.1 shows the 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 my 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 not in a failure-atomic region, a fence is inserted to guarantee completion of the writeback (Line 31).

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 4.2 shows the modifications.

4.3.2 Object Header

As the internal object representation is hidden from the user in Java, I modify the object layout to assist with the implementation. I add a 64-bit header word to each object, which I call the *NVM_Metadata* header. This header stores information about the state of the object relevant to AutoPersist. Figure 4.2 shows the fields in AutoPersist's object header word.

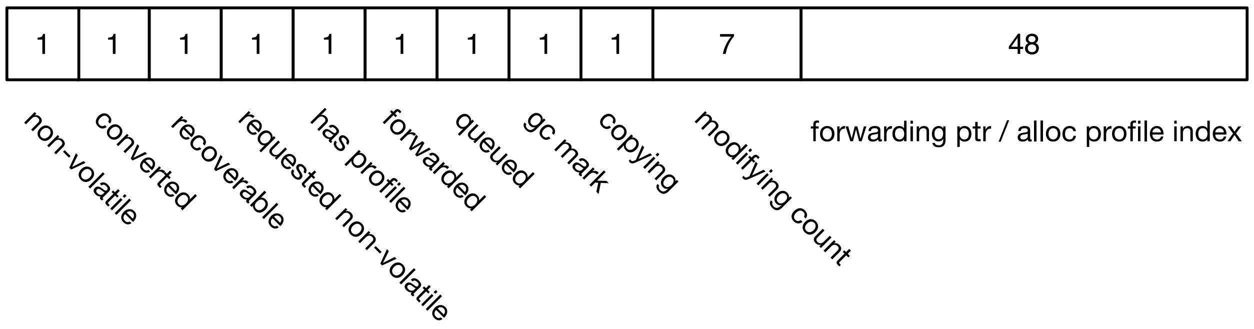


Figure 4.2: NVM_Metadata header contents.

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.

4.4 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.

4.4.1 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 3.1). However, AutoPersist adjusts the pointers *lazily*; for performance, it temporarily inserts a level of indirection for some pointers until GC occurs.

Algorithm 4.3 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
```

For example, in Figure 3.1, 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, one would have to add a pointer table, and introduce a level of indirection to all pointer accesses. Alternatively, at the time of the move, one 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 4.2). In addition, some JVM bytecodes are adjusted to check for forwarding objects.

Algorithm 4.3 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).

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, `getCurrentLocation` is called to ensure that the correct pointers are being used (Lines 8 and 11). Many of the procedures shown in Algorithms 4.1 and 4.2 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.

4.4.2 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.

Algorithms 4.4 and 4.5 shows the various procedures used for this operation. Procedure `make-ObjectRecoverable` manages the phases of the operation. First, the initial object (i.e., the one that initiates the transitive search) is passed to procedure `addToQueueIfNotConverted` (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, a *queued* bit is added in the NVM_Metadata header of each object (Figure 4.2). 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, a CAS operation is used 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 3). For each object, AutoPersist first checks whether it is already allocated in NVM. The *non-volatile* bit in the NVM_Metadata header (Figure 4.2) is set if the object is in NVM. If the bit is not set, the object is moved to NVM (Line 6). In either case, cache line writebacks must be inserted to guarantee that the entire contents of the object are persistent (Line 8). 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, AutoPersist searches all the objects that are reachable by pointers from the current object and, if necessary, add them to the work queue (Line 11). Note that fields annotated with the `@unrecoverable` marking are not searched.

Algorithm 4.4 Transitive persist (1/2).

```
 1: procedure MAKEOBJECTRECOVERABLE(object)
 2:     addToQueueIfNotConverted(object)
 3:     convertObjects()
 4:     wait for other threads to complete phase
 5:     update ptrs within NVM to objects' current locations
 6:     wait for other threads to complete phase
 7:     set recoverable flag in all objects within work queue
 8:     return getCurrentLocation(object)
 9: end procedure

10: procedure ADDTOQUEUEIFNOTCONVERTED(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
```

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 13). Recall that these updates are necessary to prevent persistent objects from pointing to volatile forwarding objects. Finally, if the object has moved, the work queue must point to the new location of the object (Line 16).

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, I observe very little wait time.

Algorithm 4.5 Transitive persist (2/2).

```
 1: procedure CONVERTOBJECTS
 2:     idx = 0
 3:     while idx != workQueue.size() do
 4:         obj = workQueue[idx]
 5:         if !isNonVolatile(obj) then
 6:             obj = moveToNonVolatileMem(obj)
 7:         end if
 8:         write back entire object to NVM
 9:         setIsConverted(obj)
10:         for (ref, offset) in nonUnrecoverableReferences(obj) do
11:             addToQueueIfNotConverted(ref)
12:             if !isNonVolatile(ref) then
13:                 ptrQueue.add(obj, offset, ref)
14:             end if
15:         end for
16:         workQueue[idx] = obj
17:         idx += 1
18:     end while
19: end procedure
20: procedure UPDATEPTRLOCATIONS
21:     while ptrQueue.size() != 0 do
22:         (obj, offset, ref) = ptrQueue.pop()
23:         ref = getCurrentLocation(ref)
24:         writeOffset(obj, offset, ref)
25:     end while
26: end procedure

27: procedure MARKRECOVERABLE
28:     idx = 0
29:     while !workQueue.isEmpty() do
30:         obj = workQueue.pop()
31:         setRecoverable(obj)
32:     end while
33: end procedure
```

The next step is to call procedure `updatePtrLocations` to update all pointer locations within the ptr queue (Lines 22 to 24). 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 AutoPersist's three object states to traditional tri-color GC terms [71], 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.

4.4.3 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, two new fields are added to the NVM_Metadata header: *copying* and *modifying count* (Figure 4.2). 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 4.6 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, I include two optimizations. First, while an object is being copied, I 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 I 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.

Algorithm 4.6 Moving object to NVM.

```
procedure MOVETONONVOLATILEMEM(obj)
    newObj = allocateNVM(sizeof(obj))
    while true do
        do
            oldHeader = readPersistentHeader(obj)
            if getModifyingCount(oldHeader) $< 0$ then
                continue
            end if
            newHeader = setIsCopying(oldHeader)
        while !CAS(obj, oldHeader, newHeader)
        copyMem(obj, newObj, sizeof(obj))
        do
            oldHeader = readPersistentHeader(obj)
            if !isCopying(oldHeader) then
                continue
            end if
            newHeader = unsetIsCopying(oldHeader)
        while !CAS(obj, oldHeader, newHeader)
        return newObj
    end while
end procedure
```

4.4.4 Allocation and Garbage Collection

Since there are now volatile and non-volatile portions of the heap, AutoPersist's runtime 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, AutoPersist 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.

4.4.5 Failure-Atomic Region Support

As described in Chapter 4.2.2, AutoPersist supports failure-atomic regions. Given the semantics of my model, there is much flexibility in choosing how to design the implementation. Currently, AutoPersist uses per-thread undo logs with write-ahead logging. As shown in Algorithm 4.1 (Lines 9, 25, and 10), 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.

4.5 DISCUSSION

4.5.1 Applying Model to Other JVM-Based Languages

Many other languages, such as Kotlin, Scala, and Clojure, are also designed to run on top of Java Virtual Machine implementations. Because of this, it is straightforward to use AutoPersist to create persistent applications in these languages as well.

Since AutoPersist augments the JVM bytecode semantics, other languages can also reap its benefits. It is only necessary for each language to ensure that the `@durable_root` and optionally the `@unrecoverable` keywords are available within the language. Additionally, the Recovery and Introspection APIs described in Chapters 4.2.4 and 4.2.5 can be defined to better match the given language's flavor; however, this is not strictly necessary, as traditionally other JVM-based languages are able to directly invoke Java APIs.

4.5.2 Applying Model to Scripting Languages

Beyond the JVM ecosystem, another potential target for my NVM programming model is scripting languages, such as JavaScript, Python, Ruby, and Perl. While scripting languages share many traits with Java, two of their common traits could have an impact on the implementation strategy. First, scripting languages are single-threaded. This would substantially simplify the NVM support needed, as one does not need to worry about data races.

The second important scripting language trait is that they are dynamically typed. This means that at compile type the layout of an object is unknown. Traditionally, scripting language runtimes create *inline caches* [58] and a internal typing system of hidden classes to help ensure efficient execution. With this infrastructure already in place, an opportunity exists to expand the hidden class type system to also include a notion of persistency. Doing so would help to ensure the efficient execution of my NVM programming model within scripting language runtimes.

4.5.3 Alternative JVM Implementations

I choose to implement AutoPersist within Maxine due to Maxine's modularity and origins as a research JVM. However, presently the most popular JVM implementation is HotSpot [72, 73]. HotSpot outperforms Maxine mainly due to its more advanced garbage collection and compilers. Hence, to fully realize my model's performance potential AutoPersist should be applied to HotSpot.

Unfortunately, HotSpot is a very old and complex code base; the original HotSpot implementation was built around 2000. Over time, as new features have been added to the JVM and performance improvements have been made, the technical debt within HotSpot has only increased. Luckily, recently Oracle Labs has undertaken efforts to build both a new compiler and ahead-of-time runtime system. The new compiler is called Graal [74, 75] while the new runtime system is called SubstrateVM [76]. SubstrateVM aims to be as fast as HotSpot while also having a much cleaner code base written in Java itself. Because of this, I believe SubstrateVM is a good target for AutoPersist.

Another reason SubstrateVM is an ideal target for AutoPersist is that it is designed to build Java applications *ahead-of-time*. This means that classes cannot be dynamically loaded throughout execution but instead must be known during SubstrateVM's ahead-of-time analysis. As discussed in Espresso [11], one complication of creating persistent applications in Java is deciding where to store class metdata. However, this is not an issue within SubstrateVM due to it being stored ahead-of-time within the application's executable.

4.5.4 Opportunities for Hardware Improvements

As described in Chapter 4.3 and shown in Algorithm 4.1 and 4.2, to implement sequential persistency in Java on x86-64 a `CLWB` and `SFENCE` is needed after every field update to an object reachable from a `@durable_root`. Given NVM's current characteristics, this can have a significant performance impact.

Recently, proposals have been made to extend Intel's asynchronous DRAM refresh domain (ADR) [28] to include all hardware caches. In such a system, commonly referred to as *eADR*, added capacitors store enough power to allow for the processor's caches to be flushed out to memory before the system loses power. In such a system, it is now no longer necessary to insert cache maintenance instructions and fences to ensure data reaches the persistent domain. Furthermore, since x86-64 processors follow a TSO memory model, then by default stores to persistent memory are sequential. Hence, persistent caches would significantly benefit AutoPersist.

Chapter 5: Improving AutoPersist's Performance Through Profile-Guided Optimizations

5.1 INTRODUCTION

While AutoPersist faithfully implements the NVM programming model proposed in Chapter 3, much performance is left on the table. This is because the runtime must correctly handle all corner cases which may arise throughout the application's execution. This results in many *persistence checks* and conditional runtime actions being placed within the generated code.

I define persistence checks as the process needed to determine if a persistence runtime action, such as moving an object to NVM, needs to be taken. While many persistence checks must be performed throughout execution, I find that often the result of the persistence checks are very predictable. Therefore, they are an ideal candidate for profile-guided optimizations.

In this chapter, I propose to leverage the multi-tiered nature of JVM compilers to reduce the overhead of persistence checks via profile-guided optimizations. Specifically, during the execution warmup phase, profiling information is collected about the behavior of each persistence check. Later, when optimized code is generated, this profiling information is read to *bias* the generated persistence check code to favor its expected behavior. Additionally, I am able to further reduce persistence check's overhead by *speculative* eliminating code paths unlikely to be taken.

Another overhead of AutoPersist's runtime is the time spent moving objects from DRAM to NVM. In AutoPersist's default implementation, initially all objects are allocated in DRAM and are only moved to NVM once they become reachable from a durable root. However, I observe that in many situations soon after an object is allocated it becomes reachable from a `@durable_root`; this is inefficient since two copies of the object must be made. To prevent this from happening, in this chapter I also describe an optimization to *eagerly* allocate objects in NVM.

In a similar manner to persistence checks, profiling can be used to identify allocation sites which allocate objects soon to become part of the persistent state. Once these sites are identified, the code is regenerated to directly allocate objects in NVM at this sites instead of in DRAM. Therefore, once the object becomes part of the durable state it does not need to be moved. Furthermore, this helps performance by reducing the number of forwarding objects present within the code. Further details of this optimization are also explained in this chapter.

5.2 AUTOPERSIST PERFORMANCE OVERHEADS

AutoPersist has the potential to dramatically ease the programming burden of creating persistent applications. However, to facilitate widespread adoption, its performance must be close to other manually optimized NVM frameworks. One significant overhead is the persistence checks required by AutoPersist around accesses to both persistent and non-persistent objects. Before I discuss the proposed solution to reduce the persistence checking overhead, in this section I describe in more detail the overhead of AutoPersist's persistence checks.

5.2.1 Store Field Persistence Checks

As described in Chapter 4.3.1, many persistence runtime actions in AutoPersist are dependent on whether an object is reachable from a durable root. Since these actions can be very expensive, it is beneficial to ensure that they are only enabled when absolutely necessary.

Because of this, AutoPersist includes many checks, called *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*.

Algorithm 5.1 store operation with persistence checks.

```
1:  procedure PUTFIELD(holder, field, value)
    [Start Persistence Check Code]
2:      [Persistence Check]
3:      if isShouldPersist(holder) or isForwarded(holder) then
4:          [Persistence Check's Guarded Action]
5:          if isForwarded(holder) then
6:              holder = getForwardingPtr(holder)
7:          end if
8:          if isForwarded(value) then
9:              value = getForwardingPtr(value)
10:         end if
11:         if !isUnrecoverable and !isRecoverable(value) then
12:             value = makeObjectRecoverable(value)
13:         end if
14:         if inFailureAtomicRegion(tid) and !isUnrecoverable(field) 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 isShouldPersist(holder) and !isUnrecoverable(field) 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
```

To better understand the concept of persistence checks, let us revisit the actions covered in Chapter 4.3.1. Algorithm 5.1 reshows the same process, but rewritten in a way to better highlight the concept of persistence checks.

Both before and after the original write (line 18), many actions are needed. Without persistence support, the function `storeField(holder,field,value)` writes value *V* into field *F* of the holder object *H* (line 27). 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 in the ShouldPersist state. If *H* is a forwarding object, its forwarding address will point to a persistent object. As explained in Chapter 4.4.1, forwarding 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, only the checks on lines 3 and 20 are considered to be *persistence checks*, as these are the checks which guard runtime actions needed when interacting with persistent objects.

5.2.2 Load Field Persistence Checks

In addition to modifying object store procedures, AutoPersist must also add a persistence check to load procedures to ensure pointers do not refer to forwarding objects. Algorithm 5.2 revisits the process covered previously in Chapter 4.4.1.

Without persistence support, `loadField(holder,field)` loads the value *V* from field *F* of

Algorithm 5.2 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 = getForwardingPtr(holder)
6:     end if
   [End Persistence Check Code]
7:     value = readField(holder, field)
8:     return newValue
9: end procedure
```

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 Chapter 5.2.1, only objects which are persistent can have forwarded objects.

5.3 CHARACTERIZING PERSISTENCE CHECKS

As shown in Chapter 5.2, AutoPersist requires many persistence checks to guard runtime actions. In this section I first evaluate the overhead of these persistence checks. Afterwards, I profile the activation behavior of persistence checks across various persistent applications.

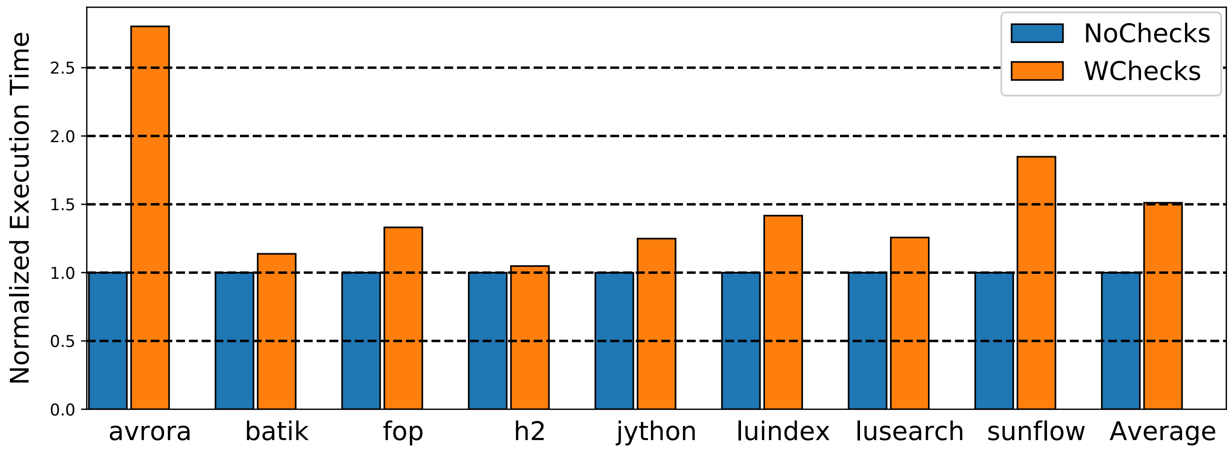


Figure 5.1: DaCapo persistence check overhead.

5.3.1 Overhead of Persistence Checks

As shown in Algorithms 5.1 and 5.2, persistence checks are used frequently in AutoPersist. However, the actions guarded by these checks are only activated if the holder object being ac-

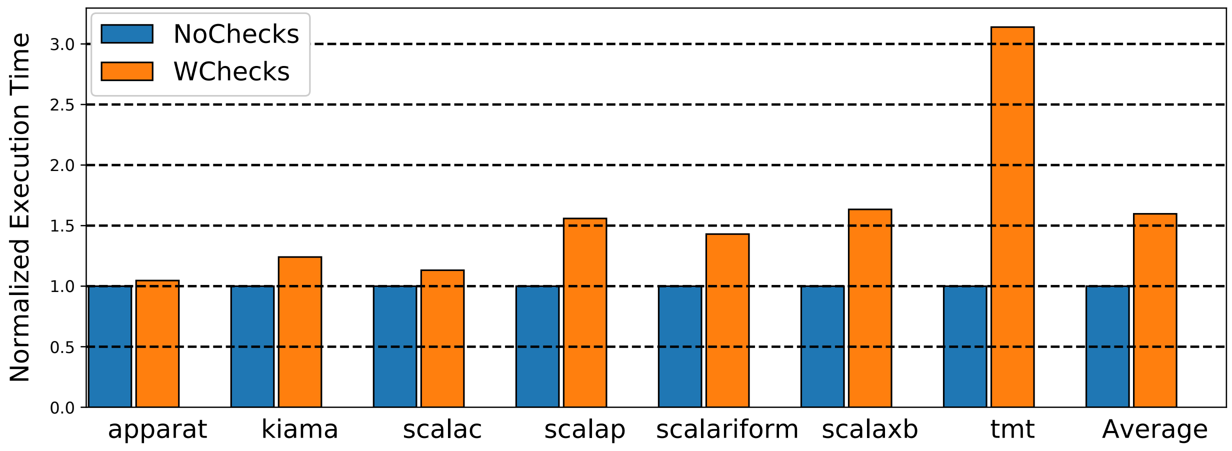


Figure 5.2: Scala DaCapo persistence check overhead.

cessed is reachable from the durable root set. Otherwise, for non-persistent holder objects, the actions are bypassed. To evaluate the cost of persistence checks guarding bypassed actions, I run the DaCapo [77] and Scala DaCapo Benchmark Suites [78]. 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. I evaluate two configurations: *NoChecks*, which does not have persistence checks, and *WChecks*, which includes the persistence checks described in Chapter 5.2. More details about the evaluation environment are in Chapter 6.

Figures 5.1 and 5.2 show the performance of the two configurations. In the worst performing benchmark, Scala DaCapo's `tmt` benchmark, *WChecks* is 214% slower than *NoChecks*. On average, I find 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 AutoPersist can significantly simplify the process of creating persistent programs, its overhead must be minimal, compared to other NVM frameworks. Clearly, the overheads of persistence checks must be drastically reduced before AutoPersist can gain widespread acceptance.

5.3.2 Persistence Check Activation Behavior

As described in Chapters 5.2.1 and 5.2.2, 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, I monitor the behavior of persistence checks across two persistent applications. The persistent applications are two versions of a persistent key-value store. Each version is based on QuickCached [79], a pure Java implementation of memcached [80], and uses a different persistent data structure internally for its key-value storage. Specifically, *Functional HashMap (Func)* uses a functional hash map as its backend, and *Hybrid B+Tree (JavaKV)* uses a B+ tree where only the leaf nodes are persistent. Chapter 6.3 contains more details about the persistent applications.

I run each persistent application with the Yahoo! Cloud Serving Benchmarks [81] and show the aggregated results. For each run, I monitor the dynamic number of persistence checks encountered, and the number of persistence checks guarding activated actions. In addition, I 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.

	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
Func	2708.4	6.4	0.24	0.03	0.05	99.86
JavaKV	2730.9	5.6	0.20	0.09	0.02	99.87

Table 5.1: Persistence check statistics when running YCSB workloads.

The profiling results are shown in Table 5.1. The results highlight 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, I 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 Chapter 5.2.2, 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 Chapter 4.4.1, forwarded objects exist only temporarily before a garbage collection cycle occurs. The results shown Table 5.1 indicate that forwarded pointers are seldomly used.

Overall, the 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.

5.4 BIASING PERSISTENCE CHECKS

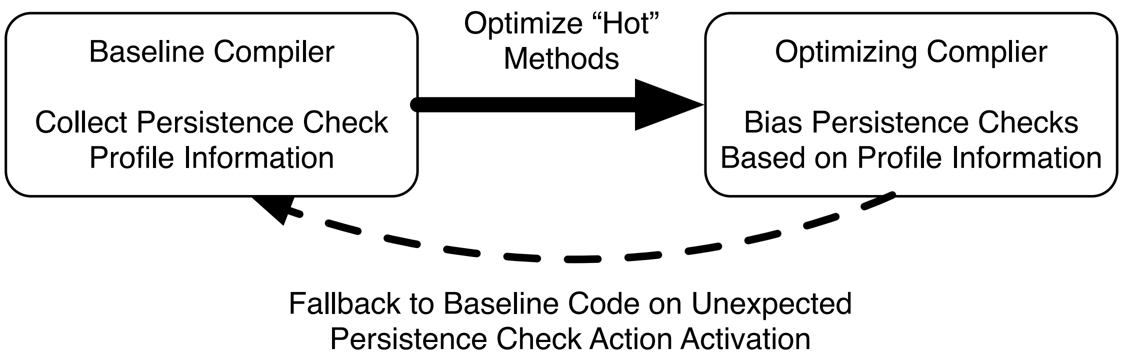

Baseline Compiler
Collect Persistence Check Profile Information
Optimize "Hot" Methods
Optimizing Complier
Bias Persistence Checks Based on Profile Information
Fallback to Baseline Code on Unexpected Persistence Check Action Activation

Figure 5.3: Overview of QuickCheck.

The previous section highlighted 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, I propose to transform individual 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, I propose to speculatively remove the actions guarded by the check. I name the solution QuickCheck. The following subsections describe QuickCheck's technical details.

5.4.1 Overview of QuickCheck

To minimize the overhead of persistence checks, I propose to *bias* each check to be specialized for its common case. Specifically, I 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 Chapter 2.4, QuickCheck can use these multiple code generation phases as its profiling and biasing phases. Figure 5.3 shows how QuickCheck can be added to the traditional JVM compilation phases. Initially, when code is generated by the baseline compiler, the compiler is augmented to also generate profile information about each persistence check site. For each persistence check site, the runtime records 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

```
1. persistence_check;
2. jmp_if_false Post_Action;

3. guarded action

4. label Post_Action:
5. code after guarded action
```

(a) Likely activation of action guarded by persistence check

```
1. persistence_check;
2. jmp_if_true Action_Activation;

3. label Post_Action:
4. code after guarded action
…
5. return;

6. label Action_Activation:
7. guarded action
8. jmp Post_Action;
```

(b) Unlikely activation of action guarded by persistence check

```
1. persistence_check;
2. jmp_if_true Action_Activation;

3. code after guarded action
…
4. return;

5. label Action_Activation:
6. jump to baseline code;
```

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

Figure 5.4: Code generation strategies for persistence checks in AutoPersist.

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, QuickCheck's 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 5.3.

5.4.2 Persistence Check Biasing Strategies

As discussed, the optimizing compiler can use the persistence check profile information collected during warm-up to generate better code. This section describes how to bias the generated code. I divide the likelihood of a check's guarded action being activated into two categories: *likely* and *unlikely*. Figures 5.4 (a) and (b) show how QuickCheck generates code in these two cases. Figure 5.4 (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 [6].

Figure 5.4 (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.

As discussed above, the optimizing compiler can use the persistence check profile information collected during warm-up to generate better code. This section describes how to bias the generated code. I divide the likelihood of a check's guarded action being activated into two categories: *likely* and *unlikely*. A given persistence check's category dictates how the compiler will generate code for it. For persistence checks guarding actions likely to be activated, the action code is placed on the fall-through path; on the other hand, unlikely to be activated actions can be moved off of the main execution path down to the end of the method.

5.4.3 Speculatively Removing Action Routines

As shown in Chapter 5.3.2, 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 5.4 (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 5.4 (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 interfere with the optimizations. As shown in Algorithms 5.1 and 5.2, 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.

5.5 IMPLEMENTING BIASED PERSISTENCE CHECKS

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

5.5.1 Persistence Check Profile Collection

As shown in Figure 5.3, 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, I modify the baseline compiler to record profiling information at each persistence check.

To store the profiling information, I 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. The proper counter is incremented depending on whether the check's guarded action is activated or bypassed.

While two persistence checks surround each write operation (see Chapter 5.2.1), only the persistence check before the write must be profiled. 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 5.1. 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.

5.5.2 Compiler Optimization Pass

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, checks are categorized 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. the following describes how the persistence checks added to both load and store operations are optimized based on their biasing state.

Load Operation Persistence Check Generation

As shown in Chapter 5.2.2, 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 5.2. 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 [74]. 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 that for other popular compilers such as LLVM, HotSpot C2, and GCC, a similar mechanism also exists.

Algorithm 5.3 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
```

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 5.3 shows a simplified version of this persistence check. If line 3 is reached, then a signal is raised. This point is registered as a *deoptimization point*, where execution transfers to the baseline compiler. To catch raised signals, I modify the runtime to install a handler which performs an on-stack-replacement [57] to transfer execution to the baseline compiler in the event of a raised signal.

Store Operation Persistence Check Generation

As shown in Algorithm 5.1, stores originally have persistence checks both before and after the write operation. For checks in the *likely*, *unbiased*, and *unlikely* state, I adjust the weight of the true branches like in Chapter 5.5.2. However, instead of biasing all the branches within the guarded action code, I 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. I chose not to profile the behavior of internal branches in the guarded action code due to the code's low activation rate, as shown in Chapter 5.3.2.

For *very_unlikely* biased persistence checks, I 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 5.4 store with *very_unlikely* biased check.

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

Algorithm 5.4 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.

5.6 OPTIMIZING OBJECT ALLOCATION

In AutoPersist, beyond reducing the overhead of persistence checks, I am also able to leverage the JVM's tiered compilation strategy to reduce the number of objects which must be moved to NVM. Specifically, a source of overhead in AutoPersist's 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 accomplish this, I modify the initial compiler to produce profiling information that is used by the optimizing compiler to reduce object handling overhead. 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 4.2) 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 allocProfile 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 (Chapter 4.4.4) 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 my NVM model's requirements. This profiling information simply helps to attain higher performance.

Beyond the reduction of copying objects between DRAM and NVM, another benefit of this optimization is the reduction of forwarding objects created. Since there are fewer forwarding objects, less runtime actions will need to be taken, and hence more persistence checks will be classified as either *unlikely* or *very_unlikely*, which will further improve performance.

5.7 DISCUSSION

5.7.1 Additional Opportunities for Removing Checks

Via QuickCheck, the runtime is able to successfully remove much of the runtime check overhead throughout execution. However, since I am leveraging profiling information, for correctness it is still necessary to validate assumptions made by the runtime.

Instead of using profiling-based analysis, it is also possible to perform formal analysis passes to guarantee application behavior. For instance, via class hierarchy analysis [72], reachability analysis [76], and pointer analysis [82, 83], it may be possible to prove a given load/store location either always interacts with a volatile object or durable object. Via this analysis, then no longer would checks have to be present at sites with known behavior; instead, any needed actions could be directly encoded in the generated code.

Leveraging escape analysis is another way to remove runtime checks. Escape analysis [65, 84] is used to identify objects whose lifetime is limited to its nested functional scope and therefore can be allocated in the stack instead of the heap. By virtue of its characteristics, any object which is identified by escape analysis as a candidate for stack allocation will never become reachable from a `@durable_root`, and hence will not need any runtime checks.

Beyond removing unnecessary runtime checks, proving an object will never become part of the persistent state also enables additional compiler optimizations. For instance, provably volatile objects do not need to follow the sequential persistency semantics of my model; instead, in accordance with the Java Memory Model [69], compiler optimization are able to eliminate redundant memory operations as well as move around memory operations to achieve better performance.

5.7.2 Improving Eager NVM Allocation

In Chapter 5.6, I described how profiling can be used to identify sites which would benefit from eagerly allocating objects in NVM. While this is an effective optimization, additional improvements can be made. Because NVM decisions are made at an allocation-site granularity, allocation sites which see both persistent and volatile objects (i.e. are *persistency-polymorphic*) cannot eagerly allocate objects in NVM. To prevent persistency-polymorphic sites from being created, either aggressive inlining or specialization can be added to the generated code. Inlining reduces the number of persistency-polymorphic allocation sites by allowing different execution paths to have private allocation sites, as opposed to sharing the same code. Likewise, method specialization can be performed to help split executions based on their expected allocation behavior.

In addition to more precisely allocating eagerly objects in NVM, the runtime overhead can be reduced if these objects are directly set to being recoverable, as opposed to merely being in NVM. Currently, I chose not to directly set these objects as being recoverable due to the overhead of persistency actions. However, as discussed in Chapter 4.5.4, if caches become persistent, then most of the cost of enforcing sequential persistency is eliminated. In this scenario, directly setting eagerly allocated objects as recoverable is likely to be beneficial.

5.7.3 Profile-Guided Support for Logging

In my programming model, the implementation is free to choose how to provide failure-atomic support. Currently, in AutoPersist I use undo logging in all scenarios. However, as described by Marathe et al. [85], depending on the usage pattern, different logging strategies are optimal. Therefore, opportunities exist to tune each failure-atomic region to use either undo logging, redo logging, or copy-on-write based on its characteristics.

Deciding which logging strategy to use is an ideal candidate to be optimized by the JVM's profile-guided optimization support. During warmup, various metrics about each failure-atomic region, such as the log size, the number of data structures modified, and the number of reads after writes to persistent data, can be collected. Then, based on this information, the optimizing compiler can choose which type of logging should be implemented for each failure-atomic region.

Note that any type of logging chosen will result in correct execution; profiling and specialization would merely serve to improve the performance of logging.

5.7.4 Runtime Support for Caching and Redundancy

Current NVM offerings implement advanced schemes to ensure data integrity and prevent wearout. For instance, to guard against permanent cell failures and enable even wear, Intel Optane DC persistent memory modules use error code correction (ECC) algorithms and also have on-DIMM logic to move around data blocks. However, even with such measures in place, there is no guarantee that a permanent fault will not occur and leave all data within the module unrecoverable.

Likewise, currently NVM is significantly slower than DRAM. In a recent paper [86], the authors found that NVM's read latency is about $3\times$ higher than DRAM and its read and write bandwidths max out at 39.4 GB/s and 13.9 GB/s, respectively. This means that applications with objects in NVM will experience a significant slowdown, even if the data is rarely overwritten.

To combat both problems, AutoPersist could automatically create multiple copies of persistent objects in different DIMMs. For resiliency, each copy could be in a different NVM DIMM whereas, for performance, read-mostly data could be cloned into DRAM. The main issue with creating these copies is keeping track of where all copies of the object reside and ensuring they remain consistent. However, in AutoPersist this also can be managed by the runtime. For instance, AutoPersist could assign each clone a virtual address that facilitates *colored* object pointers, where pointers to different copies of the object differ only by a set of tag bits. Note that colored pointers already are used by concurrent garbage collectors to help store phase information within pointers [87]. In addition, AutoPersist could add information to each object's header to help identify whether multiple copies of an object exist.

Chapter 6: Evaluating AutoPersist on Real Hardware

6.1 INTRODUCTION

In this chapter I evaluate my new NVM programming model and AutoPersist. I compare it against existing Java NVM offerings in terms of both programmability and performance. In addition, I demonstrate how the optimizations described in Chapter 5 improve the performance of AutoPersist.

6.2 INFRASTRUCTURE

6.2.1 Compiler Platform

I implement the AutoPersist framework described in Chapter 4 along with the optimizations described in Chapter 5 within the Maxine JVM [22]. Maxine is an open-source research JVM that enables the fast prototyping of new features while achieving competitive performance. I use Maxine 2.0.5, and modify both its initial tier compiler (T1X) and its optimizing compiler (Graal).

In addition, I modify its object layout to integrate AutoPersist's NVM_Metadata header (Figure 4.2), add new NVM heap regions, extend its GC (Chapter 4.4.4), and implement failure-atomic regions (Chapter 4.4.5).

I modify Maxine's baseline compiler (T1X) to collect the profiling information described in Chapter 5, and modify Maxine's optimizing compiler (Graal) [74] to use the proposed persistence check biasing and eager NVM allocation techniques. In addition, Maxine is augmented to include the handlers needed to handle the mispeculation of persistence checks biased to the *very_unlikely* state.

6.2.2 Server Platform

All tests are run on a server with 12 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. Each node has 6 memory channels, each attached to both NVM and DRAM.

In all experiments, AutoPersist reserves 20GB for each of the volatile and non-volatile heap spaces. To create the non-volatile heap, AutoPersist uses `libpmem` [8] 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. Intel's cache line writebacks (`CLWB`) and store memory fences (`SFENCE`) are used to persist values.

Note that currently I have the non-volatile heap space mapped entirely to 1 NVM DIMM. However, it is possible to stripe the NVM heap space across multiple DIMMs to improve performance [86].

6.3 APPLICATIONS

To evaluate AutoPersist and my optimizations, I perform experiments on a persistent key-value store, several persistent kernels, and two existing Java benchmark suites. In the following subsections I describe the applications used for evaluation, as well as the configurations used to evaluate AutoPersist.

6.3.1 Java Benchmark Suites

To evaluate the effectiveness of QuickCheck in reducing the overhead of persistence checks guarding bypassed actions, I run most of the DaCapo [77] and Scala DaCapo [78] benchmark suites on AutoPersist. Both benchmarks suites are commonly used to evaluate the performance of JVM implementations. To measure the optimal performance of the applications, each benchmark is run several times (with the same warm-up counts as used in [88]) before measuring the execution time.

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.

6.3.2 Persistent Key-Value Store

I implement a persistent version of a key-value store using AutoPersist. Specifically, I modify QuickCached [79], a pure Java implementation of Memcached to use persistent data structures internally for its key-value storage. The different backends compared are:

- **IntelKV.** This is Intel's `pmemkv` library [89], 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 work [34], 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 [90] and is implemented in Java. In my implementations the entire datastructure is in persistent memory.
- **JavaKV.** This backend uses the same B+ tree structure as in Intel's pmemkv library [89] *kvtree3* backend, except written in Java. Specifically, this configuration uses a hybrid B+ tree. Similar to existing works [34], in this implementation, only the leaf nodes are in persistent memory.

To evaluate the performance of these backends, I use the Yahoo! Cloud Serving Benchmark (YCSB) [81], a benchmark suite commonly used to evaluate the performance of cloud storage services. I 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, I perform five hundred thousand operations.

6.3.3 Persistent Kernels

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 6.1: Description of persistent data structures.

To isolate the behavior of AutoPersist from the effects of large applications, I also create kernels which perform a random collection of reads, writes, inserts, and deletes against five typical persistent data structures. The benchmarks are listed in Table 6.1. MArray, MList, and FARArray are hand-written to ensure correct persistent operation. FArray and FList are functional data structures from the PCollections library [90], and inherently use persistence-safe structures.

6.4 IMPLEMENTATION FRAMEWORKS

To compare AutoPersist against existing NVM frameworks, I also created my own implementation of Espresso [11]. *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. I 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.

To evaluate the persistent key-value store and kernels, I have implemented all Java-based backends and data structures in both AutoPersist and Espresso. Note that for Espresso this required creating a new persistent version of the PCollection library whereas AutoPersist is able to use the original unmodified PCollection library. In addition, I do not run the DaCapo and Scala DaCapo on *Espresso*, as these applications only serve to evaluate the runtime overheads of AutoPersist.

6.5 CONFIGURATIONS

6.5.1 QuickCheck Evaluation

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 5.4(a).
Unlikely: Bias all persistence checks to the *unlikely* state and represent them as unlikely taken branches as shown in Figure 5.4(b).
Deopt: Bias all persistence checks to the *very_unlikely* state and represent them as shown in Figure 5.4(c).
QuickCheck: Use QuickCheck to dynamically predict persistence check activation behavior and bias them accordingly.

Table 6.2: Configurations evaluated.

To evaluate the performance of my persistence check biasing optimizations, I test multiple configurations. The different configurations are shown in Table 6.2. The *Clean* configuration is the unmodified Maxine JVM, which does not have persistent support. The *Unbiased* configuration is the AutoPersist implementation, but without 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 the profiling and biasing techniques proposed in the prior Chapter.

6.5.2 AutoPersist vs. Espresso

Framework	Description
NoProfile	AutoPersist without the profiling opt. of Chapter 5
T1X	*NoProfile* but only using the initial tier compiler (T1X)
T1XProfile	*T1X* plus collecting the profiling info of Chapter 5
AutoPersist	Complete AutoPersist
Espresso	Implementation of Espresso [11]

Table 6.3: Frameworks evaluated.

Table 6.3 shows the different AutoPersist-based NVM frameworks used in the evaluation against Espresso. *NoProfile* is AutoPersist without the profiling optimizations described in Chapter 5. *T1X* is *NoProfile* but only using the initial tier compiler (T1X). *T1XProfile* is *T1X* plus collecting the profiling information described in Chapter 5. 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. *Espresso*'s implementation is described in Chapter 6.4.

6.6 PROGRAMMABILITY

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 performance. This is in contrast to *Espresso*, which needs explicit markings for each persistent object allocation, cache line writeback to NVM, and fence [11].

Framework	Applications		Kernels					Total Markings
	Func	JavaKV	MArray	MList	FARArray	FArray	FList	
AutoPersist	4	6	1	1	5	1	1	19
Espresso	55	45	49	48	63	47	14	321

Table 6.4: Number of markings for memory persistency.

Table 6.4 shows the number of markings added for each application when using AutoPersist and *Espresso*. Table 6.4 shows that AutoPersist only needs 19 markings in total. This includes the durable root markings, failure-atomic region labels, and also all unrecoverable markings added. In contrast, when using *Espresso*, the programmer needs to add 321 markings in the programs to ensure crash consistency.

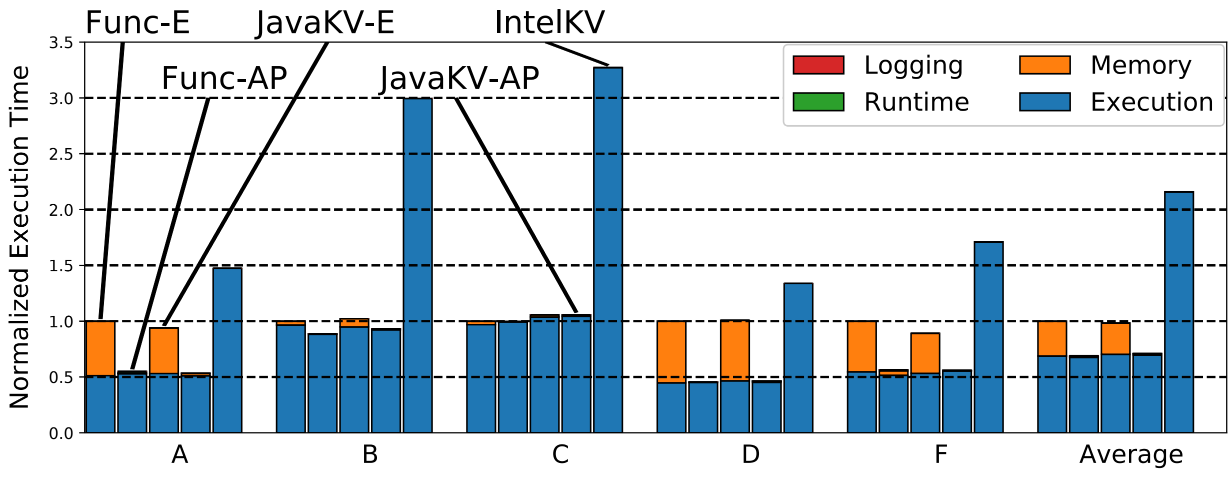


Figure 6.1: Key-value store execution time.

There is a significant difference in the number of markings required in the two frameworks. However, in my experience, even this difference does not do justice to the fact that I 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.

6.7 FRAMEWORK COMPARISON

6.7.1 Key-Value Store

Figure 6.1 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. A bar is also shown for *IntelKV*. The execution time is normalized to *Func-E*. The execution time is broken down 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 4.4). *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 one cannot instrument; all its time is *Execution*.

Looking at the Average bars, one 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 the *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 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 6.1 also shows that the *Logging* and *Runtime* times in AutoPersist are negligible. Importantly, the *Runtime* overhead is negligible because of the efficiency of AutoPersist's 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.

6.7.2 Persistent Kernels

Figure 6.2 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. 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 Chapter 6.7.1, AutoPersist

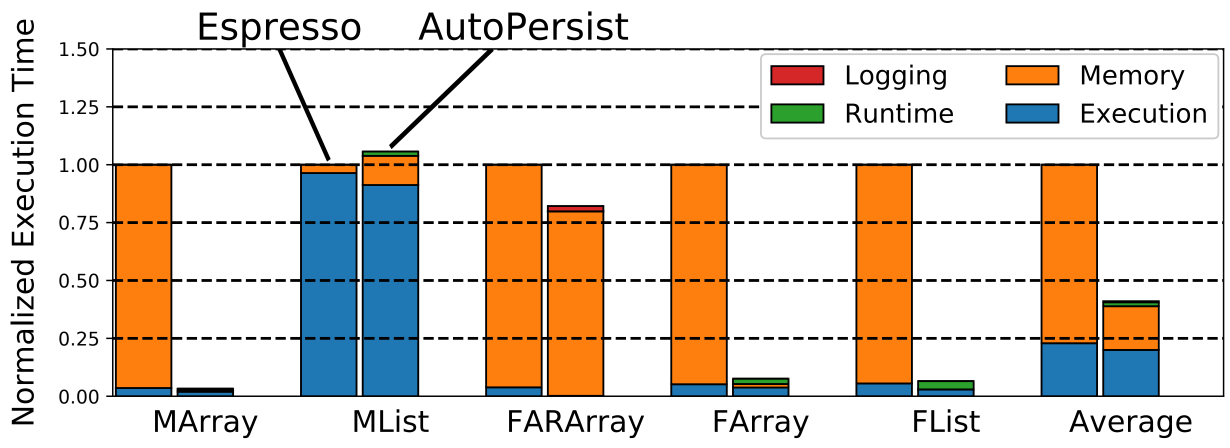


Figure 6.2: *Espresso* and *AutoPersist* kernel execution time.

inserts the minimal number of CLWBs necessary to ensure that objects reachable from the durable root set are persistent.

Unlike other kernels, 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.

6.8 ANALYSIS OF AUTOPERSIST'S PERFORMANCE

6.8.1 Persistence Check Overhead in Volatile Applications

Figures 6.3 and 6.4 show the execution time of the DaCapo and Scala DaCapo benchmarks when using the configurations described in Table 6.2. 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 AutoPersist is able to successfully profile persistence check sites.

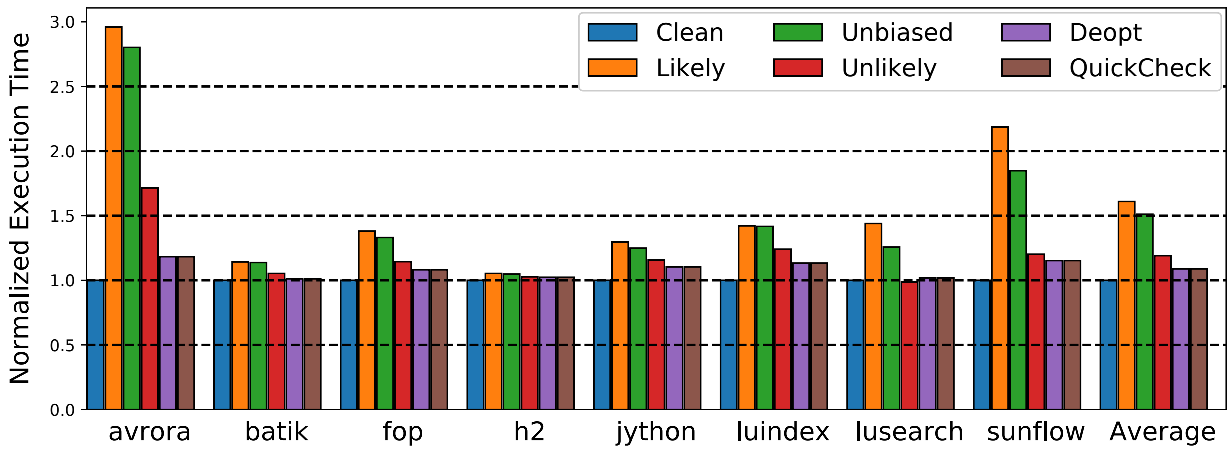


Figure 6.3: DaCapo execution time.

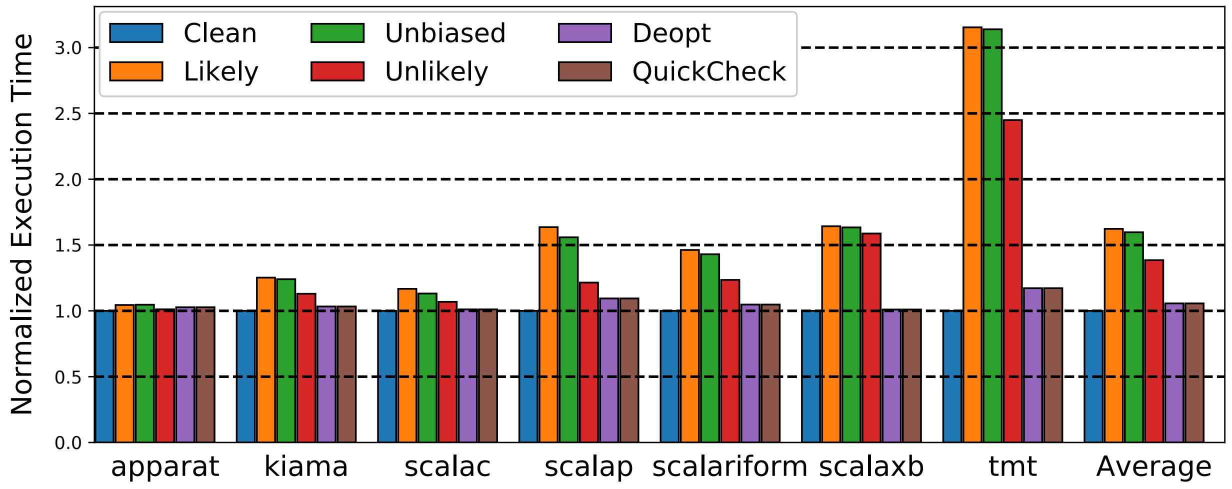


Figure 6.4: Scala DaCapo execution time.

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.

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, I profiled every object header read in AutoPersist. I recorded the memory address accessed and compared it to the memory address of the object field accessed next. I 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.

6.8.2 Persistence Check Overhead in Persistent Applications

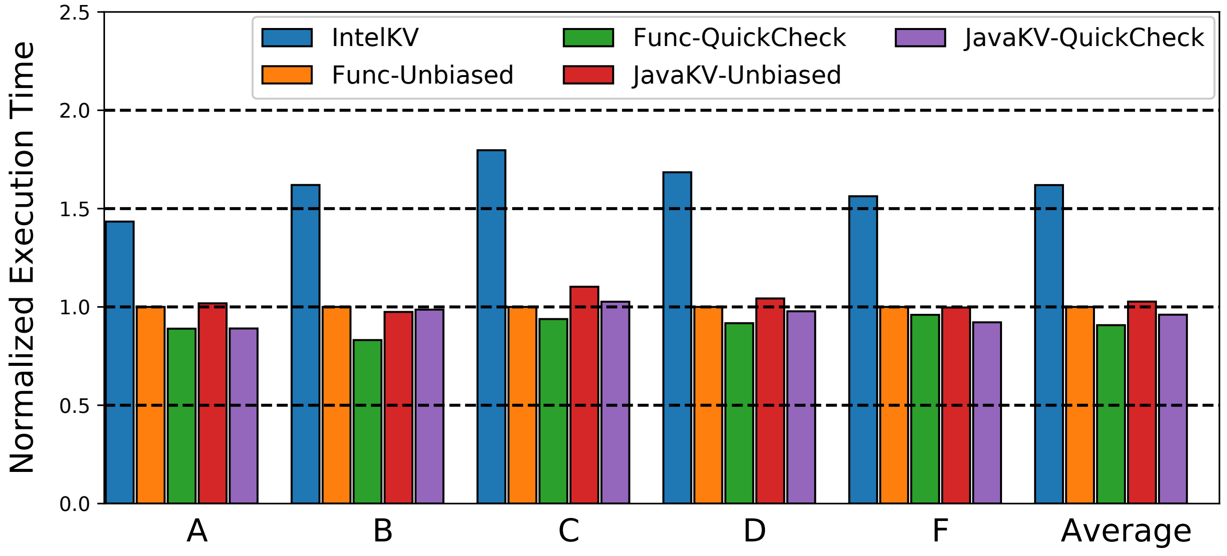


Figure 6.5: Persistent YCSB execution time.

Key-Value Store. Figure 6.5 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 Func with *Unbiased*. Two traits are noticable in the figure. First, all AutoPersist storage engines outperform IntelKV. On average, the execution time of IntelKV is 62.0% and 57.7% higher than the *Unbiased* configurations of Func, and JavaKV, respectively. This is because, when using IntelKV, objects must be serialized and transferred to IntelKV's C++ backend. In AutoPersist, 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 my biasing techniques improves performance. On average, the *QuickCheck* configurations of Func and JavaKV reduce the execution time by 9.3% and 6.5% compared to their respective *Unbiased* configurations. On average, this is an 7.9% 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 achieved without any new hardware or user involvement.

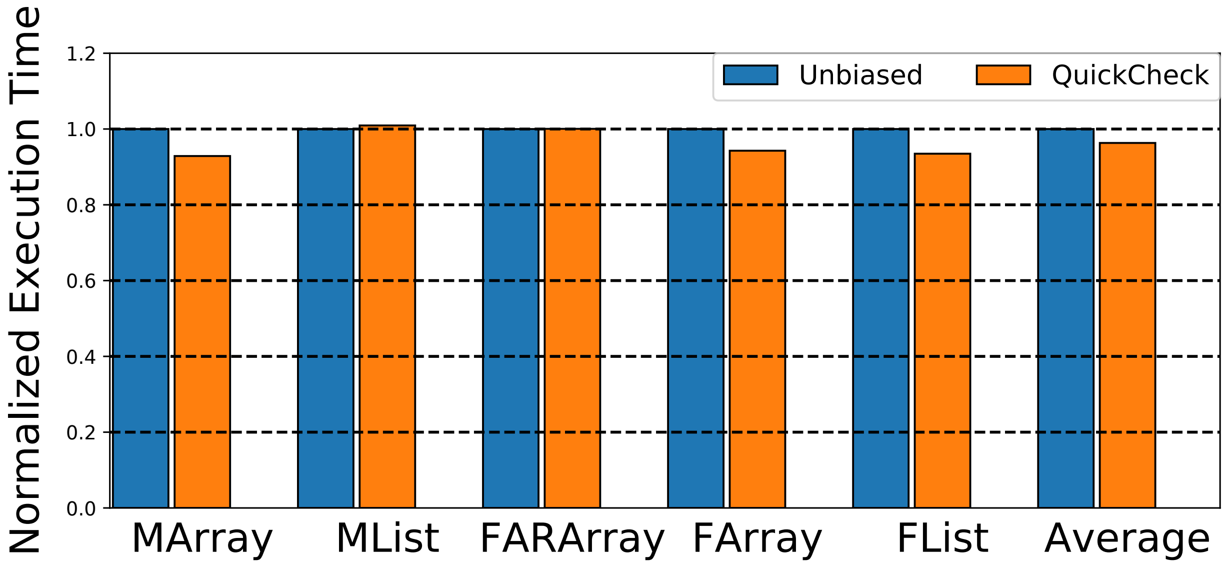


Figure 6.6: Persistent kernel execution time.

Kernels. Figure 6.6 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, the results show that *QuickCheck* is beneficial not only when running applications that do not require data persistence, but also for persistent programs.

Persistence Check Access Patterns. Table 6.5 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 categorized as Unbiased. Also, similar to Table 5.1, 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

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 6.5: Dynamic persistence check statistics while running YCSB and kernel workloads.

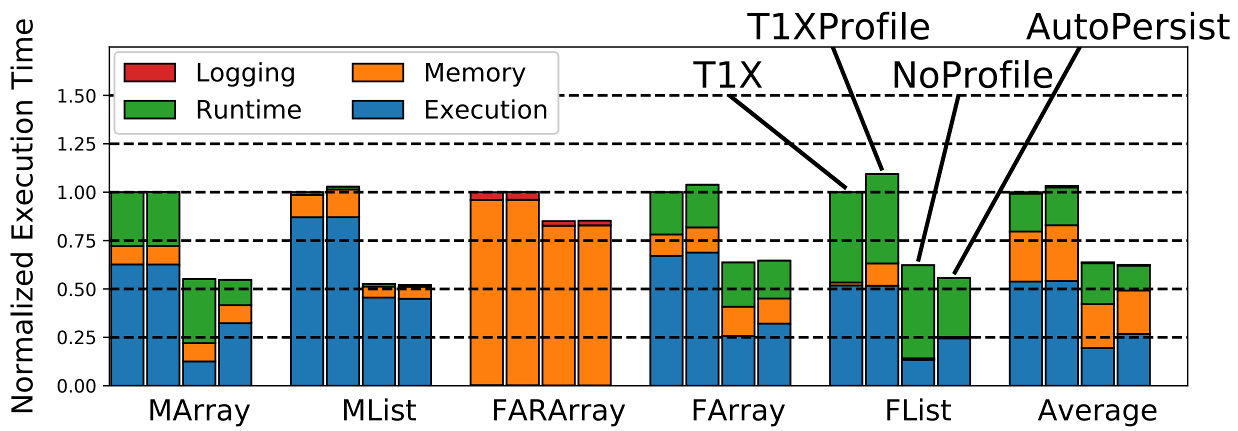


Figure 6.7: Kernel execution time with AutoPersist configs.

is because these checks are in methods which are compiled by the optimizing compiler before their `activated` and `bypassed` 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.

6.8.3 Breakdown of Persistent Kernels

To highlight the benefits of optimizations within AutoPersist, Figure 6.7 compares the execution time of the kernels in the different AutoPersist frameworks: *T1X*, *T1XProfile*, *NoProfile*, and

AutoPersist (Table 6.3). The bars are normalized to *T1X* and are broken down into the usual categories.

As shown in Figure 6.7, 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. In addition, *T1XProfile* takes only a bit longer to execute than *T1X*, which shows that the overhead of adding profiling into the baseline compiler is minimal.

Comparing *NoProfile* and *AutoPersist* shows the performance impact of the profiling pass. By eagerly allocating in NVM objects anticipated to become persistent, my pass reduces the *Runtime* by an average of 39%. However, the total execution time decreases by an average of only 2%. Nevertheless, I 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, I believe that the profiling optimization will become more important.

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

Table 6.6: *NoProfile* and *AutoPersist* event counts.

Runtime Events. To further understand the behavior of AutoPersist, the *NoProfile* and *AutoPersist* frameworks are profiled while running each kernel (Table 6.6). 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 the profiling optimization in AutoPersist. Specifically, Column 4 shows the number of objects that are eagerly allocated in NVM. The optimization allocates a large fraction of the objects eagerly. Columns 5 and 6 show the data corresponding to Columns 2 and 3. My 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.

It can be shown that the number of allocation sites in the source code that are profiled by the 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, I believe that identifying such sites manually would be hard.

Memory Overheads. The changes proposed in AutoPersist introduce new memory overheads due to the introduction of the *NVM_Metadata* header word to each object. The memory overhead of the larger header increases the memory consumption of the key-value store by an average of 9.4%. Fortunately, this overhead is tolerable due to the large memory capacity that NVM can provide.

6.9 SUMMARY

Overall, I observe that AutoPersist performs favorably against other existing techniques for using NVM in Java, such as IntelKV and Espresso. In particular, I find that, on average, AutoPersist is over twice as fast as IntelKV and also substantially outperforms Espresso. In addition, through using profile-guided optimizations, the runtime overheads within AutoPersist are minimized. Such results demonstrate that it is possible to have a NVM application environment which is both easy-to-use and also achieves high performance.

Chapter 7: ISA Support for Instruction-Level Execution Dependencies

7.1 INTRODUCTION

Manipulating persistent data structures requires the insertion of fences into the code to ensure writes propagate to NVM in a specific order. Ensuring all previous writes and cacheline writebacks have completed requires the insertion of an SFENCE [6] for x86-64 systems or a DSB [7] for AArch64 systems. Such fences are commonly needed within persistent applications to preserve the order in which writes propagate to NVM. For instance, while performing undo logging, the undo log entry must be persisted before the original element can be updated.

In undo logging, the update has an *execution dependency* on its corresponding log entry. An execution dependency means that for correctness, the execution dependence's source operation must complete before the dependence's sink can make observable memory changes. For performance, however, different log updates should be able to proceed in parallel. Unfortunately, in current ISAs, these execution dependencies are not able to be conveyed between independent (i.e. no register or memory dependency) instructions. Instead, programmers have to use fences, which enforce an execution ordering between *all instructions* and serialize independent log updates.

Likewise, in AArch64 often multi-threaded volatile applications have fine-grain store orderings which currently must be enforced via fences. For instance, Java's memory model [69] requires select fields to be finalized before an object can be passed to another thread. Unfortunately, in AArch64 this requires a fence to wait for all stores to complete.

To reduce the need for fences, this chapter proposes an ISA extension to add new instructions capable of describing execution dependencies. I call the new instructions the *Execution Dependency Extension (EDE)*, and have added it as an extension to Arm's AArch64 instruction set architecture.

EDE creates a new addressing mode (EDE addressing mode) and adds this addressing mode to store and cache maintenance instructions. An instruction is an execution dependency *producer* if a subsequent instruction may have an execution dependence with this instruction; similarly, an instruction is an execution dependency *consumer* if it has an execution dependence on a prior instruction.

A new key set, called the *execution dependency key (EDK)* set, is introduced to help link execution dependency producers and consumers. A producer-consumer pair with the same EDK is used to convey an execution dependence; in other words, if an instruction ($inst_A$) which produces a specific EDK number is followed by another instruction ($inst_B$) which consumes the same key, then this means that $inst_B$ can only make observable changes once $inst_A$ is complete. Notice that this linking of execution dependencies via EDKs is very similar to how data dependencies are linked through registers.

Via EDE, one is able to precisely convey instruction-level execution orderings without fences, thus allowing all other instructions to proceed out-of-order. Now, given the execution orderings conveyed by EDE, it is the responsibility of the hardware to honor them. In this chapter, I propose two implementations, called *IQ* and *WB*.

In IQ, EDE's execution dependencies are enforced at the issue queue. This implementation monitors execution dependencies alongside registers and memory dependencies to help decide when instructions can be issued.

While IQ achieves significant performance gains, there are still opportunities for further improvement. This is because by enforcing execution dependencies at the issue queue, all subsequent writes within the program are also delayed. To remedy this, I also propose a more aggressive design which enforces execution orderings of store and cache maintenance instructions at the write buffer; I call this design WB.

To evaluate the performance impact of EDE, I implement IQ and WB in gem5 [91] while modeling an Arm A72 with hybrid DRAM+NVM memory. For evaluation applications, I both develop kernels and port Persistent Memory Development Kit (PMDK) [8] applications to use execution dependencies, thereby minimizing the need for fences. Overall, EDE achieves on average 18% and 26% speedups across IQ and WB, respectively.

7.2 UNDERSTANDING FENCE OVERHEAD

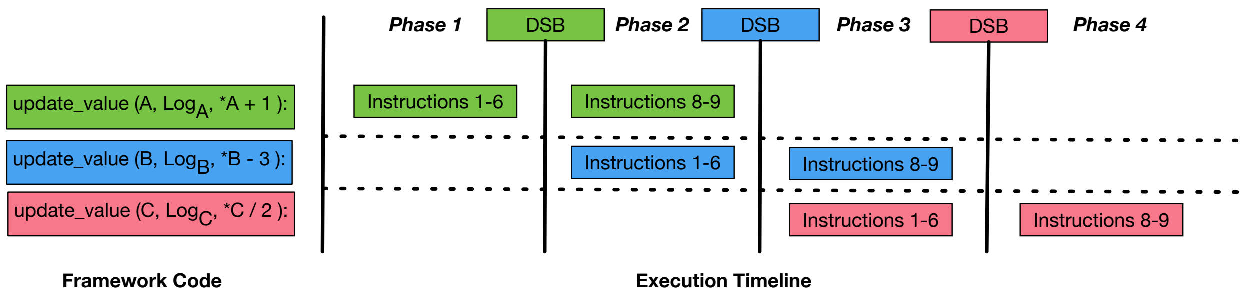


Figure 7.1: Reordering limitations imposed by DSBs

Currently, to guarantee a writeback to NVM is ordered relative to other instructions, a fence must been inserted within the code to prevent subsequent instructions from executing until the writeback is complete. To better understand how this negatively affects performance, let us consider how fences limit the amount of instruction reordering the processor can perform. As an example, I will explain how a persistent application must implement undo logging with fences.

```
  update_value:
1 MOV X1, Addr        ; move value address into register
2 MOV X2, Addr_Log    ; move log address into register
3 MOV X3, Value_New   ; load new value into register
4 LDR X4, [X1]        ; load original value into register
5 STP X1, X4, [X2]    ; log address & original value
6 DC CVAP [X2]        ; persist log entry
7 DSB                 ; wait for log entry to persist
8 STR X3, [X1]        ; store new value
9 DC CVAP [X1]        ; persist new value
```

Figure 7.2: AArch64 undo logging

Figure 7.2 shows the AArch64 instructions needed to implement an `update_value` function. The goal of `update_value` is to use undo logging to update a value in a crash-consistent manner. This involves first persistently storing the location's original value and address into a log and then updating the value. This function takes as parameters the value's address (Addr), the address of the log entry to use (Addr_{Log}), and the new value (Value_{New}).

After collecting all of the needed information into registers (lines 1-4), on line 5 this function stores both the original value and its address into the log via AArch64's *pairwise-store (STP)* instruction. It is necessary to store both the value's address in addition to the value itself in the log for proper recovery.

On line 6, a `DC CVAP` is issued to persist the log entry. Since STP is 16-byte aligned, both stored values will be on the same cacheline; hence, only one `DC CVAP` is necessary to persist both values. Also, since line 6 accesses the same address as line 5, no fence is needed to ensure the stores in line 5 execute before the `DC CVAP`. This is because the processor must honor intra-thread memory dependencies.

On lines 8 and 9, `update_value` writes the new value and ensures it is made persistent. However, to ensure the logging of lines 5-6 has completed persistently before the value itself is updated by line 8, a `DSB` is inserted on line 7. As explained in Chapter 2.1, in AArch64 `DSB`s must be inserted to ensure all prior persistent actions have completed before subsequent instructions execute. Hence, the `DSB` on line 7 ensures lines 5-6 complete before lines 8-9.

On AArch64, `DSB`s impose an ordering on *all* instructions. This can have a significant performance impact when unrelated code follows the `DSB`. Figure 7.1 shows the impact `DSB`s have when three consecutive calls to `update_value` are made. The left-hand part of the figure shows three calls to `update_value`. These three calls update different addresses (A, B, and C) and also reserve different log entry values (Log_A, Log_B, and Log_C). To help clarify these are three unrelated calls to update_value, I have also colored these calls differently (green, blue, and red).

Since these three calls are independent, ideally the processor will execute instructions from each of these three calls in parallel. Furthermore, the DC CVAPs from line 9 do not need to complete until the end of the failure-atomic region. However, due to the presence of DSBs, this is not possible.

The right portion of Figure 7.1 shows the ordering restrictions placed on the instructions by the DSBs. This Figure shows a timeline of when each of the functions' instructions can execute relative to the DSBs. To help make this clear, in the figure I label the execution phases created by the DSBs. During phase 1, only instructions from the green (first) update_value routine can execute. This is because the DSB within the routine prevents any instructions from the subsequent two update_value calls from being executed. In Phase 2, both instructions 8-9 from the green routine and instructions 1-6 from the blue (second) routine can execute concurrently. However, instructions from the red (third) routine are still blocked by the blue routine's DSB. Furthermore, even though they are unrelated, the blue routine's instructions 8-9 are waiting on the green routine's instructions 8-9 before they can complete. As can be seen in the figure, in total four phases are necessary to complete these three calls to update_value.

Ideally, since each of the calls is independent, it should only take two phases to execute these instructions. Instructions 1-6 of each routine should execute in parallel while instructions 8-9 of each routine should wait for their corresponding persistence dependence. However, because a DSB is needed to enforce each persistence dependence, these function calls are unnecessarily serialized.

Note that code such as Figure 7.1 is commonplace for failure-atomic regions when frameworks are used to manage NVM crash consistency for two reasons. First, fences are represented as inline assembly within the framework code, which means that the compiler has no information about fences' intention and hence is unable to move operations around them.

Second, traditionally NVM frameworks are built independently from the persistent applications and are then dynamically linked into the users' programs. This means that when the application's machine code is generated, the compiler is unable to inline and move around the framework's code to minimize the impact of DSBs. Hence, even if the compiler had a better understanding of the fence's intention, it would still be unable to perform substantial optimizations.

7.3 MAIN IDEA

7.3.1 Problem – Unable to Convey Instruction Execution Dependencies

As shown in Chapter 7.2, many fences are needed when performing undo logging in persistent applications. This issue, however, is the symptom of a larger problem: currently, it is not possible

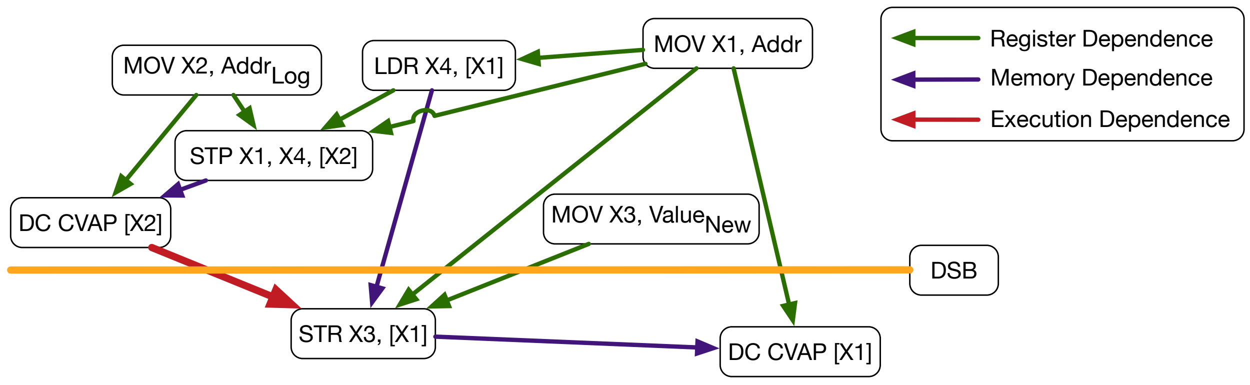


Figure 7.3: Dependency graph for the instructions in function `update_value`

to convey arbitrary instruction dependencies within the ISA.

Within traditional applications, register and memory dependencies can be conveyed by the ISA. Register dependencies are communicated by two instructions referring to the same register while memory dependencies are conveyed by two instructions accessing the same address. However, in persistent applications, there are now logical ordering dependencies which are not expressed via a memory location or a register.

Figure 7.3 shows the same instructions as in Figure 7.2, only now with the register (green arrows) and memory dependencies (purple arrows) within the code labeled. Within the figure, register dependencies go from instructions which define the register to instructions which use the same register. Memory dependencies, on the other hand, chain together all accesses of a given address. This is because, even while using relaxed memory models, memory accesses within a thread must honor program order.

This collection of register and memory dependencies imposes a set of scheduling restrictions on out-of-order (OoO) processors. In order to keep the functional units full and throughput high, processors try to execute instructions as early as possible. However, processors still must respect all register and memory dependencies within the execution.

In the example, in cases where a `DC CVAP` must follow a `STP` and `STR`, such as in lines 5→6 and 8→9 of Figure 7.3, the memory dependency between the sets instructions ensures they are not sent to memory out of order. However, as explained before, one also needs to ensure the `DC CVAP` of line 6 executes before the `STR` of line 8. To convey this in Figure 7.3, I introduce the notion of an *execution dependency*. In the figure the execution dependence between `DC CVAP [X2]` and `STR X3, [X1]` is represented by a red arrow.

An execution dependence denotes a required ordering of the specified stores and cacheline writebacks to memory. An execution dependency means that for correctness the execution dependence's source operation must complete before the dependence's sink can make observable memory changes. Note that when an instruction "completes" is defined as when the operation is observable to memory; this is different than when it is retired/committed. Chapter 7.4.2 describes in more detail the notion of completion.

Unfortunately, since `DC CVAP [X2]` does not produce a register used by `STR X3, [X1]`, nor do these instructions access the same memory address, the processor does not naturally respect this execution dependence. Therefore, it is necessary to insert a DSB in the code. Within Figure 7.3, the DSB is represented by a yellow line. Unlike register and memory dependencies, the DSB enforces an ordering across *all* instructions. Unfortunately, presently this is the only option one has to ensure the execution dependency is honored by OoO processors.

7.3.2 Solution – Encode Execution Dependencies within Instructions

Given the above limitations, in this chapter I propose to add new instructions which convey execution dependencies. Execution dependencies explicitly enforce the ordering of the specified stores and cacheline writebacks to memory. Collectively, I call my new instructions the *Execution Dependency Extension (EDE)*.

In the new instructions, in addition to the traditional memory and register dependencies, an execution dependency on an arbitrary prior instruction can also be defined. I enable this by introducing several new concepts: the execution dependency key (EDK) set, EDK producing instructions, and EDK consuming instructions. While the particulars of these new features are described in Chapter 7.4, the main idea is that EDE allows for instructions to be linked so that, given an execution dependence between two instructions, the sink instruction (EDK consumer) cannot make observable memory changes until the source instruction (EDK producer) is complete.

The benefit of EDE is that, by explicitly describing execution dependencies between instructions at the ISA level, the number of fences needed within both persistent and multi-threaded applications is substantially reduced. For instance, in the example described in Chapter 7.2, by using EDE it is possible to convey the required execution dependence between `DC CVAP [X2]` $\Rightarrow$ `STR X3 [X1]` without the need for a `DSB SYS` in between the two instructions.

By reducing the number of fences, the processor is able to achieve greater performance by allowing more instructions to be executed in parallel. While this chapter mainly focuses on the need to convey arbitrary execution dependencies within the ISA as a problem for persistence applications, EDE also has many uses cases within volatile multi-threaded applications. Chapter 7.8.1 briefly covers other use cases for EDE.

7.3.3 Hardware Support for EDE

Via EDE, it is now possible to convey instruction execution dependencies to the underlying hardware. However, it is the hardware's responsibility to use this provided information to maximize an application's performance. While many hardware implementations of EDE are possible, here I choose to evaluate two practical options. The first implementation, called IQ, enforces execution dependencies at the issue queue. In this implementation, the execution of an EDK consuming instruction is delayed until the corresponding EDK producing instruction has completed.

While IQ is effective, it is possible to achieve even further performance benefits. Since stores and cacheline writeback instructions do not complete (i.e. make observable changes) until after they are committed, in IQ it is possible for a prior producer-consumer pair to prevent the execution of subsequent store and cacheline writeback instructions.

To prevent the above stalling from happening, I also propose a more aggressive design which enforces the ordering of store and cachline writeback instructions at the write buffer; I call this design WB. In WB, store and cacheline writeback instructions are allowed to commit before their execution dependencies are satisfied. However, at the stage where their changes are pushed to memory (i.e. the write buffer), the execution dependencies are enforced. This prevents an instruction's commit from being delayed, hence allowing subsequent independent instructions to proceed unencumbered.

7.4 EDE ISA DEFINITION

This section describes EDE's specification. First, I describe the high-level concepts needed to define execution dependencies. Afterwards, I introduce the new instructions EDE provides.

7.4.1 EDE Concepts

In order to allow execution dependencies to be conveyed by the ISA, new high-level concepts must be introduced. In particular, new abstractions must be provided for defining a dependency source, a dependency sink, and linking together the source and sink. In EDE, I call the dependency source the *dependency producer*, the dependency sink the *dependency consumer*, and link them together by *execution dependency keys (EDK)s*.

Execution Dependency Keys. Normally, intra-thread dependencies are conveyed by registers or memory addresses. However, for arbitrary execution dependencies, a new convention for linking together instructions must be established. In EDE, execution dependencies are conveyed via *execution dependence keys (EDK)s*.

EDKs provide a way to link two instructions together. Like traditional registers, EDKs are directly encoded into instructions; however, unlike registers, no data is stored or loaded. Instead, EDKs are used to index an *execution dependence map (EDM)*. The EDM holds (EDK $\rightarrow$ instruction) key-value pairs. At the decode stage, an instruction's EDKs are used to interact with the EDM and determine any execution dependencies. First, based on the instruction's EDKs, the EDM is searched to check if the instruction is the sink of any execution dependencies. Afterwards, the EDM is updated to reflect any EDKs the instruction redefines. Chapter 7.4.3 shows examples of how EDKs can be used to establish execution dependencies.

Currently, EDE defines *sixteen* EDKs (EDK #0 – EDK #15). Throughout the rest of the chapter I refer to each EDK operand as `EDK #`, where # refers to the key being accessed. The EDM map itself only has to hold fifteen, not sixteen entries. This is because `EDK #0` serves as a *zero key*. When the zero key is encoded into an instruction, it means that this field is not being used and can be ignored. This is needed when a given instruction is either not an execution dependency source and/or not a sink. An example use case of the zero key can also be found in Chapter 7.4.3.

Dependency Producers. EDE introduces the concept of *dependency producers* to convey that an instruction is a source of an execution dependence. A dependency producer is an instruction to which one or more subsequent dependency consumer instructions may be linked.

Dependency producing instructions provide an EDK which is used to access the EDM. When a dependency producer instruction is decoded, the EDM is updated to store the new (EDK $\rightarrow$ instruction) link in the appropriate slot. In this way, subsequent dependency consumer instructions using the same EDK will be able to query this EDM entry and be linked to the appropriate in-flight instruction.

Dependency Consumers. As a complement to dependency producers, in EDE there also needs to be a way to convey that an instruction is the sink of an execution dependence. EDE defines a *dependency consumer* to be an instruction which is dependent on one or more prior dependency producers.

As with dependency producers, dependency consuming instructions also provide EDKs which are used to access the EDM. However, for dependency consumers, if an entry is found for the provided EDK within the EDM, then it means that this dependency consuming instruction must wait for the EDM entry's instruction to finish before it can execute itself. Note that multiple dependency consuming instructions can depend on the same dependency producing instruction. This is because each dependency consumer is merely querying the EDM, but is not modifying it in any way.

7.4.2 New Instructions

Based on the concepts defined in Chapter 7.4.1, it is now possible to define the instructions introduced in EDE. The following subsections describe the new addressing mode introduced by EDE, which memory instructions implement EDE's addressing mode, and the additional control instructions that EDE introduces to ensure correctness.

EDE's New Addressing Mode. Traditionally in ISAs, memory instructions implement multiple addressing modes. The different addressing modes use various combinations of register and immediate operands to access memory. EDE introduces an additional addressing mode for instructions with execution dependencies. I call the new addressing mode the *execution dependency addressing mode*. The new addressing mode has the following format: (EDK$_{def}$,EDK$_{use}$) <REG$_{val}$> [REG], where EDK$_{def}$ is the instruction's dependency producer key, EDK$_{use}$ is the instruction's dependency consumer key, optional <REG$_{val}$> is the value to be stored, and REG is the memory location to access.

When using this addressing mode it is possible for an instruction to be both a dependency consumer and producer. However, it is also possible for an instruction to not create a consumer or producer dependency by inserting the zero key (EDK #0) in the appropriate operand field. In this format, it is only possible for an instruction to be dependent on one previous instruction. However, as explained Chapter 7.4.2, this limitation is removed by using other control instructions introduced by EDE.

Memory Instructions. While the execution dependency addressing mode can be added to many instructions, currently EDE is only applied to AArch64's store and cacheline writeback (`DC CVAP`) instructions.

Beyond using the new addressing mode, a key component for execution dependencies is to define when a dependency producing instruction has completed. This must be defined to establish when it is safe for a dependency consumer to be sent to memory. Note that when an instruction "completes" is different than when it is retired/committed. This is because, in order to preserve precise interrupts, stores and `DC CVAP`s are sent to the memory subsystem after their retirement. The following paragraphs define when stores and `DC CVAP` are considered complete.

Stores. For store (`STR(h,b)`) and pairwise store (`STP`), completion is defined terms of the operation's visibility to other observers. EDE considers a store operation complete once the stored value(s) may be visible to other observers.

DC CVAP. AArch64 defines `DC CVAP` as being complete only once the desired address's data is guaranteed to be persistent. EDE uses the same definition for its completion.

Note that the completion point is dependent on the underlying system. For instance, in a system with Asynchronous DRAM Refresh (ADR) [30], a `DC CVAP` is considered complete once the instruction reaches the NVM's memory controller.

Other Instructions. In this chapter I only discuss adding the execution dependency addressing mode to Arm's store and cacheline writeback instructions. However when EDE is used in other domains it would be used to add this addressing mode to other instructions, such as loads and synchronization primitives, as well. Chapter 7.8.1 discusses how EDE could be used in other domains.

Control Instructions. In addition to the instructions leveraging the new execution dependency addressing mode described above, EDE also introduces three new instructions to handle more exotic control flows: *JOIN (EDK$_{def}$, EDK$_{use_1}$, EDK$_{use_2}$)*, *WAIT_KEY (EDK)*, and *WAIT_ALL_KEYS*.

JOIN (EDK$_{def}$, EDK$_{use_1}$, EDK$_{use_2}$). This instruction can wait on up to two prior dependency producers and is "completed" once both of its dependencies have completed. Via `JOIN`, it is possible for a instruction to have execution dependencies with multiple prior dependency producers. `JOIN` is also useful for EDK resolution when multiple control paths merge. For instance, if an instruction is waiting on `EDK #1` from one control flow or `EDK #2` from another, it is possible to insert a `JOIN (EDK #1, EDK #2, EDK #0)` in the latter branch to resolve the key discrepancy.

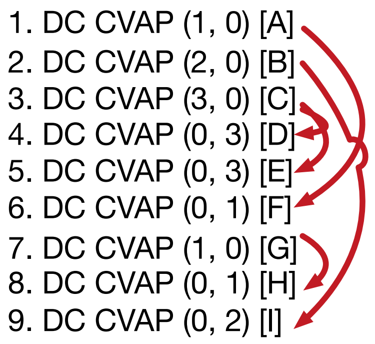
WAIT_KEY (EDK) and **WAIT_ALL_KEYS**. In the presence of function calls, without intervention it is possible for a callee function to overwrite an EDK key in use by the caller function and cause incorrect execution dependencies to be linked. To prevent this from happening, EDE introduces the `WAIT_KEY (EDK)` instruction. The `WAIT_KEY (EDK)` instruction is both a dependency producer and consumer of the same key. However, unlike other instructions, `WAIT_KEY (EDK)` is only considered complete once all prior dependency consumers of the matching key have finished. Therefore, this instruction can be used after a function call to ensure all necessary dependencies are met. Details about how to define a calling convention for EDE are in Chapter 7.8.3.

I have also introduced a new instruction, `WAIT_ALL_KEYS`, which prevents *all* subsequent instructions from executing until *all* prior dependency producers and consumers complete. This instruction can be used when compiler dependency analysis fails, such as in the presence of function calls with unknown destinations and/or unknown side effects. In addition, this instruction can be useful at the end of large transactions to ensure all persistency operations have finished.

7.4.3 EDE Example

Figure 7.4 shows how EDKs are used to define the execution dependencies among a series of instructions. In the figure, execution dependencies between instructions are shown by red arrows.

Overall, there are execution dependencies between lines 1→6, 2→9, 3→(4,5), and 7→8. Notice how by using different EDKs it is possible to define execution dependencies between multiple instructions concurrently. In addition, as shown in lines 1→6 & 7→8, EDKs can be redefined to establish new execution dependencies.



```
1. DC CVAP (1, 0) [A]
2. DC CVAP (2, 0) [B]
3. DC CVAP (3, 0) [C]
4. DC CVAP (0, 3) [D]
5. DC CVAP (0, 3) [E]
6. DC CVAP (0, 1) [F]
7. DC CVAP (1, 0) [G]
8. DC CVAP (0, 1) [H]
9. DC CVAP (0, 2) [I]
```

Figure 7.4: EDK Examples

```
  update_value:
1 MOV X1, Addr
2 MOV X2, Addr_Log
3 MOV X3, Value_New
4 LDR X4, [X1]
5 STP X1, X4, [X2]
6 DC CVAP (1,0) [X2]
7 DSB
8 STR (0,1) X3, [X1]
9 DC CVAP [X1]
```

Figure 7.5: EDE undo logging

Figure 7.5 shows how to apply EDE to the `update_value` routine previously described in Figure 7.2. Remember that previously on line 7 a DSB was necessary to ensure line 6 completed before line 8 executed. Now, instead, the instructions on lines 6 and 8 use EDE's execution dependency addressing mode to convey this dependence. In Figure 7.5, line 6 is modified to record that this instruction is a dependency producer of `EDK #1` and consumes the zero key. Similarly, line 8 is modified to be a dependency consumer of `EDK #1` and to produce the zero key. By doing so, the DSB on line 7 is no longer necessary and can be removed.

7.5 HARDWARE IMPLEMENTATION

In this Chapter I describe how EDE is implemented in hardware.

7.5.1 Mapping Producer-Consumer Pairs

As described in Chapter 7.4.1, the execution dependence map (EDM) is a fifteen-entry map which holds (EDK $\rightarrow$ instruction) key-value pairs. In EDE's implementation, the EDM stores the in-flight instruction tags of dependency producers. At the decode stage, while the register file is being accessed, the EDM is accessed to match all of the instruction's EDKs. If the consumer EDK is not found in the EDM, then the instruction does not have an execution dependence. However, if the EDK is present in the EDM, then the instruction is registered to have an execution dependency on the corresponding in-flight instruction. Likewise, at the decode stage, if the decoded instruction has a producer EDK, then the proper EDM slot is updated to store the instruction's tag.

Once the instruction has completed from EDE's perspective, it is necessary to remove this matching entry from the EDM. Therefore, upon completion of a dependency producing instruction, the corresponding EDK slot is queried within the EDM. If the queried instruction tag matches the completed instruction's tag, then the entry is removed.

Maintaining EDM Consistency. In Out-of-Order processors, sometimes squashes are needed to flush speculative state out of the pipeline. In this situation, the EDM must also be reverted to a non-speculative state. To accomplish this, two copies of the EDM are maintained: one at the current non-speculative state ($EDM_{no-spec}$) and another at the current speculative state (EDM_{spec}). Note that register files also commonly maintain two mappings [92]. Throughout normal execution, the EDM$_{spec}$ is used by the front-end. However, on a speculation squash, EDM$_{no-spec}$ is copied into EDM$_{spec}$ before execution restarts.

To maintain EDM consistency across context switches, in EDE requires that all of the in-flight dependency producers complete before performing the context switch. However, since Arm machines already must insert a `DSB` into context switch handlers, this is already done. Hence, EDE does not need to provide instructions to allow for the EDM state to be saved and restored.

7.5.2 Determining Instruction Completion

In EDE it is crucial to detect when an instruction completes. As explained in Chapter 7.4.2, when an instruction completes is dependent on the type of instruction.

EDE considers store operations complete once the stored value(s) may be visible to other observers. In the implementations, stores become visible to other observers once they are pushed from the write buffer onto the memory subsystem.

For `DC CVAP`s, EDE only considers them complete once the desired address's data is guaranteed to be persistent. In the implementations, the memory system provides an acknowledgement when the operation has reached the persistent domain.

7.5.3 Implementing EDE's Memory Instructions

Once execution dependencies are identified, the processor's back-end must ensure all dependencies are upheld. While many strategies are possible, this chapter proposes two solutions: *IQ*, which enforces the execution dependencies at the issue queue, and *WB*, which waits until the write buffer to enforce these dependencies. Each design is described below.

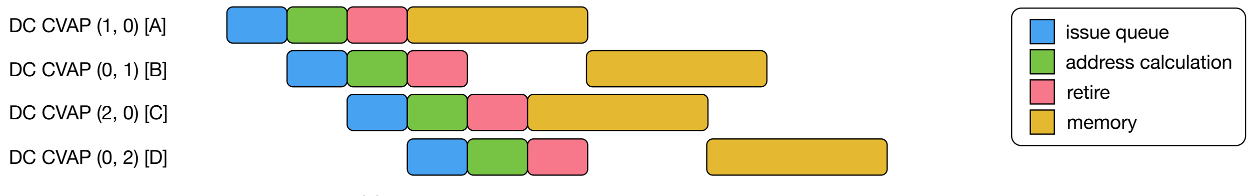
Enforcing Dependencies within Issue Queue. In IQ, all required execution orderings are enforced at the issue queue. This is done by adding an additional stage to each instruction's wakeup logic. Normally, an instruction is deemed ready-to-execute once all of its register and memory dependencies have been met. IQ now adds additional state to also monitor the status of each instructions' execution dependencies. In particular, I add the *execution dependencies ready (eDepReady)* flag to each instruction within the issue queue. Now, only once an instruction's execution dependencies are satisfied (in addition to all other pre-existing dependencies) is the instruction marked as ready-to-execute.

When an instruction enters the issue queue, IQ checks to see if it has any outstanding execution dependencies. If so, then the `eDepReady` flag is unset. Otherwise, since the instruction is not waiting on any execution dependencies, then the `eDepReady` is set.

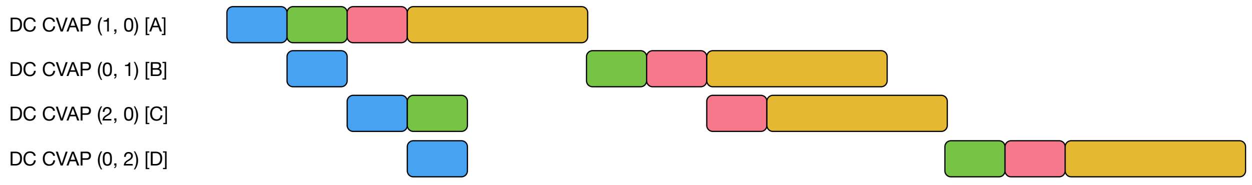
When an instruction has completed, IQ sets the `eDepReady` flag of all matching dependency consuming instructions within the issue queue and checks if they can be marked as ready-to-execute. Once an instruction is marked as ready-to-execute, the existing issue queue scheduling logic proceeds as before unmodified.

Drawbacks of IQ. While IQ faithfully implements EDE, performance is left on the table because store and cache maintenance operations do not enter the memory subsystem until after they are committed. Hence, this means that, in IQ, prior execution dependency pairs can impact the performance of subsequent memory operations.

To better understand this limitation, consider Figures 7.6(a)&(b). Each figure shows a different execution timeline for a set of four `DC CVAP` instructions to different addresses with two sets of



(a) Ideal enforcement of execution dependencies



(b) Best-case enforcement of execution dependencies with *IQ* implementation

Figure 7.6: Comparison between ideal and IQ execution timelines

execution dependency pairs. In each timeline, each instruction is broken down into its *issue queue*, *address calculation*, *retire*, and *memory* phases. Also, in the timelines, there are gaps between the phases when the instructions are stalled for various reasons described below. Note that in the figure the length of each phase is not to scale. For instance, the figure portrays the memory phase as being three times as long as the other phases; however, in practice the length of the memory phase can be substantially longer.

Figure 7.6(a) shows an ideal timeline for when each of the instructions should execute their phases while also ensuring all execution dependencies between the instructions are upheld. Note that to preserve precise interrupts, instructions must be both issued and retired in order. Since there is an execution dependency between `DC CVAP (1, 0) [A]` and `DC CVAP (0, 1) [B]`, `DC CVAP (0, 1) [B]` must wait to perform its memory phase until `DC CVAP (1, 0) [A]` has completed.

Unlike `DC CVAP (0, 1) [B]`, `DC CVAP (2, 0) [C]` does not have an execution dependence on `DC CVAP (1, 0) [A]`, and hence can proceed to begin its memory phase immediately after it commits. However, since `DC CVAP (0, 2) [D]` is dependent on `DC CVAP (2, 0) [C]`, it must wait to perform its memory phase until `DC CVAP (2, 0) [C]` has completed.

Figure 7.6(b) shows the timeline for when IQ is used to enforce the execution dependencies. In this example, the execution takes significantly longer than the ideal (figure 7.6(a)). This is because even though `DC CVAP (2, 0) [C]` is not dependent on either `DC CVAP (1, 0) [A]` or `DC CVAP (0, 1) [B]`, `DC CVAP (2, 0) [C]` cannot retire until `DC CVAP (0, 1) [B]` has retired. However, since IQ enforces execution dependency ordering at the issue queue, `DC CVAP (0, 1) [B]` it unable to proceed to the address calculation phase until `DC CVAP (1, 0) [A]`'s memory phase is complete. Hence, IQ is unable to allow the ideal amount of parallelism between these instructions.

While the code example provided in Figure 7.6(a) is very simple, this pattern of pairwise instruction dependencies is common within NVM applications. As described in Chapter 7.2, a primary motivating factor for EDE is to allow multiple log updates to proceed in parallel. Unfortunately, in IQ, the performance benefits when log updates are in close succession will be diminished by having execution dependencies enforced at the issue queue.

Enforcing Dependencies within Write Buffer. To resolve the performance limitations of IQ, a second implementation is also introduced where execution dependencies are resolved within the write buffer. I call this implementation WB. WB allows instruction execution to continue as normal until retirement. At this point, WB controls in which order instructions in the write buffer are allowed to be sent to the memory subsystem.

In WB, instruction execution dependencies are kept track of within the write buffer. Now, instructions can only interact with memory once their execution dependencies are satisfied. When a dependency producing instruction is completed, then all of the linked dependency consuming instructions within the write buffer are updated to reflect that their execution dependencies have been satisfied.

Overall, the design of WB is very similar to IQ, except now execution dependencies are being enforced at the write buffer rather than the issue queue. However, by allowing instructions to commit without stalling for execution dependencies, WB has noticeable performance benefits over IQ.

7.5.4 Implementing EDE's Control Instructions

Recall that EDE introduces three new control instructions: `JOIN` $(EDK_{def}, EDK_{use_1}, EDK_{use_2})$, `WAIT_KEY (EDK)`, and `WAIT_ALL_KEYS`. Each of these instructions waits for potentially multiple prior EDE instructions to complete. The following paragraphs describe how each of these instructions are implemented.

JOIN $(\mathbf{EDK}_{def}, \mathbf{EDK}_{use_1}, \mathbf{EDK}_{use_2})$. As described in Chapter 7.4.2, the purpose of this instruction is twofold: both to allow a subsequent instruction to wait on multiple dependencies and also to rename EDKs.

Both IQ and WB implement this instruction by holding in-flight `JOIN` instructions information in a side buffer. This buffer is called the *join queue* and `JOIN` instructions are inserted into it after they are decoded. As dependency producers are completed, the join queue is notified; once a `JOIN` instruction is completed, the issue queue (in IQ) or write buffer (in WB) is notified.

WAIT_KEY (EDK) and **WAIT_ALL_KEYS**. As described in Chapter 7.4.2, these instructions monitor the status of many prior instructions before completing. To keep track of the information needed to decide when these instructions complete, I introduce a new set of counters to track the number of per-EDK in-flight dependency producers, and also a new counter to keep track to the total number of in-flight EDE instructions. These counters are updated as instructions commit and complete. Note that these counters are incremented at the commit stage, not the decode stage. This is to ensure the count accurately represents the number of dependency producers and consumers which precedes the control instruction. Once either `WAIT_KEY` or `WAIT_ALL_KEYS` reaches the head of the reorder buffer, it checks the appropriate counter and is retired once the counter reaches zero.

7.6 EXPERIMENTAL SETUP

7.6.1 Simulator Environment

To evaluate EDE and my proposed hardware implementations, I implement both IQ and WB within the gem5 simulator [91]. I have added EDE's instructions both into gem5's AArch64 frontend and as built-ins within Clang+LLVM [93] version 8.0.

Simulator - ISA	gem5 - AArch64 + EDE
Compiler	Clang-LLVM 8.0 + EDE builtins
Processor Parameters	
Processor	OoO core, 3-instr decode width, 3GHz
Ld-St queue	16 entries each
L1 I-cache	32KB, 2-way, 2-cycle access latency
L1 D-cache	48KB, 3-way, 1-cycle access latency
L2 cache	256KB, 16-way, 12-cycle data
L3 cache	1MB/core, 16-way, 20-cycle data
Main-Memory Parameters	
Capacity	DRAM: 2GB; NVM: 2GB
NVM latency	150ns read ; 500ns write
NVM region size	256B
NVM Write Buffer	128 slots
DRAM Type;	2400MHz DDR4
DRAM Ranks per Channel	2
DRAM Banks per Rank	16

Table 7.1: Simulation architectural parameters.

Within gem5, I use an out-of-order configuration meant to model Arm's A72 processor [94]. The main simulator architectural parameters are shown in Table 7.1. I have also modified gem5 to model a hybrid DRAM+NVM memory system. In my setup, both NVM and DRAM requests are sent to one controller. However, the physical address space is split so that part of the address space targets NVM while the other part targets DRAM. The DRAM interface models 2400MHz DDR4 while the NVM interface has a 128 slot on-DIMM write buffer and includes asymmetric read and write latencies.

7.6.2 Applications

To evaluate the performance impact of EDE, I use a combination of kernel applications as well as benchmarks available within the PMDK [8] repository. The applications used are listed in Table 7.2. As kernel applications, I have created two benchmarks which perform a series of modifications to an array. In *update*, an operation consists of updating a random index within the array while in *swap*, an operation is to swap the values of two random elements within the array; in both benchmarks, undo logging is used to maintain crash consistency.

Kernel Applications.	
update	perform updates on random elements within an array.
swap	perform pairwise swaps between random array elements.
PMDK Applications	
btree	B-tree implementation with between 3 and 7 keys per node.
ctree	Crit-bit trie [95] implementation.
rbtree	Red-black tree implementation with sentinel nodes.
rtree	Radix tree implementation with radix 256.

Table 7.2: Evaluation applications

I also have modified a collection of PMDK applications to use EDE. Specifically, within its repo, PMDK provides the *pmembench* benchmark suite which can be used to evaluate the performance of several persistent data structures. To evaluate EDE, I have modified the PMDK API to leverage EDE while performing undo logging and have updated the *tree_map* data structures to use the new API. As shown in Table 7.2, the evaluation tests four data structures: *btree*, *ctree*, *rbtree*, and *rtree*. In each benchmark, a single operation consists of inserting a new element into the data structure.

Across all benchmarks, multiple operations are grouped into a transaction. Specifically, in the simulations, I set the applications to have 100 operations per transaction and to run 1,000 transactions, resulting in each application performing 100,000 operations. In the simulations, I run each application to completion while precisely simulating the time spent performing these operations.

7.6.3 Configurations

Configuration & Description
Baseline (*B*): Use DSBs to enforce crash-consistent ordering.
EDE + IQ (*IQ*): Use EDE and target IQ hardware.
EDE + WB (*WB*): Use EDE and target WB hardware.
Unsafe (*U*): Do not use DSBs. Allows unsafe reorderings.

Table 7.3: Benchmark configurations.

Throughout the evaluation, I compare four different configurations: *B*, *IQ*, *WB*, and *U*. Table 7.3 describes each configuration in detail. *B* uses DSBs as needed to ensure a correct crash-consistent ordering in AArch64. Both *IQ* and *WB* use EDE instead of DSBs to properly order each log entry persist with their following update. However, *IQ* uses the IQ hardware support while *WB* uses the proposed WB design. Finally, in *U* all DSBs from the execution are removed. Note that in *U* the hardware may perform reorderings which violate crash-consistency requirements. This configuration is shown only to provide more perspective about EDE's performance gains.

7.7 EVALUATION

7.7.1 Execution Time

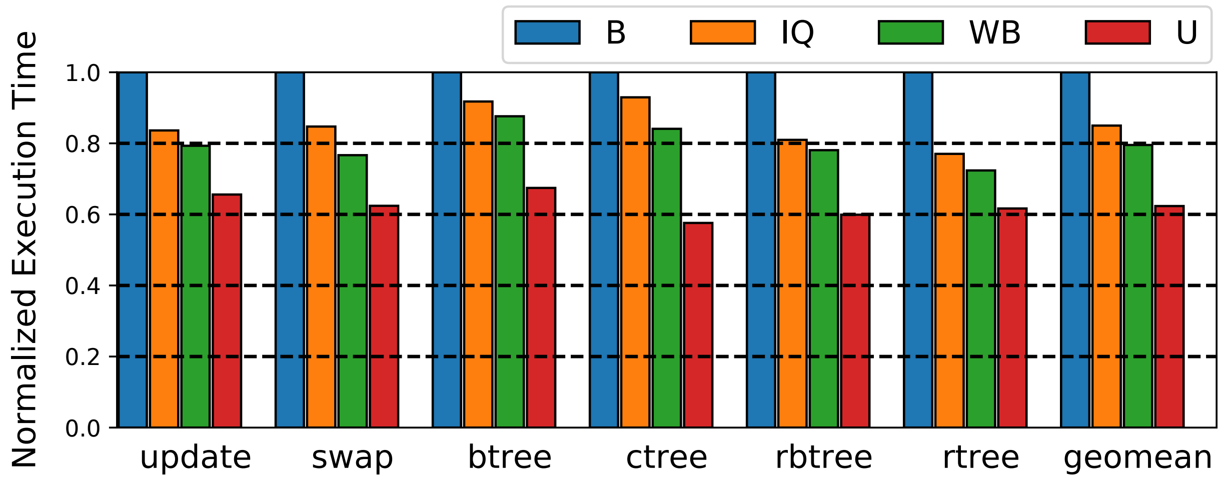


Figure 7.7: Application execution time.

Figure 7.7 shows the execution time for the applications and configurations described in Chapter 7.6. In the figure all execution times are normalized to *B*. Based on their geometric mean, *IQ*, *WB*, and *U* reduce the execution time by 15%, 20%, and 38%, respectively.

Most benchmarks attain similar amounts of improvement. Across all benchmarks, *IQ* performs better than *B*. This is because EDE allows for much lighter fences than the DSBs needed in *B*. Likewise, *WB* performs better than *IQ* across all benchmarks, on average by 5%. This is because, as discussed in Chapter 7.5.3, since IQ enforces execution dependencies before the commit phase, it is unable to maximize the amount of parallelism described within EDE. Since WB enforces execution dependencies within the write buffer, it is able to allow more writes to proceed out of order.

On average, *WB* is able to attain 54% of the execution time reduction of *U* (20% vs. 38%). This is recovering a significant portion of the time spent ensuring NVM is updated in a crash-consistent order. Remember that since *U* removes all fences from the code, it allows for reorderings which could prevent data recovery. Overall, the results show that EDE is able to significantly reduce the overhead of maintaining a crash-consistent ordering.

7.7.2 Issue Queue Throughput

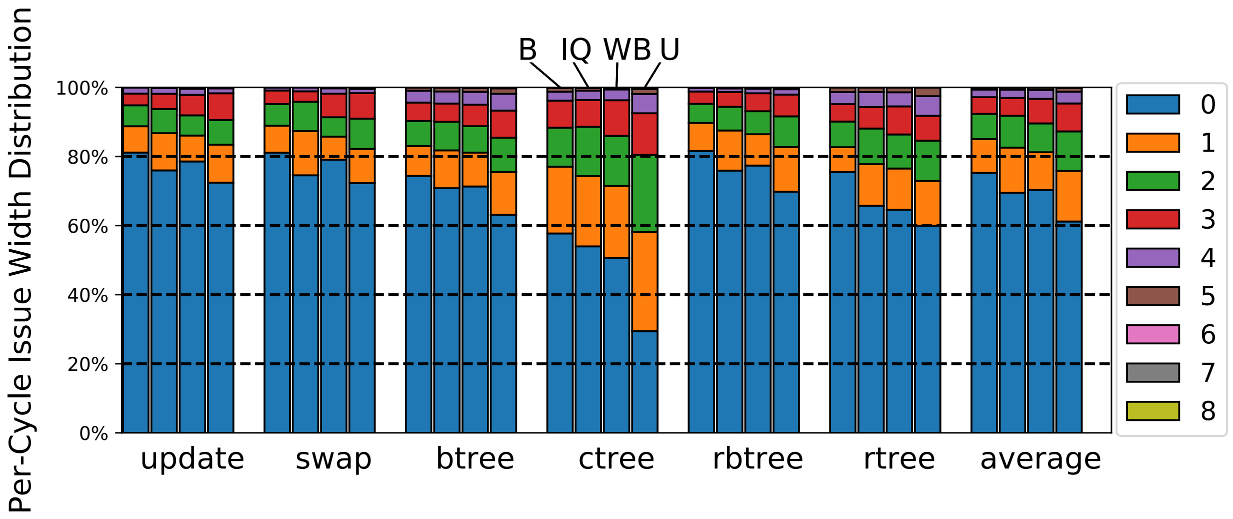


Figure 7.8: Distribution of # instructions issued each cycle.

Figure 7.8 shows the distribution of the number of instructions issued each cycle. In the figure, each color represents the percentage of cycles the processor issued the labeled number instructions. As shown on the y-axis, cumulatively the bars for each setup stack up to account for 100% of the simulated cycles. Note that although the simulated architecture has a 3-instruction decode width, the issue queue has a width of 8. This configuration has been developed by Arm to most accurately model its A72 core.

As shown in the figure, all implementations issue 0 instructions in the majority of cycles. This is to be expected, as writes to NVM have a significant latency and can cause the pipeline to fill. On average, the IPC is 0.40, 0.46, 0.49, and 0.64 for the *B*, *IQ* *WB*, and *U* configurations, respectively.

Across all benchmarks, *IQ* and *WB* spend fewer cycles being unable to issue instructions than *B*. *IQ* and *WB* each spend 5% more cycles issuing instruction than *B*. While *IQ* and *WB* both spend on average 30% of cycles issuing instructions, *WB* in general issues more instructions during these active cycles. On average, when issuing instructions, *WB* is able to issue 8% more instructions than *IQ*. This is because *WB* does not perform any blocking at the issue queue while *IQ* forces instructions to remain there until execution dependencies are satisfied.

7.7.3 Pending Writes to NVM

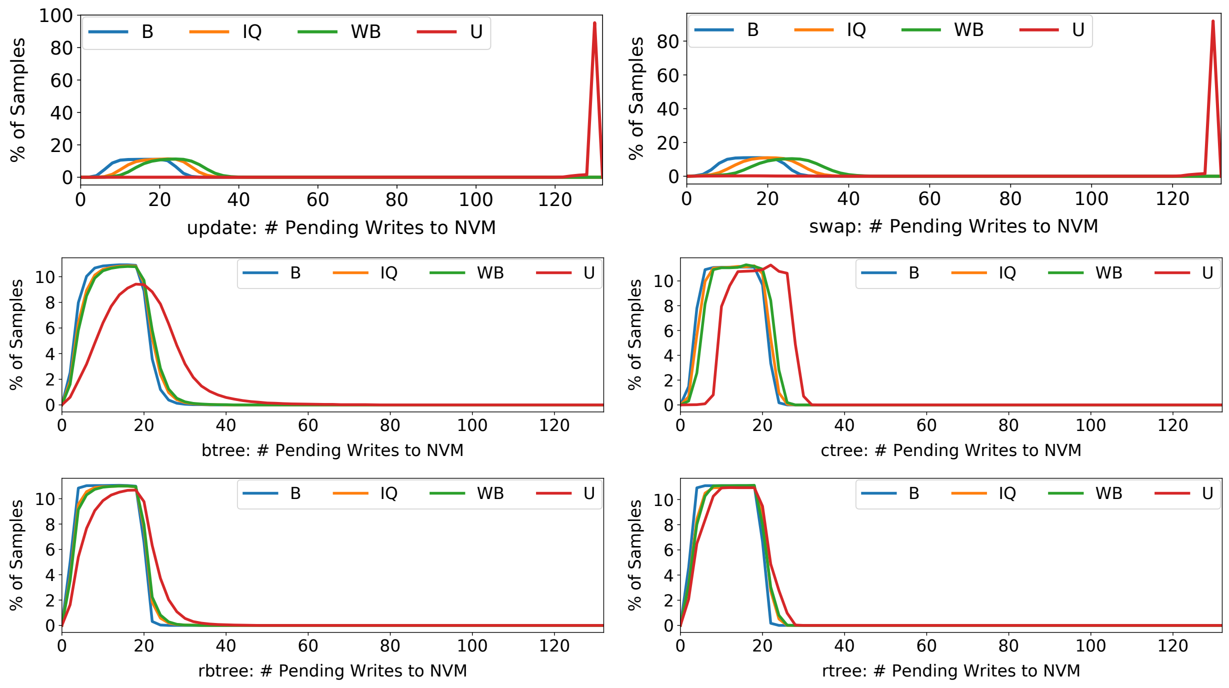


Figure 7.9: Distribution of Pending Writes to NVM.

Figure 7.9 shows for each benchmark the distribution of the number of pending writes to NVM. Each configuration is represented by a different colored line. The x-axis shows the number of pending writes to NVM while the y-axis shows is the percentage of samples with the observed number of pending NVM writes. A sample is taken each time a store reaches the NVM media.

As described in Chapter 7.6, pending NVM writes are held in a 128 slot on-DIMM write buffer. In general, it is better to have a large number of pending writes; having the buffer full provides more coalescing opportunities and also enables the memory controller to generate optimal write-back schedules. However, because fences stall the execution, it can be hard for crash-consistent applications to issue writes quick enough to keep the buffer full.

Across all benchmarks, *U* has the most number of pending NVM writes. This is because *U* does not issue fences which stall writes from being issued. For the kernel benchmarks, *U* is able to keep the memory controller full since it writes to NVM at a high frequency. In the PMDK benchmarks, since other work must be done to maintain the data structures, the number of pending NVM writes is lower.

For the other configurations, *WB* on average has more pending writes than *IQ* to NVM, while both EDE implementations have more pending writes than *B*. As discussed in Chapter 7.5.3, since *WB* allows writebacks to NVM quicker than *IQ*, this behavior is to be expected.

7.8 DISCUSSION

7.8.1 Other uses for EDE

In this chaper I focus on how EDE improves the performance of undo logging in persistent applications. However, beyond NVM applications, EDE also would have a significant impact in existing applications where fine grain ordering is needed. For example, EDE can be used when multi-threaded coordination is needed, such as for the initialization of singleton data, when following a publisher pattern, or for inserting data into concurrent data structures. In these situations, one must ensure the data is properly initialized before any flags are set to indicate the data is available. Via EDE, this can be done without a fence.

Such multi-threaded coordination of published data is commonplace within code. For instance, the Java Memory Model [69] requires `final` fields to be initialized before they are read by another thread. In addition, traditionally Java Virtual Machine (JVM) implementations store metadata alongside object fields which also must be initialized before another thread can access the object. This coordination of small data subsets is an ideal target for EDE.

Another example where multi-threaded coordination of published data is needed is the kernel's use of circular buffers. Often kernels use circular buffers to store tracing and logging data collected through the runtime. Ideally, the buffers should handle being read and updated by multiple threads concurrently. Via EDE, popping and pushing data from circular buffers can be performed in a lock-free manner without the use of fences.

Finally, concurrent garbage collectors, such as ZGC [87] and Shenandoah [96] require careful coordination of the movement of data alongside concurrent updates and dynamic code loading. In this environment, EDE can be used to minimize the overheads of garbage collection barriers.

Note that the use cases described above require the EDE addressing mode to also be added to loads. It is straightforward to enforce execution dependencies on loads in the issue queue by using the same technique as done within the IQ implementation.

7.8.2 Compiler Support for EDE

As described in Chapter 7.6, this chapter leverages EDE through the use of new built-in intrinsics added to Clang+LLVM. However, it is desirable to more fully integrate EDE into the compiler. Specifically, I believe the compilers' internal representation (IR) can easily be modified to incorporate execution dependencies. For instance, in Java Virtual Machine (JVM) compilers, a sea-of-nodes [97] representation is commonly used. Like Figure 7.3, this representation creates a dependency graph to record data, memory, and control dependencies between instructions. It is straightforward to also introduce execution dependencies into the representation.

In addition, I believe that compilers and frameworks, not application developers, should define execution dependencies. For the use cases described in Chapter 7.8.1, it is possible for a compiler to automatically create execution dependencies during its initial IR generation. Similar automatic support would be possible for persistent applications once persistency support is integrated into languages. It is also possible to expose EDE to framework developers via new ordering types for C/C++ atomics [98] and Java VarHandles [99].

Finally, by fully integrating EDE into compilers, it possible for EDKs to be virtualized and for the compiler to automatically assign logical EDK values. Existing allocation techniques such as graph coloring [100] and linear scan [101] are straightforward to repurpose for EDK assignments.

7.8.3 EDK Calling Convention

An important consideration for any ISA is its calling convention. Establishing a standardized calling convention allows for code from multiple sources to be used together. Normally, registers not used to pass parameters are divided into two categories: *caller-saved* and *callee-saved* registers. In a similar manner, for EDKs the concept of caller-saved and callee-saved *keys* is introduced.

For caller-saved keys, the caller must assume that the key will be overwritten within the called function. Hence, for caller-saved keys, after the function returns and before the next dependency consumer instruction, a `WAIT_KEY (EDK)` instruction must be inserted.

For callee-saved keys, the caller function performs no action. However, within the called function, either (a) a `WAIT_KEY (EDK)` is inserted at the end of the called function or (b) all instructions which overwrite the key must also be a dependency consumer of the same key value. Note that option (b) causes the dependencies to be chained together, so that subsequent dependency consumers must wait for the entire chain to complete before they can execute.

7.8.4 Applying EDE to Other ISAs

In this thesis, I have proposed to add EDE as an extension to Arm's AArch64 architecture. However, it is also possible to add this extension to other ISAs such as x86-64, POWER, and RISC-V. The key difference for each architecture would be to determine when to use EDE to add execution dependencies not already enforced by the given ISA's memory model.

Note that in x86-64, cacheline writebacks (CLWBs) are an unordered operation. Hence, EDE can be used to create an execution dependence between a CLWB and a store to an unrelated address. Furthermore, on x86-64, currently a memory fence is needed to ensure a load does not bypass a prior store; this is a problem for garbage collection. EDE is able to enforce this dependency without a fence as well.

7.8.5 EDE Security Implications

Currently, to prevent spectre and meltdown variants on existing Arm hardware, the recommended mitigation is to insert a `DSB` and instruction synchronization barrier (`ISB`) in tandem [102]. Instructions using EDE still follow AArch64's barriers, so existing spectre mitigations will also be effective for EDE.

Chapter 8: Conclusion

The emergence of byte-addressable non-volatile memory devices promises to radically alter how applications will persist data in the future. However, before such technology can truly become popular, two main issues have to be solved. First, NVM frameworks must be introduced that enable programmers to easily create persistent applications while still having high performance. Second, current gaps in hardware support that have a significant performance impact on persistent applications must be corrected.

In this thesis, I provided possible solutions to these issues. First, I presented the design and implementation of AutoPersist, a new programmer-friendly NVM framework in Java. AutoPersist simplifies creating persistent applications by allowing existing built-in libraries to be used and requiring minimal annotations from the programmer. Furthermore, through its JVM integration and use of profile-guided optimizations, AutoPersist is able to provide a programmer-friendly interface while still attaining high performance.

Second, I also proposed a new ISA extension, named Execution Dependency Extension (EDE), which allows for arbitrary instruction-level execution dependencies to be conveyed and honored by the underlying hardware. Through EDE, coarse fences are no longer needed for persistent applications, thereby substantially improving performance.

Chapter 9: References

[1] 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, 2018.

[2] 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, “Phase-change random access memory: A scalable technology,” *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, July 2008.

[3] H. Akinaga and H. Shima, “Resistive random access memory (reram) based on metal oxides,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[4] J. Xu and S. Swanson, “NOVA: A log-structured file system for hybrid volatile/non-volatile main memories,” in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu pp. 323–338.

[5] “Java Flight Recorder Runtime Guide.” [Online]. Available: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170

[6] “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, 2015.

[7] Arm, “Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile,” https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf, 2020.

[8] “Persistent Memory Development Kit.” [Online]. Available: http://pmem.io/pmdk/

[9] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379 pp. 91–104.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950380 pp. 105–118.

[11] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "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*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173201 pp. 70–83.

[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589 pp. 133–146.

[13] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660224 pp. 433–452.

[14] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064204 pp. 468–482.

[15] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037730 pp. 135–148.

[16] J. E. Denny, S. Lee, and J. S. Vetter, "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*, ser. HPDC '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907303 pp. 125–136.

[17] N. Cohen, D. T. Aksun, and J. R. Larus, "Object-oriented recovery for non-volatile memory," *Proceedings of the ACM on Programming Languages,*, vol. Vol. 2, no. OOPSLA, pp. 153:1–153:22, 2018.

[18] L. Marmol, M. Chowdhury, and R. Rangaswami, "Libpm: Simplifying application usage of persistent memory," *ACM Trans. Storage*, vol. 14, no. 4, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3278141

[19] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for non-volatile main memory is hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3124680.3124729 pp. 13:1–13:8.

[20] T. Shull, J. Huang, and J. Torrellas, "Defining a high-level programming model for emerging nvram technologies," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ser. ManLang '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3237009.3237027 pp. 11:1–11:7.

[21] T. Shull, J. Huang, and J. Torrellas, "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*, ser. PLDI'19, June 2019.

[22] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, java," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 30:1–30:24, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400689

[23] T. Shull, J. Huang, and J. Torrellas, "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*, ser. VEE'19, April 2019.

[24] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, Jan 2010.

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

[26] "Persistent Memory Documentation." [Online]. Available: https://docs.pmem.io/persistent-memory/

[27] Arm, "Armv8-A architecture evolution,"
https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/
armv8-a-architecture-evolution, 2016.

[28] "SNIA NVDIMM Messaging and FAQ." [Online]. Available: https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf

[29] A. Raad, J. Wickerson, and V. Vafeiadis, "Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360561

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

[31] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[32] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC 17. USA: USENIX Association, 2017, p. 349362.

[33] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC 18. USA: USENIX Association, 2018, p. 9931005.

[34] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "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*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915251 pp. 371–386.

[35] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. Berkeley, CA, USA: USENIX Association, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2750482.2750495 pp. 167–181.

[36] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 439–451.

[37] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proceedings of the ACM on Programming Languages (PACMPL)*, 2017. [Online]. Available: http://infoscience.epfl.ch/record/231400

[38] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304046 p. 441454.

[39] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, p. 865876, June 2014. [Online]. Available: https://doi.org/10.14778/2732951.2732960

[40] E. Giles, K. Doshi, and P. Varman, "Hardware transactional persistent memory," 2018.

[41] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540744 pp. 421–432.

[42] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2830772.2830802 p. 672685.

[43] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 336–349.

[44] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 361–372.

[45] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124539 pp. 178–190.

[46] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA 17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080240 p. 175186.

[47] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712 pp. 265–276.

[48] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *Distributed Computing*, C. Gavoille and D. Ilcinkas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327.

[49] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080229 pp. 481–493.

[50] H.-J. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2926697.2926704 pp. 55–67.

[51] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct 2014, pp. 216–223.

[52] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ser. ISCA 2020. ACM, 2020.

[53] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 77–89.

[54] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[55] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 507–519.

[56] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[57] U. Hölzle, C. Chambers, and D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," in *Proc. of PLDI*, July 1992, pp. 32–43.

[58] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the Smalltalk-80 system," in *Proc. of POPL*, 1984.

[59] D. Grove, J. Dean, C. Garrett, and C. Chambers, "Profile-guided receiver class prediction," in *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/217838.217848 pp. 108–123.

[60] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer, "Memento mori: Dynamic allocation-site-based optimizations," in *Proceedings of the 2015 International Symposium on Memory Management*, ser. ISMM '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2754169.2754181 pp. 105–117.

[61] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes," in *Conference on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '89. New York, NY, USA: ACM, 1989. [Online]. Available: http://doi.acm.org/10.1145/74877.74884 pp. 49–70.

[62] G. Ottoni, "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*, ser. PLDI 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192374 pp. 151–165.

[63] T. Shull, J. Choi, M. J. Garzaran, and J. Torrellas, "Nomap: Speeding-up javascript using hardware transactional memory," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 412–425.

[64] C. Wimmer and H. Mössenböck, “Automatic object colocation based on read barriers,” in *Modular Programming Languages*, D. E. Lightfoot and C. Szyperski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 326–345.

[65] L. Stadler, T. Würthinger, and H. Mössenböck, “Partial escape analysis and scalar replacement for java,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2544137.2544157 p. 165174.

[66] V. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, “Persistent memory transactions,” 2018.

[67] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872381 pp. 399–411.

[68] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, “Persistency for synchronization-free regions,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192367 pp. 46–61.

[69] J. Manson, W. Pugh, and S. V. Adve, “The java memory model,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040336 pp. 378–391.

[70] S. Blackburn and J. N. Zigman, “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*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=648123.747394 pp. 250–258.

[71] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, “On-the-fly garbage collection: An exercise in cooperation,” *Commun. ACM*, vol. 21, no. 11, pp. 966–975, Nov. 1978. [Online]. Available: http://doi.acm.org/10.1145/359642.359655

[72] M. Paleczny, C. Vick, and C. Click, “The java hotspottm server compiler,” in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM01. USA: USENIX Association, 2001, p. 1.

[73] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, “Design of the java hotspot client compiler for java 6,” *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 7:1–7:32, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1369396.1370017

[74] O. Community, "Graal Project," http://openjdk.java.net/projects/graal/.

[75] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2509578.2509581 p. 187204.

[76] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: Application initialization at build time," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360610

[77] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167488 pp. 169–190.

[78] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder, "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*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 657–676.

[79] "QuickCached,"
https://github.com/QuickServerLab/QuickCached.

[80] "Memcached: A distributed memory object caching system." [Online]. Available: https://memcached.org/

[81] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152 pp. 143–154.

[82] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 96. New York, NY, USA: Association for Computing Machinery, 1996. [Online]. Available: https://doi.org/10.1145/237721.237727 p. 3241.

[83] L. O. Andersen and P. Lee, "Program analysis and specialization for the c programming language," Ph.D. dissertation, 2005.

[84] B. Blanchet, "Escape analysis for object-oriented languages: Application to java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 99. New York, NY, USA: Association for Computing Machinery, 1999. [Online]. Available: https://doi.org/10.1145/320384.320387 p. 2034.

[85] V. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, "Persistent memory transactions," 2018.

[86] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019.

[87] "ZGC - Open JDK Wiki." [Online]. Available: https://wiki.openjdk.java.net/display/zgc/Main

[88] J. Eisl, M. Grimmer, D. Simon, T. Würthinger, and H. Mössenböck, "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*, ser. PPPJ '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2972206.2972211 pp. 14:1–14:11.

[89] "Pmemkv: Key/Value Datastore for Persistent Memory." [Online]. Available: https://github.com/pmem/pmemkv

[90] "PCollections." [Online]. Available: https://pcollections.org/

[91] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 17, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[92] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1496–1514, 1987.

[93] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO 04. USA: IEEE Computer Society, 2004, p. 75.

[94] Arm, "Arm Cortex-A72 MPCore Processor: Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.100095_0003_06_en/cortex_a72_mpcore_trm_100095_0003_06_en.pdf, 2020.

[95] D. R. Morrison, "Patriciapractical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, p. 514534, Oct. 1968. [Online]. Available: https://doi.org/10.1145/321479.321481

[96] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "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*, ser. PPPJ '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2972206.2972210 pp. 13:1–13:9.

[97] C. Click and M. Paleczny, "A simple graph-based intermediate representation," in *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, ser. IR 95. New York, NY, USA: Association for Computing Machinery, 1995. [Online]. Available: https://doi.org/10.1145/202529.202534 p. 3549.

[98] "C++ Atomic Types and Operations." [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html

[99] "JEP 193: Variable Handles." [Online]. Available: https://openjdk.java.net/jeps/193

[100] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47 – 57, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0096055181900485

[101] C. Wimmer, "Linear Scan Register Allocation for the Java HotSpot Client Compiler," M.S. thesis, Johannes Kepler University Linz, 2004.

[102] "Arm Whitepaper: Cache Speculation Side-channels." [Online]. Available: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper