Thèse n° 7187

Software Support for Non-Volatile Memory (NVM) Programming

Présentée le 15 juillet 2021

Faculté informatique et communications
Laboratoire d'informatique à très grande échelle
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

David Teksen AKSUN

Acceptée sur proposition du jury

Prof. A. Argyraki, présidente du jury
Prof. J. R. Larus, directeur de thèse
Prof. M. Scott, rapporteur
Prof. M. Seltzer, rapporteuse
Prof. E. Bugnion, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

We are at the very beginning of time for the human race.
It is not unreasonable that we grapple with problems.
But there are tens of thousands of years in the future.
Our responsibility is to:
do what we can,
learn what we can,
improve the solutions, and
pass them on.
— Richard P. Feynman

To my family…

Acknowledgements

Looking back at the previous years, I can easily say that my journey was very similar to a voyager during the times of the Age of Exploration. I was searching for new routes, the journey was long, and the future was uncertain. Despite all the difficulties and all the challenges, this journey would not have been possible without the right captain to guide the ship.

I am very lucky to have Prof. James Larus, who was a magnificent thesis advisor, as the captain of the ship. I am very honored to be chosen by Jim, a world-renowned researcher and professor. I am eternally grateful to Jim for giving me this opportunity and allow me to do research with him. He has allowed me to explore and work on many interesting ideas and research projects. He has supported me with his deep and broad knowledge, and patiently guided me throughout difficult times. His weekly constant feedback and his supportive attitude were irreplaceable. I admire Jim's thinking, writing style, and presentation skills and I will always see him as a great role model. I adore Jim's inquisitive approach to challenging topics. He can break apart very complex subjects, even in unrelated fields, and always finds the right question to ask. His fatherly attitude towards his students and his expectation of excellence are what make him so unique. I enjoyed TA'ing his courses, as I saw his diligence and perfection in a different context. I appreciate all the time he has spent with me and I wouldn't have this journey any other way.

Prof. Edouard Bugnion has a very special place for me. Ed was very supportive and helpful throughout this journey and I thank him a lot for this. I learned a lot from Ed. I will always remember his comments about each presentation in our lab meetings, his simple way of approaching problems, and his methodology of finding the right abstraction to analyze complex systems. To this day, I still use the design principles that I learned from Ed's course for building systems.

I would like to thank Prof. Edouard Bugnion, Prof. Margo Seltzer, Prof. Michael L. Scott, and Prof. Katerina Argyraki for being in my thesis committee and providing valuable feedback.

I am very grateful to Nachshon Cohen, who was my collaborator. He is very smart, thoughtful, and a very great researcher as he can come up with great ideas. He taught me a lot of things, not just about research, but about all aspects of life. I will always consider him as a dear colleague and a true friend.

I would like to thank Tania for being supportive and keeping everything organized. Everything ran smoothly thanks to her.

Maggy was a mom figure and took care of each student. She was very direct and always encouraged every one around her to be a better person. She always kept her smile. Maggy has supported me throughout the most difficult days and I will always remember her support.

Throughout my PhD, I met a lot of smart and hard-working people. However, I cannot tell how much I appreciate my lab mates, Sahand and Mahyar, who always come to my help when in need. Mahyar Emami is a real friend that everybody needs. He has always opened his home to me, has shared everything with me without a question, and I will always remember our research discussions with him. Sahand is a very reliable friend, who is always there for his friends. He was one of my closest friends throughout my PhD and I have a lot of good memories with him.

I would like to thank Mia and Georg for always supporting and helping me. Their insights about research and their feedbacks were very useful. I would like to thank Marios for the research discussions and his feedback. While I met him towards the end of my PhD, I would like to thank Prof. Sanidhya Kashyap for his feedback.

I am very grateful to the rest of my colleagues and friends from VLSC and DCSL at EPFL, who made the experience much better: Stuart, Adrien, Sam, Konstantinos, Jonas, Stanko, George, Dmitrii. I would like to thank my friends and colleagues who made the PhD experience much better and motivated me from EPFL: Sepand, Luis, Siddarth, Atri, Jasmina, Berker, Onur, Tiziano, Kristina, and Jeremie. I would like to thank the people that I know outside EPFL: My childhood friend Doruk, Maggy's son Jimmy, Mendel, and Mushky.

I am very grateful to Madan for providing me with the opportunity to do an internship in MSR. Many thanks to Saeed and Todd for their collaboration.

I would like to extend my thanks to the people at Intel for providing me with access to their servers equipped with Optane, which enabled my research: Mustafa, Sanjay, Arunarasu, Brent, and Aaron.

Finally, I would like to thank my family. My mother, my father, and my grandmother have always supported me since I was born. I will always remember their sacrifices and honor the way in which they have raised me.

Lausanne, June 29, 2021 D. A.

Abstract

Non-Volatile Memory (NVM) is an emerging type of memory device that provides fast, byte-addressable, and high-capacity durable storage. NVM sits on the memory bus and allows durable data structures designs similar to the in-memory equivalent ones. Expensive serialization/deserialization operations, usually associated with block-based storage, are not necessary. Unfortunately, using NVM is not as simple as placing a data structure in NVM and expecting persistence. As NVM sits on the memory bus, processor caches buffer the cache lines from it that are referenced by load and store instructions.

The volatility of the caches and possible power failures at a random point in a program complicate the design and require ways to handle these failures. Prior work focused on implementing crash-consistency mechanisms to correctly recover a program's data after a failure. The goal was to minimize the use of costly cache line write-back and fence instructions, which are necessary for correct durability. Moreover, commercial NVM devices are slower compared to DRAM and affect the design. In addition to performance overheads, correctly implementing crash consistency is hard. The programmers need to carefully reason about cache line write-back instructions and the order of persistent writes. Finding bugs can take from minutes to hours.

In this thesis, we use checkpointing as an effective crash-consistency mechanism with low overhead for building durable data structures. We also describe an inter-procedural dataflow analysis for fast detection of NVM programming bugs. We show the practicality of these ideas through three primary contributions and their implementations.

Firstly, we present *InCLL* to address NVM checkpointing. *InCLL* is a novel technique that uses fine-grained checkpointing and in-cache-line logging to minimize the number of explicit write-back and fence instructions in the fast path of data structure modifications. We evaluate *InCLL* both on DRAM and Optane devices for the Masstree data structure. Secondly, we present a new checkpointing design, *CpNvm*, which minimizes NVM write-back instructions on the critical path of the execution. We achieve low overheads for Masstree and Memcached for write-heavy workloads, and almost no overheads for read-only workloads. In addition, we present *FlowNvm* to find NVM programming bugs. *FlowNvm* can identify NVM programming pattern violations and anti-patterns at compile-time using inter-procedural dataflow analysis.

Keywords: non-volatile memory, NVM, durable data structures, checkpointing, dataflow analysis

Résumé

Non-Volatile Memory (NVM) est un nouveau type de dispositif mémoire qui fournit un stockage durable rapide, adressable par octet, et de grande capacité. La NVM se situe sur le bus mémoire et permet de concevoir des structures de données durables similaires aux structures équivalentes en mémoire. Les opérations coûteuses de sérialisation/désérialisation, généralement associées au stockage par blocs, ne sont pas nécessaires. Malheureusement, l'utilisation de la NVM n'est pas aussi simple que de placer une structure de données dans la NVM et de s'attendre à de la persistance. Comme la NVM se trouve sur le bus mémoire, les caches des processeurs mettent en mémoire tampon les lignes de cache qui sont référencées par les instructions de chargement et de stockage.

La volatilité des caches et les éventuelles pannes de courant à un moment aléatoire d'un programme compliquent la conception et nécessitent des moyens de gérer ces pannes. Les travaux antérieurs se sont concentrés sur la mise en œuvre de mécanismes de cohérence en cas de panne pour récupérer correctement les données d'un programme après une panne. L'objectif était de minimiser l'utilisation d'instructions coûteuses de réécriture et des barrières mémoire qui sont nécessaires pour une durabilité correcte. De plus, les dispositifs NVM commerciaux sont plus lents que la DRAM et affectent la conception. En plus des surcharges de performance, la mise en œuvre correcte de la cohérence en cas de panne est difficile. Les programmeurs doivent raisonner soigneusement sur les écritures de lignes de cache et sur l'ordre des écritures persistantes. Trouver des bugs peut prendre de quelques minutes à quelques heures.

Dans cette thèse, nous utilisons le checkpointing comme un mécanisme de cohérence en cas de panne efficace et à faible surchargeen pour construire des structures de données durables. Nous décrivons également une analyse interprocédurale du flux de données pour la détection rapide des bugs de programmation NVM. Nous montrons l'aspect pratique de ces idées à travers trois contributions principales et leurs implémentations.

Tout d'abord, nous présentons *InCLL* pour traiter le checkpointing NVM. *InCLL* est une nouvelle technique qui utilise le fine-grained checkpointing et le in-cache-line logging pour minimiser le nombre d'instructions de réécriture explicite et de barrières mémoire dans le chemin rapide des modifications de structure de données. Nous évaluons *InCLL* à la fois sur des dispositifs DRAM et Optane pour la structure de données Masstree. Ensuite, nous présentons une nouvelle conception de checkpointing, *CpNvm*, qui minimise les écritures NVM sur le chemin critique de l'exécution. Nous obtenons de faibles surcharges pour Masstree et Memcached pour les charges de travail à forte intensité d'écriture, et presque aucune surcharge pour les charges de travail en lecture seule. De plus, nous présentons *FlowNvm* pour trouver des bugs de programmation NVM. *FlowNvm* peut identifier les violations et les anti-modèles de programmation NVM au moment de la compilation en utilisant l'analyse du flux de données interprocédural.

Mots clefs : non-volatile memory, NVM, structures de données durables, points de contrôle, analyse de flux de données

Contents

List of Figures

List of Tables

1 Introduction

Non-volatile memory (NVM) is an emerging type of memory that offers fast, byte-addressable durable storage. It is possible to store and manipulate pointer-based data structures such as B+ trees and hash maps directly in NVM using load and store instructions. These structures are accessible even after a power failure or a process crash.

One promising use for NVM is constructing robust, high-performance internet and cloud services, which often maintain very large, in-memory data structures and need to quickly recover from faults or failures. In data centers, failures are rare, but they do occur. Barroso [20] reports that, at Google, in a year, a server is restarted between 1.2 – 16 times. Approximately 45% of machines restart at least once in a 6-month period and approximately 5% of machines restart more than once per month.

A challenge in restarting large-scale services is the size of computer memories, which has grown far faster than bandwidth to external storage devices, exacerbating a long-standing bottleneck in persisting and restoring memory state [16]. For example, Facebook [71] reports that restoring 120GB of data for a single machine can require 2.5 – 3 hours due to low disk bandwidth and in-memory format translations. NVM can avoid most of these costs and provides higher bandwidth than disk.

NVM programming, using NVM as a durable memory device, requires crash-consistency. Crash consistency ensures that after a reboot, the program data is in a consistent state and will not lead to any failures. Unlike traditional block-level storage, NVM programming requires careful reasoning about cache lines and ordering persistent writes.

In the rest of the section, we expand on:

1. background information about NVM devices,
2. key challenges and issues associated with NVM programming,
3. our contributions to NVM programming to provide crash consistency using checkpointing and to find programming mistakes using dataflow analysis.

1.1 Non-volatile Memory (NVM)

Non-volatile memory devices promise performance characteristics similar to DRAM and allow byte-addressable access. On top of these features, an NVM device also provides persistence, making it effectively a durable storage media. There are many competing technologies such as PCM [34, 105, 192], STT-RAM [162], ReRAM [14], and CTN [47] for constructing NVM devices.

While NVM devices are still evolving, several products are currently available in the market. These are 3D-XPoint Optane devices from Intel and ReRAM devices from Crossbar. There are also substitute technologies such as battery-backed devices that provide functionalities similar to NVM technologies. We first cover the range of NVDIMMs in the market then continue our discussion with the actual NVM technologies.

In practice, durable memory interfaces come in the form of a non-volatile dual in-line memory module (NVDIMM). NVDIMMs can be attached to the memory bus of a server, which is NVM compatible, to provide durable memory. There are several alternative NVDIMM options to choose from, such as NVDIMM-F, NVDIMM-N, and NVDIMM-P [106, 152].

NVDIMM-F is NAND Flash storage attached to the memory bus. Compared to DRAM, it can have higher capacities. It has the bandwidth of Flash storage and is much slower than DRAM. NVDIMM-F uses block-based storage.

NVDIMM-N is battery-backed DRAM with NAND Flash. The program accesses the DRAM with a byte-addressable interface. If a power failure occurs, the data from DRAM is copied to the Flash storage. Either a battery or a supercapacitor ensures power during the data transfer. The capacity is limited by the DRAM size. In 2019, the market share of NVDIMM-N was 72.5% [152].

NVDIMM-P interface combines DRAM and a non-volatile device such as PCM, STT-RAM, ReRAM, and CTN to provide high capacity main memory. NVDIMM-P provides a byte-addressable interface and can be used as a durable memory device.

NVDIMM-P interface is the recent technology that allows a modern NVM technology to be deployed to run on a server. In this thesis, we consider only NVM devices with an NVDIMM-P interface.

There are several NVM technologies that are commercially available, such as Intel Optane and Crossbar ReRAM. Crossbar Inc. [13] released NVM devices based on ReRAM technology. The specifications state that the ReRAM device has a maximum of $4\mu s$ random read latency and a maximum of $4\mu s$ of write latency without caching. The device supports 10^5 write cycles.

1.1.1 Intel Optane

Intel released their 3D-XPoint NVM device based on PCM technology in cooperation with Micron. The NVM device is commercially available under the brand name of Intel® Optane™ DC Persistent Memory (Intel Optane DCPMM). Optane comes in a variety of capacities ranging from 128GB to 512GB [15]. Optane devices are cost-effective compared to DRAM. Dell's server price list specifies approximately $2700 for 128GB of DRAM and $940 for 128GB of Optane (DRAM is approximately $3x$ costly) [12].

Device	Read	Write
DRAM	106	77
Optane	7	2
Optane-Interleaved	39	14

Table 1.1 – Approximate DRAM and Optane maximum bandwidth (in GB/s) characteristics.

Table 1.1 depicts the maximum bandwidths for the Intel Optane device. Optane is slower than DRAM [188, 91]. Yang [188] reports that NVM's sequential read latency is $2x$ DRAM and its random read latency is $3x$. However, peak NVM read bandwidth is approximately 6% of DRAM and peak write bandwidth is approximately 3% of DRAM. Interleaving can improve NVM bandwidth, but DRAM still has higher bandwidth.

For the rest of the thesis, we focus solely on Optane as an NVM device and ignore the other technologies. There are several reasons for this decision. Servers using Optane are widely available. Optane is exhaustively studied [69, 91, 120, 180, 184, 188] and its performance characteristics are well-known. The reported measurements match the promise of NVM, which is a high-capacity durable memory device with performance nearing DRAM. Many enterprises such as Facebook, Oracle, Huawei, Alibaba, and Tencent have already incorporated Intel Optane into their data centers or are in the process of integration [5, 56].

We will not consider Crossbar ReRAM, as we do not have access to ReRAM based devices and its reported latencies are an order of magnitude worse than DRAM. We will not consider NVDIMM-N as battery-backing introduces its own fault-tolerance problems and complications. Batteries can fail, increase hardware cost, consume space, require cooling, and overall complicate fault-tolerance and system testing [97].

Intel Optane Operation Modes

Optane has different modes of operation [8] in which Optane acts either as volatile memory or as a durable memory device. These modes are *Memory Mode* and *App-Direct Mode*. There is also *Mixed Mode*, which is a combination of these two modes.

Memory Mode uses Optane as the backing store for volatile main memory. DRAM is used as a direct-mapped physically indexed and physically tagged cache for NVM with 4KB granularity [69]. Since NVM offers high-density memory, the program's main memory benefits from Optane's large capacity. DRAM caching can improve performance compared to directly accessing NVM. Unfortunately, the DRAM cache is volatile and the NVM backing store is not kept consistent, so the program data is not accessible after program execution or power failure.

App-Direct Mode provides durable memory to a program. The program maps the NVM device into its virtual address space and accesses memory locations in NVM using load and store instructions. The writes that are flushed from the caches into NVM are durable and available even after a power failure. In this mode, DRAM acts as a normal volatile memory and does not act as a cache.

Mixed Mode allows use of both Memory Mode and App-Direct Mode. A portion of NVM memory can be used for Memory Mode while the remaining portions can be allocated for App-Direct use. This allows some part of NVM to be used as volatile main memory while the rest is durable memory.

For the remainder of the thesis, we focus on App-Direct Mode and consider NVM as a durable memory device. We mainly focus on the performance challenges that arise from introducing crash consistency and checking the correctness of the implementation. We briefly discuss Mixed Mode to extend the capacity of DRAM. We defer a more detailed discussion of NVM programming in App-Direct Mode to section 2.1.

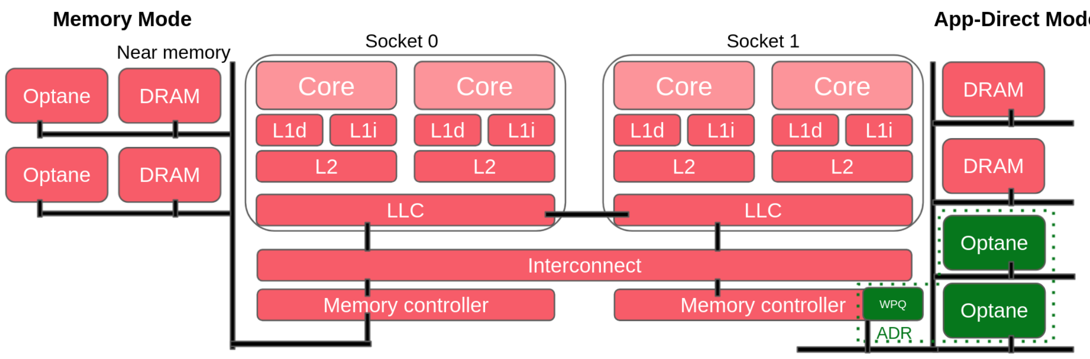


Figure 1.1 – Intel Optane platform.

Figure 1.1 shows an Intel Cascade Lake *x86-64* machine [155, 188] using Optane with different modes. We briefly explain each component in the figure. The cores perform computation and data processing. The program normally uses L1 (data and instruction), L2, LLC (last-layer cache) caches to ensure fast data access. Caching buffers modifications to memory.

Optane uses Intel's proprietary DDR-T interface [188] to communicate with the memory controller. The interface is similar to DDR4 and uses cache-line granularity (64B) for data transfers. It is possible to bypass the processor caches and directly send data to the memory controllers using *x86-64* non-temporal move instructions.

The left side shows one possible configuration, which is the Memory Mode. DRAM acts as a cache to NVM and can buffer the modifications.

The right side of the figure depicts App-Direct Mode. The region within the green boundary indicates the persistence domain and is power-fail safe. Any data within the persistent domain is ensured to be durable by the underlying hardware. The persistence domain includes Optane and Write-Pending Queues (WPQs) of the memory controller. On power failure, Asynchronous DRAM Refresh (ADR) allows WPQs to drain its contents to NVM. The rest of the hardware including CPU registers, caches, and DRAM are volatile and lose data upon power failure. We explain the importance of the ADR assumption to our work in section 2.1.1.

1.2 Principle Problems for Programming NVM

The simple model presented in figure 1.1 depicts the critical issues related to NVM programming: processor caches are volatile. During a power failure, all program data residing in the caches are lost. This power failure can leave the program data in NVM in an unrecoverable state and can lead to runtime errors after a restart [104]. The cache lines can be evicted differently than the program order leading to unexpected states [40]. Furthermore, failure can happen while an operation is ongoing.

```
book* create_book(title, author, ...){
    //allocate and fill the book
    b→ title = title;
    b→ author = author;
    ...
    return b;
}
```

Figure 1.2 – Book example for a library program.

Consider the library example in figure 1.2 to motivate inconsistencies due to power failure. The library program stores a list of books. The code in the figure 1.2 creates a new book, which is later added to the list and made visible to the users. This program is incorrect.

Book Attributes	Title	Author
Initial NVM State	-	-
Cache State	Computer Architecture	Patterson
Possible NVM State 1	-	-
Possible NVM State 2	Computer Architecture	-
Possible NVM State 3	-	Patterson
Possible NVM State 4	Computer Architecture	Patterson

Table 1.2 – Possible NVM states for book example.

Table 1.2 shows the possible states that can occur after a crash. Initially, the `title` and the author fields are not modified and contain old data. Once the create_book function completes, the caches reflect the changes to the book and contain the name as well as the author of the book.

If additional code for durability is not introduced, power failure at the end of the create_book function can leave the book data in NVM in a corrupt state. There are 4 possible NVM states for the book data if we just consider the two fields. The book is not updated in NVM and the old data remains unwritten (State 1). Only the title (State 2) or the author (State 3) fields of the book are updated. The book is updated completely (State 4).

A crash at the end of the create_book function creates uncertainty about the possible NVM states. If the book is added to the list in States 1-3, then users might see incorrect data after a power failure.

```
book* create_book_tx(title, author, ...){
    transactional{
        //allocate and fill the book
        b→ title = title;
        b→ author = author;
        ...
        return b;
    }
}
```

Figure 1.3 – Book example with transactions.

Transactions [76] can be used to provide atomic durability. Figure 1.3 sketches the use of a transaction to avoid bad states.

Book Attributes	Title	Author
Initial NVM State	-	-
Cache State	Computer Architecture	Patterson
Possible NVM State 1	-	-
Possible NVM State 2	Computer Architecture	Patterson

Table 1.3 – Possible NVM states for transactional book example.

Table 1.3 shows the states for execution of the book creation function with transactions. When the code region atomically executes, either the book is created properly or the NVM region is left untouched. There is no partial update of NVM after execution. For safety, the book insertion to the list should also be a single atomic transaction.

1.3 Crash-consistency Challenges

There is no free lunch when it comes to using a crash-consistency mechanism for NVM. There are trade-offs between different crash-consistency mechanisms. They have different runtime performance, space, recovery overhead costs. We leave the detailed discussion of crash-consistency mechanisms to section 2.2.

1.3.1 Cost of Ordering Persistent Writes

To provide atomic durability [155], crash-consistency mechanisms usually rely on cache line flush instructions to force cache line contents back to NVM and fence instructions to ensure that these instructions complete. Otherwise, the program can use non-temporal move instructions with a fence instruction for persistent writes. Executing these instructions for persistent writes is costly and can reduce program performance [118, 177].

1.3.2 Cost of Using Optane

Besides the cost of ordering persistent writes to NVM, Optane has intrinsic costs. Crash consistency designs must take into account the peculiar characteristics of the underlying hardware. Optane has latencies close to DRAM and provides good performance. However, DRAM is still significantly faster than Optane [91, 188]. Moreover, Optane performs worse under heavy concurrent load, mixed-access patterns (mixed read/writes), and random access patterns [91, 188].

1.3.3 NVM Programming Correctness

NVM programming is a challenging endeavor. Programs [126] that use NVM as a durable memory device are mostly written in bug-friendly unmanaged and unsafe programming languages, such as C/C++ [109]. Reasoning about cache line write-back instructions and ordering persistent writes is not conventional in programming and can be challenging.

Using a library for NVM programming remains challenging. The programmer still needs to use the library correctly, for example, finding the right boundaries for a transaction. In addition, libraries such as Intel PMDK offer a complicated programming interface that expects a programmer to use its logging interface to denote persistent memory regions for a transaction.

1.4 Thesis Statement

There should be a diverse set of tools for NVM programming to address the programmer's needs. The goal of this thesis is to expand the software support for NVM programming by providing low-overhead crash-consistency mechanisms through the use of checkpointing techniques and static analysis techniques for discovering NVM programming bugs.

Instead of using transactions, we suggest using checkpointing [1]. Checkpointing avoids eagerly propagating changes to NVM and can persist writes in batches. We show that periodic checkpoints every few milliseconds are achievable with low overheads for NVM programming. In the thesis, we also show that, by carefully exploiting DRAM, the slowdown due to crash-consistency can be greatly reduced over transaction-based systems.

In addition, NVM programming requires tools that reassure a programmer after writing a program. If a power failure occurs, the last thing a programmer wants is to have a program failure after a restart or to find the program data in an inconsistent state. It is necessary to have tools that allow the programmer to reason about the order of the persistence of writes. To achieve this goal, we introduce a static data flow analysis tool to find NVM programming bugs during the compile time.

The statement of this thesis is:

> *We can provide software support for NVM programming by building checkpointing tools that allow building durable data structures with low crash-consistency overhead. We can also build tools based on dataflow analysis for finding NVM programming bugs without executing the program code.*

Firstly, we present *InCLL* a fine-grained checkpointing technique tailored for NVM with an undo logging strategy. *InCLL* shows that fine-grained intervals, on the order of milliseconds, are feasible to build durable data structures for NVM. Instead of solely relying on an external undo log for backing up data before doing in-place updates, *InCLL* keeps the log inside a cache line to avoid explicit cache-line write-back instructions, effectively improving runtime performance. *InCLL* ensures that no cache-line write-backs are necessary for the fast path data structure modifications using in-cache-line logging and fine-grained checkpointing. We describe our modifications to Masstree [117] and evaluate *InCLL*.

Secondly, we present *CpNvm* to correct the shortcomings of the *InCLL* algorithm. *InCLL* was specifically implemented for Masstree and required careful reasoning about the data structure layout. *InCLL* periodically executes a privileged instruction to flush the entire cache hierarchy (Dalí-style cache flush). The invalidation and eviction create challenges in multi-tenant systems and impact runtime performance by periodically increasing the cache-miss rate. Moreover, *InCLL* does not take the performance characteristics of Optane into account and experiences significant performance degradation when running on Optane.

With *CpNvm*, we marry the ideas from traditional application-level checkpointing and durable data structure design. We tailor *CpNvm* for Optane by using DRAM as a middle layer to do write-combining before propagating modifications to NVM. Heavily read or written data resides in DRAM, where it can be accessed at a fraction of the cost, and only modified values propagate to NVM periodically. *CpNvm* API is general and easy to use with a simple API. We show that *CpNvm* has low overhead for Masstree and Memcached with minimal code changes in both benchmarks.

Finally, we present *FlowNvm* to find bugs in NVM programs using static analysis. To reduce the state explosion of a program, we reduce the problem to a dataflow analysis and explore properties across all program paths. *FlowNvm* detects issues in program ordering, which can leave program data in NVM in an inconsistent state. *FlowNvm* defines programming patterns and anti-patterns and uses inter-procedural data flow analysis to detect the validity of the patterns and occurrence of the anti-patterns to find bugs. *FlowNvm* design is inspired by well-known correctness bugs such as failing to order the persistent writes properly or performance bugs such as flushing a cache line unnecessarily.

In summary, this thesis makes the following contributions:

- *InCLL*, a fine-grained checkpointing technique for building durable data structures with an undo logging approach, which minimizes the number of explicit cache line write-back instructions for fast-path data structure modifications.

  Specifically, its contributions are:

  - Fine-Grained Checkpointing, a technique to ensure a consistent, quickly recoverable data structure in NVM after a system failure.
  - In-Cache-Line Logging, an undo-logging technique that enables recovery of the state from the beginning of an epoch without requiring cache-line flushes in the normal case.
  - Implementation of these techniques for the Masstree data structure and making Masstree durable.
  - Evaluation of durable Masstree to demonstrate a low (< 20% DRAM evaluation) runtime overhead cost.

- *CpNvm*, a checkpointing tool using DRAM write-combining during execution and an NVM copy for fast recovery with a simple application-level checkpointing API.

  Specifically, its contributions are:

  - An application-level checkpointing API with checkpointing-based crash-consistency guarantees with low-overheads.
  - The design and implementation of *CpNvm*, a new C/C++ library that implements these checkpoints at low cost.
  - Evaluation of *CpNvm* using Masstree and Memcached on Optane, demonstrating its low run-time overhead.

- *FlowNvm*, static inter-procedural data flow analysis tool for finding NVM programming bugs using program properties and anti-patterns.

  Specifically, its contributions are:

- Identification and description of several NVM programming patterns and anti-patterns.
- Description of static program analysis for identifying many instances of these pattern violations and the existence of the anti-patterns.
- Implementation of an extension to LLVM for C/C++ applications that identifies and reports these bugs.
- Evaluation of this tool on existing NVM programs and demonstration that the tool can detect and report previously known errors as well as a new bug.

1.5 Thesis Organization

The thesis organization starts with detailed background information followed by the contributions. Chapter 2 provides the background context for NVM Programming, crash-consistency mechanisms for NVM, and checkpointing. Chapter 3 describes the design, implementation and evaluation of *InCLL*. Chapter 4 describes the design, implementation and evaluation of *CpNvm*. Chapter 5 describes the design, implementation and evaluation of *FlowNvm*. Chapter 6 summarizes the key points of the thesis, presents the future work, and concludes.

1.6 Bibliographic Notes

The major implementation work for *InCLL* was done in 2018-2019 and Optane evaluations were done in 2020. *InCLL* is a joint work previously published with Nachshon Cohen and Hillel Avni in the paper: "Fine-Grained Checkpointing with In-Cache-Line Logging". Nachshon came up with the idea and contributed to the implementation. Hillel Avni contributed to the design. The Optane evaluation was done independently.

2 Background

In this chapter, we describe NVM programming in detail and provide the technical background to understand the contributions. We describe how programmers can use NVM as durable memory, the ordering axioms for properly persisting writes, and the failure model. We illustrate high-level approaches for designing crash-consistency mechanisms. We explain the performance advantages of using checkpointing as a relaxed crash-consistency model.

2.1 NVM Programming

We first describe the NVM programming specification [164] and explain how to order the writes to make them persistent. We specify the axioms and instructions that are necessary for ordering persistent writes. We specifically focus on the *x86-64* instruction set and Linux operating system, which is our evaluation setting. We discuss some other instruction sets, deprecated configurations, and operating systems.

The Storage Networking Industry Association (SNIA) is a non-profit organization that produces standards for the storage industry. SNIA's NVM Programming Model specification is currently the accepted standard for using NVM. The first version was published in 2013 and contains the memory-mapped file paradigm for using NVM as a durable memory device. We discuss the recent version of the standard (v1.2 [164]) published in 2017. We mainly focus on the App-Direct Mode with a byte-addressable interface, not the File I/O Mode, which provides block-level access.

The NVM Programming Model specifies that the primary method to provide durable memory address space for NVM programming is to use *memory-mapped* files with the *Direct Access (DAX)* feature. A programmer uses the `mmap` system call on Linux and `MapViewOfFile` call on Windows to map NVM into the processor's virtual address space. A programmer then accesses NVM through loads and stores to these mapped addresses.

Traditionally, modifying data in durable storage media such as SSD or disk requires bringing pages from the storage media to page caches, which are in DRAM memory. In this situation, calling `mmap` [10] duplicates the persistent data in memory. The second copy in the page cache wastes space [24] and can cause performance degradation [100]. Both Linux and Windows support the *DAX* feature, which allows a program to bypass page caches and access NVM directly. DAX is a crucial feature for NVM Programming as it eliminates the unnecessary overhead of the kernel I/O stack and context switching. However, using DAX introduces programming challenges when dealing with failures such as system crash, power failure, or reboot.

2.1.1 Failure Model

To understand the challenges related to NVM programming, we must first discuss its failure model. Our failure model is based on the *fail-stop/crash-recovery* as described in Atlas [32]. If a failure such as a power failure occurs, the process crashes and stops. After the cause of the failure is mitigated, such as providing power again, the process restarts and continues execution. We do not consider Byzantine faults [103].

Because of the failure model, the notion of persistence domain becomes important. A persistence domain specifies which parts of the hardware are durable (retain values) even after a power failure. The current industry standard is to have *Asynchronous DRAM Refresh (ADR)* as the persistence domain [155]. We described ADR in section 1.1.1. ADR is available with Cascade Lake *x86-64* processors.

The NVM Programming specification also includes another persistence domain configuration, *Enhanced Asynchronous DRAM (eADR)* [11]. *eADR* extends the persistence domain to include the processor caches, effectively making caches durable. The eADR feature is provided with 3rd generation Ice Lake chips.

For the thesis, we only consider the ADR persistence domain. Both *InCLL* and *FlowNvm* require and assume an ADR system. On the other hand, *CpNvm* offers benefits to eADR systems as its main goal is to improve the performance of a data structure while using Optane.

2.1.2 NVM Programming Primitives

> *The key design decisions in this thesis stem from the fact that processor caches are* **volatile** *(ADR persistence domain) and a program might crash at a random point.*

In *x86-64*, NVM programming primitives allow (1) ordering writes and (2) guarantee that the data is in the persistence domain. Ordering and persistence can be done using cache line flush or non-temporal move combined with fence instructions. For normal stores to the memory, controlling the eviction of cache lines explicitly is necessary for persistence. *x86-64* provides cache line flush instructions (`clflush`, `clflushopt`, `clwb`) or entire cache hierarchy flush instruction (`wbinvd`) to evict cache lines from the software. Another approach is to bypass the caches for persistent writes and propagate data directly to the persistence domain. *x86-64* provides non-temporal move instructions (`movnti`,`movntq`, `movntps`, `movntpd`, `movntdq`, `maskmovq`, `maskmovdqu`) to bypass the caches. Fence instructions are useful for ordering both cache line flush instructions and weakly-ordered non-temporal move instructions. *x86-64* provides fence instructions (`sfence`, `mfence`) to ensure that outstanding memory operations are finished.

Cache Flush

These instructions flush cache lines:

clflush Invalidates and transfers a single cache line to the memory controller.

clflushopt Invalidates and transfers a single cache line to the memory controller. Instruction executes asynchronously.

clwb Transfers a single cache line to the memory controller. The cache line is not invalidated. Instruction executes asynchronously.

wbinvd A privileged serializing instruction that invalidates the entire cache hierarchy of the CPU and writes back all dirty cache lines.

Invalidating a cache line removes it from the cache hierarchy. Invalidating ensures that the next access to it will trigger a *cache miss*. `clflush` is synchronous and ensures that the operation completes when the instruction finishes. `clflushopt` and `clwb` are asynchronous instructions that are non-blocking and can be run concurrently for different cache lines. Both `clflushopt` and `clwb` initiate the cache line transfer. However, these instructions return before the cache line is flushed to the memory controller.

The `clwb` instruction is expected not to invalidate the cache line explicitly. However, it should be noted that Cascade Lake chips use `clflushopt`'s mechanism for the implementation of `clwb` and `clwb` invalidates the cache line [170]. We do not rely on `clwb` instruction in our checkpointing tools.

`wbinvd` invalidates all cache entries and writes back dirty cache lines on the CPU that executes the instruction. The `wbinvd` instruction flushes and writes back the L1 and L2 caches local to the CPU and the shared L3 caches. It is synchronous and waits until completion. As `wbinvd` invalidates the cache hierarchy, subsequent program performance will be lower due to cache line misses. Sizes of the caches are a major factor in the execution time of `wbinvd` instruction. As `wbinvd` is serializing, a fence instruction is not necessary for ordering.

Non-temporal Move

Non-temporal move instructions allow bypass of the caches [88]. The cache line corresponding to the memory address is not fetched and the cache hierarchy is not used. Writes directly propagate to the memory controller using *write-combining (WC)* buffers. Non-temporal move instructions are weakly ordered and require fence instructions for ordering. Otherwise, non-temporal stores to two different memory regions can happen independent of the program order and can lead to bugs.

Fence

The Intel *x86-64* memory model is *Total-Store-Order (TSO)* [146]. In TSO, stores cannot be reordered after stores. Raad et al. [147] provide a formal model for the semantics of the ordering of operations for *x86-64* systems. In the thesis, we focus only on a subset of the model. We consider the ordering of the write instructions, non-temporal move instructions, cache-line flush instructions, and fence instructions.

These instructions act as a fence:

sfence Orders stores and flushes.

mfence Orders loads, stores, and flushes.

Both `sfence` and `mfence` are useful for ordering stores and flushes. However, `mfence` also orders loads. Usually relevant work in this field solely focuses on `sfence` as a fence instruction [159]. A single fence instruction is sufficient to ensure the durability of all the previous cache line write-back and non-temporal move instructions.

```
x = 1;
clflushopt(&x);
sfence();
```

(a) Write followed by `clflushopt` and fence.

```
x = 1;
clflush(&x);
```

(b) Write followed by `clflush`

```
ntm(&x, 1);
sfence();
```

(c) Non-temporal write followed by fence.

Figure 2.1 – Possible ways to write data to NVM durably.

Figure 2.1a shows an example of `clflushopt` usage. Figure 2.2 shows how multiple cache lines are flushed concurrently. Completion of the write back is ensured by the `sfence`.

We can think of `clflush` as executing a flush and a fence instruction. The reason is that once `clflush` executes, it is synchronous and runs to completion. Figure 2.1b shows an example for `clflush`.

Non-temporal move instructions are weakly ordered. It is possible that two writes to the

```
// x and y are in different cache lines
x = 1;
y = 1;
clflushopt(&x); // initiate write back of cache line belonging to x
clflushopt(&y); // initiate write back of cache line belonging to y
sfence(); //both x and y are durable
```

Figure 2.2 – Flushing multiple cache lines.

same cache line do not occur in program order unless there is a fence instruction in between. Figure 2.1c shows an example for non-temporal move instructions.

Another important idea is to order the writes to the same cache line. Ensuring that a write reaches NVM is expensive. On the other hand, ordering the writes to a cache line is almost free. If there are two writes to the same cache line, the order of writes reaching the cache is the same as the order of reaching the persistence domain.

It is possible to use release memory ordering [25] from C++11 to order writes to the same cache line. A release fence creates a *happens-before* relation between writes. This release fence ensures that the writes are ordered as intended by the program order. In *x86-64*, a release fence does not incur any extra runtime overhead and only prevents the compiler from reordering writes.

Ordering Axioms

In this section, we define the formal ordering axioms, which we use for building crash-consistent NVM programs. Formally, we assume different memory regions x, y and denote the cache line addresses via $c(x)$ and $c(y)$. $c(x) = c(y)$ denotes that the memory regions are in the same cache line. Table 2.1 defines the formal operations and the relations that we use.

Operation	Explanation
$w(x)$	write to memory region x
$w_n(x)$	non-temporal write to memory region x
$f(x)$	synchronous flush memory region x (e.g., `clflush`)
$f_o(x)$	asynchronous flush memory region x (e.g., `clflushopt, clwb`)
$p_f()$	persist fence (e.g., `sfence`)
Relation	**Explanation**
$<_{hb}$	standard happens-before
$<_p$	persists-before
$\sim_p$	both operations can persist in any order

Table 2.1 – Formal operations and relations for NVM programming.

The main goal is to ensure the persistence of writes. For crash-consistency mechanism design, it is necessary to be able to order the persistent writes correctly. Specifically, our goal is to persist $w(x)$ before persisting $w(y)$ if x is written before y.

The following axioms determine the order of persistence for normal write operations:

- 1) $w(x) <_{hb} w(y) \wedge c(x) = c(y) \Rightarrow w(x) <_p w(y)$ (same cache-line)
- 2) $w(x) <_{hb} w(y) \wedge c(x) \neq c(y) \Rightarrow w(x) \sim_p w(y)$ (different cache-line)

The first axiom states that if the writes $w(x)$ and $w(y)$ are to the same cache line, the first write $w(x)$ will not persist later than the second write $w(y)$. The second axiom states that two writes can persist out-of-order if the writes are to different cache-lines.

```
// Initially x=0, y=0
x = 1;
y = 1;
// Possible NVM state: x, y ∈ {0,1}
```

Figure 2.3 – Program without flush, non-temporal move, and fences.

The second axiom can lead to the situation as presented in figure 2.3. The program writes value 1 to x, then writes value 1 to y where initially both x and y are 0. If the program crashes after executing `y = 1` and is restarted afterward, NVM can be left in a state where the value of y is 1 and the value of x is 0. This is inconsistent with the program order and can lead to hard-to-find bugs [115].

We need to ensure that a write $w(x)$ persists before write $w(y)$. The following axioms show how to enforce this ordering:

- 3) $w(x) <_{hb} f_o(x) <_{hb} p_f() <_{hb} w(y) \Rightarrow w(x) <_p w(y)$ (asynchronous flush)
- 4) $w(x) <_{hb} f(x) <_{hb} w(y) \Rightarrow w(x) <_p w(y)$ (synchronous flush)
- 5) $w_n(x) <_{hb} p_f() <_{hb} w(y) \Rightarrow w(x) <_p w(y)$ (non-temporal move)

Axioms 3, 4, 5 ensure the durability of the writes by propagating the write to the persistence domain. The specific programming pattern ensure that the second write does not propagate to the persistence domain before the first write. The third axiom specifies that asynchronous flush needs a fence instruction to complete. The fourth axiom specifies that after synchronous flush instruction completes, the write is durable. The fifth axiom specifies that a non-temporal move requires a fence instruction to order the two writes. In these axioms $(3-5)$, the second write $w(y)$ can be done using non-temporal move instruction $w_n(y)$ and the persistence order remains the same.

2.1.3 Alternative and Earlier Programming Models

The current NVM programming specification [164] evolved over many years. We briefly mention earlier models to set our work apart from the previous models. When persistence domain did not include the buffers in the memory controllers, programmers used the `pcommit` [4] instruction to explicitly flush memory controller buffers to NVM. The `pcommit` is now deprecated. In earlier models, `clflush` was weakly ordered and required the use of the `mfence` instruction [88].

ARM has a weak memory ordering model [148]. Stores can be reordered with respect to other stores [110], which can be prevented by using a fence instruction such as `dmb`. ARM provides durability and ordering instructions. The "Data or unified Cache line Clean by VA to PoP" (`dc cvap`) instruction, which is similar to `clflushopt`, flushes a cache line to the memory controller asynchronously. ARM also provides "Data or unified Cache line Clean by VA to PoDP" (`dc cvadp`) instruction, which flushes a cache line asynchronously to NVM directly. This is useful for cases where the programmer does not trust the persistence of the ADR domain. "Data Synchronization Barrier" (`dsb`) is a memory barrier instruction, which has similarities to `mfence` as it orders loads and stores while ensuring the completion of the previous write-back instructions.

If *DAX* mode is not used, range-based `msync` or file-based `fsync` and `fdatasync` calls can be used. These calls are suitable for the File I/O mode and ensure that the changes to the memory-mapped file get propagated to NVM durably. These calls are block-based. For *DAX* it is better to do cache line flushes for performance [187].

There are many persistency models proposed in academia [90, 137, 146, 148] and different instruction set and hardware proposals [126, 159] to handle persistence.

2.1.4 Optane Configuration

In this section, we explain Optane configuration [6] used for evaluating our checkpointing contributions. The bug finding tool does not require a machine with Optane. We use `ipmctl` and `ndctl` command-line programs to configure NVM. `ipmctl` program is useful for configuring the operation mode and enable/disable interleaving. `ndctl` program is useful for using Optane as a durable memory device with a byte-addressable interface in App-Direct Mode. Figure 2.4 and 2.5 show how to configure NVM.

```
> ipmctl create -goal MemoryMode=100 #Memory Mode configuration (1)
> ipmctl create -goal PersistentMemoryType=AppDirect #App-Direct configuration (2)
> ipmctl create -goal MemoryMode=25 Reserved=50 #Mixed Mode configuration (3)
```

Figure 2.4 – Provisioning NVM for different operation modes using `ipmctl`.

```
> ndctl create−namespace −m fsdax #Namespace creation using fsdax (1)
> mkfs.ext4 −F /dev/pmem0 #Create a file system to manage the device (2)
> mount −o dax /dev/pmem0 /mnt/memext4 #Mount (3)
```

Figure 2.5 – Durable memory device configuration using `ndctl`.

2.2 Crash-Consistency Mechanisms for Programming

To recap, *crash consistency* allows a program to ensure a consistent state after events such as power failure, system failure, reboots. We present high-level design ideas [114] for building crash-consistency mechanisms. The high-level design ideas are mainly software-based solutions and do not require architectural changes to the hardware.

To implement crash-consistency mechanisms, we need to ensure that a set of operations can execute with *atomic durability* [73]. *Atomic durability*, is an **all-or-nothing** characteristic. It states that either all operations commit their changes to durable storage or none of the operations commit to durable storage.

Completion of a set of operations can be thought of in terms of either *transactions* [76] or *checkpointing* [99]. Transactions provide *atomicity, consistency, isolation, durability* (ACID) [145] properties. Transactions traditionally have stronger guarantees.

An alternative is to use checkpointing and persist data after a passage of time. Checkpointing [153] takes a snapshot of the program state and stores the state in a safe location where it can be accessed after a restart. The state can be the program state [138] including the stack and the registers or a subset of the program data in the heap [185]. In either case, recovery after a crash returns the state to the last valid checkpoint. The safe media can be durable storage such as disk and SSD [138], NVM [185], parallel file system [22] or memory of other processes [139]. We provide a detailed discussion about checkpointing in section 2.3.

Implementations of transactions and checkpointing relies on several design ideas and mechanisms. The *x86-64* ensures that up to 8-byte aligned stores are atomically executed [157]. We describe the high-level software design patterns (e.g., undo logging, redo logging, recovery via resumption, copy-on-write, operational logging) to provide a context for our design conributions.

Undo logging [37, 53] allows in-place updates. The program first stores the old value in a durable backup (undo log). Then, the program modifies the data in place and updates it with the newer value. If the set of operations commits, the program discards the backup. The new value is the latest version. If the set of operations fails, the program uses the backup data to replace the new value with its original old value.

Redo logging [68, 175] redirects both reads and writes to the latest version of the data in the redo log. During program execution, the program writes to the backup (redo log) and does not modify the program data. If the set of operations commits, the program applies the redo log to the original data and updates the old value. The log replay is idempotent. Even if a failure occurs during log replay, it is possible to replay the same log. If the set of operations fails, the program discards the redo log. No changes are necessary to the original data and the old value remains intact.

The main difference between *undo logging* and *redo logging* is the difference between in-place updates and read/write redirection. The program has to store the old value to the backup durably before doing the in-place update. On the other hand, redo log redirection can avoid ordering writes on the critical path of the program. However, read and write redirection incurs cost [118, 177].

Recovery via resumption [89, 113] stores sufficient information about the program execution. If the program fails, after a restart, the program can complete the execution of the failed code regions. Recovery via resumption executes parts of the program code after a restart, which is unlike undo and redo logging that apply data modifications to memory regions.

Copy-on-write (COW) [118] creates a copy of the original data for modifications. The copy is modified during the execution. If the set of operations commits, the program atomically replaces the original data with the updated data. In the program, the copy becomes the original. Afterward, the program discards the old data. If the set of operations fails, the program discards the copy and no changes are necessary for the original. However, this implies that there must be a pointer referencing the object to be atomically updated.

The main difference between *redo logging* and *copy-on-write (COW)* is atomic updates. Both redo logging and copy-on-write write to a copy of data stored elsewhere. The difference is that redo logging modifies the original data at commit time, while COW only updates a pointer to the copy.

Operational logging [122] logs only high-level operations. After a restart, the program needs to re-execute the operations in the log to restore that state of the data structure.

Checksum [183] is useful for minimizing explicit write-back instructions by checking the validity of the stored data after a restart. The program operates on the data and computes a checksum. After a restart, if the checksum matches the program data, then the program data is usable and correct.

There are many design parameters one needs to consider in designing a crash-consistency mechanism. Overall, the trade-offs are based on 1) runtime performance, 2) recovery overhead, 3) volatile memory and durable memory overhead, 4) programming simplicity, 5) validation and verification simplicity. There is no one-size-fits-all solution. This leads to many proposals for durable data structures and runtime systems for NVM programming [32, 37, 108, 112, 121, 175, 149, 185].

2.3 Checkpointing

Checkpointing is useful in many domains and is widely used in areas such as databases [77, 161], high-performance computing [58, 178], virtualization [179, 191], debugging [59, 167], process migration [111], and computations in power-constrained IoT environments [66, 96]. Checkpointing designs differ with respect to availability, checkpointing overhead, and fault-tolerance.

A program can take checkpoints periodically [153, 171], at user-specified points [138], after hitting the limit of the log threshold size [49], or during program shutdown [102]. We denote the time spent between two checkpoints as an *epoch*. There is no one-size-fits-all solution for determining the time to initiate checkpointing. Selecting the optimal time at which to take checkpoints is known as the *checkpoint placement* problem and the solutions are domain-specific [99]. The size of the checkpoint, the amount of lost data, the time spent on checkpointing, and the runtime overhead are factors in determining the placement of checkpoints. Different systems such as real-time systems, distributed systems, shared-memory systems all have different constraints.

Checkpointing can be implemented at different privilege levels [99]. There are two major levels, kernel-level checkpointing [70] and user-level checkpointing [50]. In *kernel-level checkpointing*, the kernel thread can checkpoint a program without changing the program code. The kernel is responsible for recovery after a restart.

User-level checkpointing uses a checkpointing library. User-level checkpointing can occur *transparently* to the program. User-level checkpointing can also be done using user-directed annotations [138] requiring changes to the program. These code changes can exclude memory locations and determine when to take a checkpoint. Another form of user-space checkpointing is *application-level checkpointing*. The programmer modifies the program code to implement checkpointing and recovery functionality. In all these approaches, checkpointing can use timer interrupts to periodically persist program data.

There are many optimizations to reduce the checkpointing size and minimize the programmer effort. For libraries and approaches that require program modification, *compiler-assisted checkpointing* [21, 44, 57] minimizes the programmer effort.

Another optimization is *incremental checkpointing*, also known as delta checkpointing [49, 59, 138]. Incremental checkpointing records only the modifications for the current epoch instead of checkpointing the entire program data. This can reduce the checkpoint size.

Likewise, *data compression* techniques are useful for reducing the size of checkpoints [139, 140]. The main parameters are the time spent on compression, the compression ratio of the program data, total time spent in checkpointing. Approximate checkpointing techniques can further compress program data by allowing lossy compression [36, 131]. During checkpointing, approximate checkpointing techniques allow loss of modifications within a bounded error. Approximate checkpointing is domain-specific and is usually used in scientific programs. Speculative checkpointing [172] eagerly checkpoints pages from the working set of the program to avoid checkpointing pages during checkpointing pause interval.

Long-running, large-scale programs such as HPC scientific programs use checkpointing [55, 58, 92, 138, 154, 178]. Traditionally, HPC checkpointing occurs in intervals of seconds to hours. The granularity of checkpointing is a page [1].

Databases [77, 161] often rely on checkpointing as well. However, different databases use checkpointing in different contexts. Relational database designs, typically follow the ARIES [124] recovery model, where transactions provide ACID guarantees [145] and checkpointing acts as a means to truncate logs and speed-up recovery time. Checkpointing can store relational database critical data structures such as the *Transaction Table* and *Dirty Page Table*. To reduce the cost of checkpointing the structures, the database can use fuzzy checkpointing techniques [165]. Fuzzy checkpointing takes a snapshot of the tables asynchronously and allows execution to proceed.

3 Fine-Grained Checkpointing with InCLL

Computer Science is a science of abstraction - creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.
— Alfred Aho

The main challenge in building durable data structures for NVM is to design systems that effectively deal with the volatility of the caches and provide high runtime performance and low recovery cost. After a restart, the program should be able to access its data and continue execution without encountering a failure or program data inconsistency. Providing crash consistency for correct recovery after a restart is an essential feature of NVM programming. However, crash-consistency mechanisms often rely on persisting writes eagerly along the critical path of a program that leads to the execution of expensive cache line write back and fence instructions. The overhead from explicit write-back and fence instructions can be significant for program performance [118, 189]. Previous work [17, 40, 128, 189] focuses on minimizing the use of these instructions.

In this chapter, we introduce *InCLL*, which is a durable data structure design technique for minimizing explicit write-back instructions for fast-path data structure modifications, such as insertions, updates, and deletions. The entire data structure resides in NVM, which is stored durably and accessible after a restart. *InCLL* breaks the execution into epochs and periodically writes back the caches to NVM using *fine-grained checkpointing*. If a crash occurs, after a restart, the data structure can contain modifications from the failed epoch. *InCLL* provides a new undo-logging technique (*in-cache-line logging*) to roll back modifications from the failed epoch. The use of an in-cache-line log can avoid explicit write-back and fence instructions. We implement the adaptation of these ideas into Masstree and make Masstree durable. We provide the evaluations of *InCLL* both for DRAM and Optane.

This chapter is organized into the following sections:

- Section 3.1 introduces *InCLL* by describing the design goals and the motivation based on simulation and explains the contributions of *InCLL*.

- Section 3.2 describes the two main components of *InCLL*, which are fine-grained periodic checkpointing for periodically propagating writes from the cache hierarchy to NVM and in-cache-line logging to ensure correct recovery.
- Section 3.3 describes the Masstree data structure and explains how *InCLL* incorporates the two components (fine-grained checkpointing and in-cache-line logging) into Masstree.
- Section 3.4 evaluates *InCLL* on DRAM and Optane devices.
- Section 4.5 compares *InCLL* to related work and this section is presented in chapter 4 after introducing *CpNvm*.

We specifically note that we use *InCLL* as a general umbrella term for the entire system. The *InCLL* idea is a combination of in-cache-line logging, fine-grained checkpointing, and persistent memory allocation. However, we use InCLL$_p$, InCLL$_1$, InCLL$_2$, and InCLL$_a$ as specific instances of an in-cache-line log.

3.1 Introduction

In section 2.2, we explained how transactions can be useful for persisting program data atomically. Typically, transaction implementations rely on a form of write-ahead logging such as redo logging or undo logging. Undo logging relies heavily on explicit write-back and fence instructions on the critical path of a data structure, which can be costly [118, 177]. This is due to the nature of the implementation of an undo log [37, 141].

For undo logging, each write is first logged durably to a backup location (log entry) before applying it to the data structure. Traditional undo logging in total can require the execution of two write-back and two fence instructions for a single write operation during execution. The logging mechanism first writes back and fences the log entry to make it durable, then updates the undo log and makes the changes to the log durable. It is also possible to reduce logging to a single write-back followed by a fence instruction. In any case, doing a write back is necessary in the critical path of a program.

As we explained in section 2.2, an alternative is to use checkpointing instead of transactions. Checkpointing saves a program's entire state on durable media. After a failure, the program continues execution by first restoring the last recorded state. As the durable media is usually slow, copying the program state to the durable media takes time and is expensive. For this reason, the program takes checkpoints infrequently on the order of minutes to hours to minimize this cost.

InCLL uses fine-grained checkpoints instead of traditional checkpoints. Fine-grained checkpoints divide the program execution into epochs. *InCLL* keeps only the data structure state in NVM, not the entire program state. *InCLL* ensures that after a crash, it is possible to recover the state of the data structures to the end of the last successful epoch. At the beginning of each epoch, *InCLL* takes a checkpoint by flushing the processor caches to NVM. This ensures that the data structures in NVM are in a consistent state before beginning the new epoch.

There are many benefits to using fine-grained checkpointing. The entire data structure can be placed in NVM benefitting from NVM's high capacity. NVM has lower performance compared to DRAM, however, as new technologies emerge, it is possible for NVM performance to improve significantly.

Unlike traditional checkpointing, a separate copy of the data structure is not necessary. The data structure that resides in NVM acts as both the data structure and the checkpoint. *InCLL* can support fine-grained intervals on the order of milliseconds, as the time spent on checkpointing is bounded by the cache sizes and the number of dirty cache lines. The hardware batches the flush of dirty cache lines, which can further reduce this cost.

One issue with fine-grained checkpointing is dealing with the modifications done within a failed epoch. Modifications done in an epoch can propagate from the caches to NVM, which can complicate recovery. If a crash happens during an execution, *InCLL* treats that epoch as a *failed epoch.* After a restart, the data structure can contain modifications from the failed epoch and these modifications must be rolled back.

To roll back the modifications, *InCLL* needs to track data structure modifications. A traditional solution would be to use an undo log. However, undo logging can be expensive due to write-back and fence instructions on the critical path of the program. Even if an undo log is a general solution and applicable to any data structure, we don't fully benefit from using fine-grained checkpointing, which aims to minimize the number of write-back and fence instructions on the critical path.

Before introducing the in-cache-line log idea, we describe additional system assumptions for persistence. *InCLL* was done in era that predates Optane. The main assumption was that a single cache line is the unit of persistence as the memory controller uses cache-line granularity (64-bytes) for data transfers. Therefore, it is possible to persist data with cache-line granularity as long as writes are ordered to the same cache line correctly.

The real contribution of *InCLL* is an in-cache-line log. The *key idea* of an in-cache-line log is to use cache-line granularity for atomic persistence and build an undo log without any write-back and fence instructions.

The main difference between an in-cache-line log and a traditional undo log is the placement of the log. An in-cache-line log resides in the same cache line with the data that it is logging. To contrast, a typical undo log resides in an external space separate from the data. We place the log with its data in the same cache line, because two persistent writes to the same cache line can be ordered without a write back or a fence (Section 2.1.2). This allows in-cache-line logging to avoid write-back or fence instructions for modifications that it can log.

We implement *InCLL* for Masstree [117], which is a combination of B+ tree and Trie data structure. We extend in-cache-line logging to provide a persistent memory allocator. We evalute the system on a machine equipped with Intel Optane (Section 3.4.1). We demonstrate that the results are feasible in simulation (Section 3.4.7) similar to what we previously reported [38]. The Optane evaluation incurs significant overheads, which led us to an improved design that we present in the next chapter (Chapter 4).

The main contributions of this work are:

- Fine-Grained Checkpointing, a technique to ensure a consistent, quickly recoverable data structure in NVM after a system failure.
- In-Cache-Line Logging, an undo-logging technique that enables recovery of the state from the beginning of an epoch without requiring cache-line flushes in the normal case.
- Evaluation of *InCLL* for durable Masstree both in simulation and on Optane.

3.2 InCLL Design Overview

The main novelty of *InCLL* design is the combination of fine-grained NVM checkpointing with an in-cache-line log to avoid expensive write-back and fence instructions. In this section, we first describe the Masstree data structure to motivate the design of the *InCLL* approach. Then, we describe the essential components that allow *InCLL* to be usable in practice, which are fine-grained checkpointing, in-cache-line logging, and external logging. Fine-grained checkpointing allows periodic checkpointing using cache flushes. An in-cache-line log is an undo log that does not use explicit write-back and fence instructions. External logging is used for cases where the capacity of an in-cache-line log is not sufficient. We described the ordering axioms and persistency model assumptions in section 2.1.2.

3.2.1 Masstree Data Structure

Masstree [117] is a high performance, production-quality Trie and B+ Tree hybrid data structure, carefully designed to exploit prefetching, optimistic navigation, cache-line awareness and fine-grained locking. Masstree is also used as an index for in-memory databases such as Silo [169].

We provide an abstraction of the Masstree data structure (Figure 3.1) to simplify the discussion for the *InCLL* approach. Overall, Masstree contains two types of nodes, which are *leaf nodes* and *internal nodes*. We focus on the leaf nodes as *InCLL* takes advantage of the leaf node design. Another reason for focusing on leaf nodes is that there is roughly an order of magnitude more leaf nodes than internal nodes. We describe the parameters, the arrays for storing data, and the `permutation` field that plays a crucial role in insertions and deletions.

```
 1 class basenode; // lock, version, meta information
 2 template <int width=15>
 3 class leafnode : public basenode{
 4     basenode *parent, *prev, *next;
 5     uint64_t permutation; // which key/vals are active
 6     keytype keys[width];
 7     valuetype *vals[width];
 8     void remove(keytype key){
 9         int idx = find_idx(key);
10         remove_idx(&permutation, idx);
11     }
12     void insert(keytype key, valuetype *val){
13         int idx = find_idx(&permutation);
14         keys[idx] = key;
15         vals[idx] = val;
16 
17         // make insertion visible
18         add_idx(&permutation,idx);
19     }
20     void update(int idx, valuetype *val){
21         vals[idx] = val;
22     }
23 };
```

Figure 3.1 – Masstree's leaf node structure.

Figure 3.1 depicts the internal structure of a Masstree leaf node. The number of key-value pairs in a leaf node is a template parameter (Line 2). The default leaf node holds 15 keys and 15 pointers to values. The keys and the values are stored separately in two different arrays, the `keys` array (Line 6) and the `vals` array (Line 7).

The structure does not store the value buffers inside Masstree. The value buffers are stored in an external area separate from the Masstree data structure and accessed using the pointers in the `vals` array.

Masstree uses the 64-bit `permutation` field (Line 5) to make insertions and deletions atomically visible. In addition, the `permutation` field is used to find unoccupied entries within the `keys` array and the `vals` array. Figure 3.2 shows the layout of the `permutation` field. The 64-bit field is divided into 4-bit regions. The lowest 4-bit entry represents the number of elements. Each of the other 4-bit regions represent an ordered array of 15 elements for 15 indices of the `keys` and `vals` arrays. Simply, the `permutation` field can be considered as a bitmap indicating if an index entry is used that also provides ordering information for the entries. In figure 3.2, there are two indices that are used, where indices 2 and 4 are occupied.

Deletions (Line 8) modify only the `permutation` field (Line 10) and remove the visibility of the

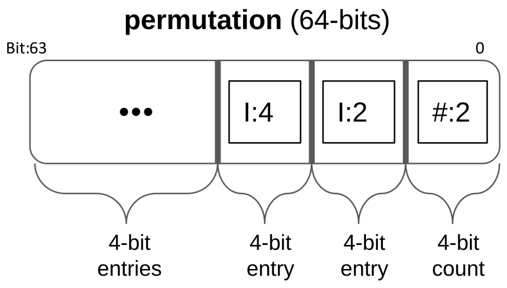


Figure 3.2 – Layout for the `permutation` field.

deleted entry. Insertions (Line 12) add the key to the `keys` array (Line 14), modify the pointer in the `vals` array (Line 15) to point to the *value buffer* that contains the value, and updates the `permutation` field (Line 18) to make modifications visible. Masstree splits a node when a free slot does not exist.

Masstree supports node splits and merges, however, these operations happen less frequently than modifications to leaf nodes. The full details of the Masstree algorithm are quite involved [117], but are not necessary to understand the contributions of *InCLL*.

Masstree leverages epoch-based memory reclamation for allocations and deallocations. The default setting for epoch-based memory reclamation is 64ms.

3.2.2 Fine-Grained Checkpointing

InCLL breaks the program execution into 64ms epochs, which is also memory reclamation epoch in Masstree. We provide evaluations for shorter and longer epoch intervals (Section 3.4). *InCLL* flushes the entire cache hierarchy to NVM using the serialized `wbinvd` instruction at the beginning of an epoch, effectively making the entire Masstree data structure durably consistent.

3.2.3 In-Cache-Line Logging

In-cache-line logging is an undo log technique that eliminates write-back and fence instructions entirely. For the Masstree implementation, *InCLL* embeds undo logs inside the Masstree leaf node.

The undo logging algorithm for using an in-cache-line log is straightforward. The program first stores the old value for the Masstree leaf node data in the in-cache-line log. Then the program updates the data.

One major issue with the in-cache-line log is the capacity of the log. The log occupies space within the same cache line as its data. If the in-cache-line log is small, then the log cannot capture all the modifications for a cache line. On the other hand, if the in-cache-line log occupies a larger space, this can reduce program performance by reducing the effectiveness of caching. In addition, while a log entry can be discarded when it is no longer useful, an in-cache-line log cannot be discarded as it always resides along with its data.

A small in-cache-line log implies limited coverage within a cache line. It is possible that the in-cache-line log space is not sufficient to record all the modifications for a cache line. For such cases, *InCLL* falls back to using a traditional undo log.

3.2.4 External Logging

The external log is a standard *undo log* [32, 40, 100, 141] and ensures that the modifications for the failed epoch can be correctly rolled back after a power failure. The main purpose of external logging is to handle complex operations that cannot be handled by the in-cache-line log. We describe each case where an external log is used in section 3.3.3.

We prefer object-level granularity logging, which logs an entire node. The reason for this design decision is the possible performance benefits and the simplicity of the design. When a complex operation occurs for a leaf node, we log the entire node once within an epoch, and all the subsequent modifications to the node will not require subsequent logging. Object-level logging does not require pervasive changes to the Masstree code. Furthermore, the log size only grows during an epoch and is reset by periodic checkpointing.

We describe the implementation details about how to combine an in-cache-line log with an external log in section 3.3. Overall, the main idea is to use an in-cache-line log for fast-path operations, while using the external log for slow-path operations and cases with insufficient in-cache-line log capacity.

In practice, the combination of in-cache-line logging and external logging works well. If Masstree updates are random, most accesses are to different nodes. *InCLL* uses the in-cache-line log to handle the few modifications and uses the external log infrequently. On the other hand, if the modifications are ordered or if there are hot nodes that are accessed frequently, most modifications go to a small set of nodes. External logging captures the modifications of the popular nodes efficiently and is used once per node per epoch, which amortizes the cost of logging a node in an epoch.

3.3 InCLL Implementation

We implement *InCLL* approach into the Masstree [117] data structure and make Masstree durable[1]. We assume that *InCLL* is a technique for expert library developers and the implementation requires careful programming.

[1]Our code is available at: https://github.com/epfl-vlsc/Incll

3.3.1 Fine-Grained Checkpointing Intervals

We explained the fine-grained checkpointing methodology in section 3.2.2. We implement a kernel module to execute the priviliged `wbinvd` instruction (which takes around 4.3ms to execute, more details in section 3.4.7).

On top of this, we implement an 32-bits epoch counter with a durable monotonically increasing index. A 32-bit index is sufficient for keeping track of fine-grained epochs and wraps after 8 years for 64ms epochs. If program data lives longer, a background thread could run every 8 years to reset all epoch indices back to zero.

InCLL also keeps track of failed epochs for correct recovery. After a restart, the recovery code adds the epoch that had a crash to the *failed epoch* set. *InCLL* uses the recovery code to revert the modifications that are done in a failed epoch.

3.3.2 In-Cache-Line Logging

In-cache-line logging requires data structure layout changes to embed the log inside the cache line. Figure 3.3 shows the layout for the Masstree leaf node visually. *InCLL* uses InCLL$_p$ for logging the `permutation` field and InCLL$_{1,2}$ for logging a pointer to a value buffer in the `vals` array.

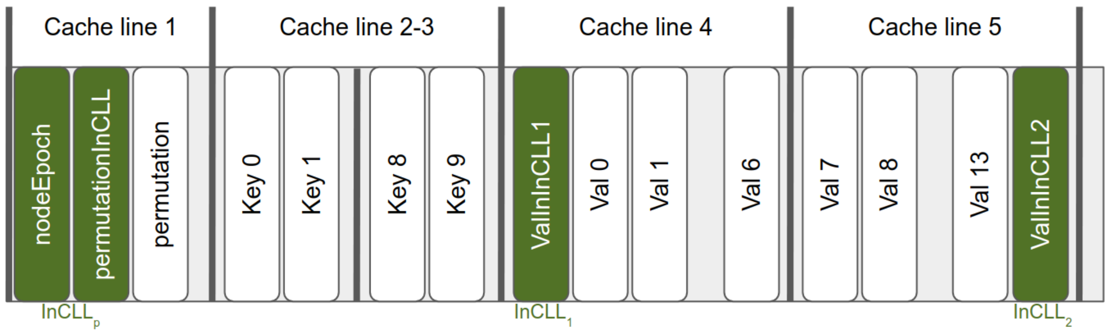


Figure 3.3 – Visual durable Masstree leaf node layout.

Figure 3.4 shows the implementation of a leaf node structure using in-cache-line logging. We reduce the number of keys and value entries from the default 15 to 14 (Line 11) to make space for two in-cache-line logs, InCLL$_1$ (Line 20) and InCLL$_2$ (Line 22). Each InCLL$_{1,2}$ can log a single pointer entry that points to a *value buffer*. Each InCLL$_{1,2}$ is carefully designed to be cache-aligned single words (8-bytes). InCLL$_1$ resides in the same cache line as pointers 0 – 6, while InCLL$_2$ resides in the same cache line as pointers 7 – 13 (as shown visually in figure 3.3). We use another in-cache-line log, InCLL$_p$ (Line 16) for the `permutation` field. Deletions, insertions, and updates modify the `permutation` field.

```
class basenode; // lock, version, meta information
struct ValInCLL{
    long idx:4;
    static const INVALIDIDX=−1;
    long ptr:44; // 48 bits minus 4 least significant bits
    long lowNodeEpoch:16; // last 16 bits of the epoch;
    ValInCLL(ptr, idx);
    ValInCLL():ptr(nullptr),idx(INVALIDIDX);
    void set(ptr, idx);
};
template <const int width=14>
class leafnode : public basenode{
    basenode *parent, *prev, *next;
    uint62_t nodeEpoch; // InCLLp field, 62 bits
    bool logged, InsAllowed; // InCLLp fields, 2 bits
    uint64_t permutationInCLL; // InCLLp field, 64 bits
    uint64_t permutation; // which key/val are active, 64 bits
    keytype keys[width];
    alignas(64) struct {} ALIGN; // align to cache line
    ValInCLL InCLL1; // same cache line as vals[0..6]
    valuetype *vals[width];
    ValInCLL InCLL2; // same cache line as vals[7..13]
};
```

Figure 3.4 – InCLL's leaf node structure.

In-Cache-Line Log Insertion and Deletion

We describe InCLL$_p$ layout for both insertions and deletions due to their similarity in using the log. Figure 3.4 shows the fields of InCLL$_p$. Insertions and deletions leverage the properties of the `permutation` field to minimize logging. Then, we describe the order of operations that are necessary for correct recovery. Figure 3.5 depicts the implementation details for insertions and deletions.

InCLL$_p$ Layout

For insertions, a slot for the key-value pair is unoccupied at the beginning of the failed epoch. There is no key-value entry to recover and logging the `permutation` field is sufficient. *InCLL* uses the `permutationInCLL` field to log the state of `permutation`. The same idea holds for subsequent insertions.

Deletions remove a key-value pair by updating the `permutation` field to indicate that the entry is unoccupied. As the entries in the `keys` and `vals` array are not overwritten, logging the `permutation` field is sufficient. The same idea holds for subsequent deletions.

Mixed insertions and deletions are more complicated. Logging the `permutation` field is not sufficient as insertion after a deletion can overwrite a key-value entry. Overwriting an entry complicates the recovery as the original key-value pair from the beginning of the epoch does

```
void leafnode::InCLL(bool InCLLallowed, permInCLL, valInCLL[2]){
    if(curEpoch != nodeEpoch){
        isInsertionsAllowed = true;
        isLogged = false;
        if(higher(curEpoch) != higher(nodeEpoch))
            isLogged = logNode();
        if(!isLogged){
            permutationInCLL = permInCLL;
            InCLL1 = valInCLL[1];
            InCLL2 = valInCLL[2];
            // order writes to the same cache line
            atomic_thread_fence(memory_order_release);
        }
        InCLL[1,2].lowNodeEpoch = lower(nodeEpoch);
        nodeEpoch = curEpoch;
    } else if(!isLogged && !InCLLallowed)
        isLogged = logNode();
    // order writes
    atomic_thread_fence(memory_order_release);
}
void leafnode::remove(keytype key){
    int idx = find_idx(key);
    InCLL(true, permutation, ValInCLL(), ValInCLL());
    InsAllowed=false;
    remove_idx(&permutation,idx);
}
void leafnode::insert(typetype key, valuetype *val){
    int idx = find_idx(&permutation);
    InCLL(InsAllowed, permutation, ValInCLL(), ValInCLL());
    keys[idx] = key;
    vals[idx] = val;
    add_idx(&permutation,idx);
}
void leafnode::update(int idx, valuetype *val){
    ValInCLL& InCLL = (idx<7) ? InCLL1 : InCLL2;
    InCLLallowed = (InCLL.idx == idx || InCLL.idx == INVALIDIDX);
    if(InCLL.idx == INVALIDIDX)
        InCLL.set(vals[idx], idx);

    ValInCLL vc1 = ValInCLL (vals[idx], idx);
    ValInCLL vc2 = ValInCLL(nullptr, INVALIDIDX);
    if(idx>=7)
        swap(vc1, vc2);
    InCLL(InCLLallowed, permutation, vc1, vc2);
    vals[idx] = val;
}
void higher(uint32_t epoch); // get upper 16 bits of epoch
void lower(uint32_t epoch); // get lower 16 bits of epoch
```

Figure 3.5 – Durable Masstree operations.

not exist anymore.

As shown in figure 3.4, we add nodeEpoch (Line 14), permutationInCLL (Line 16), insAllowed (Line 15), and logged (Line 15) fields to handle insertions and deletions. *InCLL* falls back to using the external log to handle mixed insertions and deletions. It does this by using the insAllowed field to allow insertions for the node within the epoch. After a deletion, *InCLL* sets the insAllowed value to false to disable mixed insertions and deletions. If insertions are not allowed for a node within an epoch, then *InCLL* logs the entire node to the external log and sets the logged field to indicate that the node is logged. If the entire node is logged, subsequent operations do not use any form of logging for the remainder of the epoch.

Masstree can grow large and contain many key-value pair entries. It is not feasible to reset all the in-cache-line log entries for a new epoch. For this reason, InCLL$_p$ also keeps track of the current epoch number using the nodeEpoch field.

Overall, InCLL$_p$ contains four fields:

- nodeEpoch: Keeps track of the epoch number for the leaf node.
- permutationInCLL: Acts as a backup log for the value of the permutation field at the beginning of the epoch indicated by nodeEpoch. Recovery code uses permutationInCLL to restore the value of the permutation field.
- insAllowed: Allows logging to fall back to using external log for mixed deletions and insertions. Indicates if the use of InCLL$_p$ is possible.
- logged: Indicates that the node is logged using the external log.

Insertion and Deletion Algorithm

We motivate the discussion for insertions and deletions using figure 3.5. The lines refer to figure 3.5 for this section. We discuss recovery in detail in section 3.3.4.

The algorithm runs as follows. *InCLL* first checks if the node was modified in the current epoch (Line 2). If the node is not modified, *InCLL* records the value of permutation in permutationInCLL (Line 8). It updates the nodeEpoch (Line 15) to current epoch and continues the operation. If the node is not from the current epoch and if insertions are not allowed (Line 16), *InCLL* logs the node (Line 17).

The recovery code can access leaf node fields and uses the information within an in-cache-line log to revert the changes. That is why, if nodeEpoch is updated before permutationInCLL, recovery can fail. If the write to nodeEpoch happens before the write to permutationInCLL, then the recovery code assumes that an older value for permutation is the value at the beginning of the epoch, which is not true. An older value from a previous epoch leads to an inconsistent state for the leaf node.

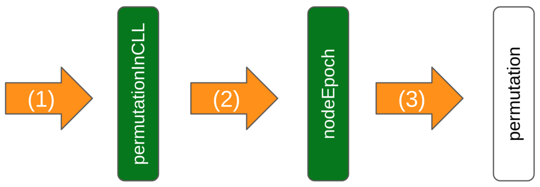


Figure 3.6 – Order of operations using InCLL$_p$ fields for insertions and deletions.

Figure 3.6 shows the order of operations of writes to `permutation`, `permutationInCLL`, and `nodeEpoch`, which is crucial for correct recovery. *InCLL* (1) updates the value of `permutationInCLL` to the current value of the `permutation` field, (2) sets the value of `nodeEpoch` to the current epoch, (3) modifies the value of the `permutation` field to reflect the insertion or the deletion of the key-value pair. *InCLL* does not require ordering for `insAllowed` and `logged`, as these fields are semantically transient.

The `permutation`, `permutationInCLL`, and `nodeEpoch` fields are all in the same cache line. It is sufficient to use a release memory fence [25] (described in section 2.1.2) to ensure ordering for the writes. By ensuring the order to the same cache line, an in-cache-line log can avoid write-back and fence instructions.

We described the algorithm to handle cases where InCLL$_p$ is sufficient for reverting changes within an epoch for insertions and deletions. We explain why this ordering is essential for recovery and how data propagates to persistence domain. Recovery only needs to revert the `permutation` field to correctly recover for these operations. There are four possible cases that we need to consider for cache line eviction as `permutation`, `permutationInCLL`, and `nodeEpoch` are in the same cache line.

1. If `permutationInCLL` is not modified, then `permutation` is also not modified, so there are no insertions and deletions to a node. This is safe for in-cache-line log recovery.

2. If only `permutationInCLL` is modified and if the cache line is in persistence domain, then recovery is not applied. The `nodeEpoch` field is not updated to the value of a failed epoch that had a crash. Recovery is unnecessary as `permutation` is not modified yet.

3. If both `permutationInCLL` and `nodeEpoch` are modified and if the cache line is in the persistence domain, then recovery is applied. However, the value of `permutation` and `permutationInCLL` are the same. Recovery is safe even if it is unnecessary.

4. If `permutationInCLL`, `nodeEpoch`, and `permutation` are modified and the cache line is in the persistence domain, then recovery is applied. As `permutation` is sufficient for correct recovery, *InCLL* reverts the value of `permutation` using `permutationInCLL`.

In-Cache-Line Log Update

InCLL updates are more complicated as they involve the $\text{InCLL}_{1,2}$. Figure 3.4 shows the fields of $\text{InCLL}_{1,2}$. Updates require logging the value pointers in the `vals` array. Figure 3.5 depicts the implementation details for updates.

$\text{InCLL}_{1,2}$ Layout

We embed $\text{InCLL}_{1,2}$ structures into the same cache lines as the `vals` array. Each $\text{InCLL}_{1,2}$ structure is capable of logging a single pointer entry to a *value buffer* within a single cache line. We compact the layout of $\text{InCLL}_{1,2}$ to specifically fit into a single 8-byte word. A single word layout allows 7 other pointer entries within the same cache line for a total of 14 key-value entries for each leaf node. If we use two words for a single $\text{InCLL}_{1,2}$ structure, then the number of entries reduces to 6 per cache line. Using more space for $\text{InCLL}_{1,2}$ reduces the number of key-value pairs which a leaf node can hold down to 12. The reduction in leaf node capacity would further incur a performance penalty.

We make the observation that the entries within the `vals` array are pointers to value buffers. We can log a pointer more efficiently than a standard value as the current *x86-64* only uses the lower 48 bits for addressing. The upper 16 bits are equal to the value of the 47^{th} bit for a valid memory address. Since all the memory allocations are aligned on 16 byte boundaries, the lower 4 bits are always zero as well. Overall, we can get 20 free bits for a pointer entry.

InCLL uses the free bits to store a compact epoch number and the index for the key-value pair. As shown in figure 3.4, the log contains the index (Line 3), value pointer (Line 5), and lower 16 bits of the epoch number (Line 6).

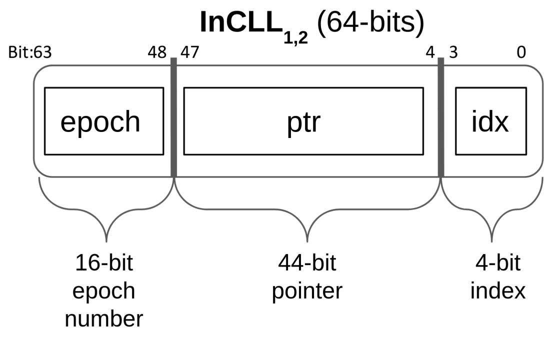


Figure 3.7 – $\text{InCLL}_{1,2}$ layout.

Figure 3.7 shows the $\text{InCLL}_{1,2}$ layout. *InCLL* uses bits 0 – 3 for the index of the logged entry. 4 bits are sufficient to represent all indices for the `vals` array entries and one extra *invalid* value. *InCLL* uses bits 4 – 47 to log the value of the pointer. *InCLL* uses the bits 48 – 63 for storing the lower 16 bits of the epoch number (`lowNodeEpoch`).

When *InCLL* modifies the node for the first time in the epoch, *InCLL* updates the $\text{InCLL}_{1,2}$ to the current epoch by updating the `lowNodeEpoch`. Our assumption is that we can combine the 16 bits from the `lowNodeEpoch` with the 16 higher bits of `nodeEpoch` from the InCLL$_p$ to form a full epoch number. If 16 bits are insufficient to correctly encode the epoch, *InCLL* falls back to using the external log, which happens approximately once an hour (2^{16} epochs with $64ms$ intervals).

There are architectural designs that can use the upper bits for addressing. Systems that use 5-level paging extend the addressable virtual address space and use the upper bits $48-57$ for addressing. As it uses the upper bits for the epoch number, we cannot use the same design for 5-level paging. One possible solution is to use the external log for addresses higher than 2^{48}.

Update Algorithm

We motivate the discussion for updates using figure 3.5. The lines refer to figure 3.5 for this section. We discuss recovery in detail in section 3.3.4.

The algorithm runs as follows. *InCLL* first checks if the node was modified in the current epoch (Line 2) similar to the insertions and deletions. If the node is modified for the first time in the epoch, then *InCLL* uses InCLL$_{1,2}$ for logging as well. *InCLL* first determines which InCLL$_{1,2}$ to use depending on the index number for the entry (Line 35). *InCLL* records the index and the value entry in the appropriate InCLL$_{1,2}$ and updates the epoch numbers. *InCLL* compacts the value of the logged pointer and the slot number into a single word and updates the corresponding InCLL$_{1,2}$ entry (Lines 9–10). Then, *InCLL* updates the epoch number for InCLL$_{1,2}$ by setting the `lowNodeEpoch` using the lower 16 bits of the `nodeEpoch`.

On top of the base case, if a node is updated in the current epoch and if the node is not logged, there are three special cases to consider.

1. Modification to the same entry with the same index: InCLL$_{1,2}$ contains sufficient information for recovery, so logging is unnecessary. Since the modification is to the same index, in-cache-line log is allowed (Line 36). This allows the bypass of in-cache-line logging (Line 2) and external logging (Line 16) to avoid unnecessary operations.

2. One of InCLL$_{1,2}$ is used, but the other one is not used: We can still use the second InCLL$_{1,2}$ structure to optimize performance. As one of the InCLL$_{1,2}$ is used from a previous modification within an epoch, the in-cache-line logging path (Line 2) is not usable. Also, logging path is not usable (Line 16) as in-cache-line log usage is allowed (Line 36). *InCLL* sets the value of the second InCLL$_{1,2}$ directly (Line 38), which avoids unnecessary external logging and allows the use of the second cache line.

3. One of the InCLL$_{1,2}$ or both of them are used and the modification is to an unlogged entry: In this case, the in-cache line log capacity is insufficient. That is why, *InCLL* logs the entire node.

3.3.3 External Logging Algorithm

We describe the cases for using the external logging. Internal nodes are useful for data structure traversal and benefits from caching. Increasing the size of the internal nodes by introducing an in-cache-line log can be detrimental to cache performance. Moreover, leaf nodes are modified much more frequently.

InCLL uses the external log for tracking internal nodes by introducing an epoch number (`nodeEpoch`) to each internal node similar to a leaf node. It checks if an internal node is logged within the epoch by checking the equality of `nodeEpoch` and `curEpoch`.

InCLL uses the external log for infrequent leaf node splits and merges as these operations touch most of the fields in a node and individually dealing with each modification would complicate the algorithm. *InCLL* also uses external log to handle insertions after deletions and updates to multiple entries within a cache line for the `vals` array.

The nodes that are accessed by *InCLL* are always locked. One benefit derived from locking is that the node is logged only once within an epoch. For external logging, we use thread-local logs and each log works independently, which avoids synchronization. As there are no dependencies in the log entries, each thread can run logging in parallel during the program execution. Likewise, during recovery, each thread can roll back independently. This is in stark contrast to traditional undo log rollback that requires the log to be applied in the reversed execution order, which limits the concurrency of the recovery procedure.

3.3.4 Recovery

InCLL consists of two types of undo logging mechanisms, in-cache-line logging and external logging. The two separate mechanisms require a separate approach to recovery. *InCLL*, first runs the external log roll-back mechanism. Afterward, *InCLL* lazily recovers the nodes that solely rely on in-cache-line log.

Figure 3.8 shows the high-level algorithm for the recovery. After a restart, *InCLL* inserts the epoch number that lead to a crash to *failed epoch* set. For correct recovery, the following relation holds `lastFailedEpoch` < `currExecEpoch` < `curEpoch`, where `lastFailedEpoch` is the epoch number of the last failed epoch, `currExecEpoch` denotes the first epoch in the current execution. *InCLL* sets the value of `currExecEpoch` to be the first epoch in the current execution after the restart and once recovery completes, the node continues operation from the `curEpoch`. This relation allows recovery to be applied first then allows in-cache-line logging.

The recovery algorithm as follows. First, *InCLL* rolls back the external logs (Line 4). Each thread iterates over the thread-local external log separately (Line 6) and copies the contents of each node from the log to the corresponding region in NVM (Line 7). Then, other components of the node such as an in-cache-line log and the locking mechanism is recovered (Line 8).

```
uint64_t currExecEpoch; // first epoch in the current execution
lock recoveryLocks[K];
// before first access to durable Masstree
void durableMasstree::recovery(){
# parallel for
    for each entry L in external log do: // for a node n
        memcpy(L→addr, L→content, L→size);
        n→nodeRecovery();
}
// before first access to a leaf node
void leafnode::lazyNodeRecovery(){
    if(unlikely(nodeEpoch<currExecEpoch)){
        int idx = hash(this);
        recoveryLocks[idx].acquire();
        if(nodeEpoch<currExecEpoch){
            nodeRecovery();
        }
        recoveryLocks[idx].release();
    }
}
void leafnode::nodeRecovery(){
    // InCLL_p
    if(failedEpoch.find(nodeEpoch))
        permutation = permutationInCLL;

    // InCLL_1
    uint64_t epoch = higher(nodeEpoch) | InCLL1.epoch;
    if(failedEpoch.find(epoch))
        vals[InCLL1.idx] = InCLL.ptr;
    // InCLL_2
    epoch = higher(nodeEpoch) | InCLL2.epoch;
    if(failedEpoch.find(epoch))
        vals[InCLL2.idx] = InCLL.ptr;

    // order writes
    atomic_thread_fence(memory_order_release);
    nodeFix();
}

void leafnode::nodeFix(){
    basenode::initlock(); // might be in bad state after crash
    InCLL[1,2].epoch = lower(currExecEpoch);

    // order writes
    atomic_thread_fence(memory_order_release);
    nodeEpoch = currExecEpoch;
}
```

Figure 3.8 – Durable Masstree recovery.

We describe the combined recovery after explaining the lazy recovery algorithm. External log rollback does not require synchronization and each thread can run independently until completion.

In-cache-line log structures are embedded into leaf nodes unlike the external log. Recovering in-cache-line log is more involved. Iterating over the entire data structure to reset the in-cache-line log is an option. However, the data structure traversal can be expensive depending on the Masstree size and can delay the restart. Moreoever, there can be nodes that are not logged by the external log, which are using an in-cache-line log.

InCLL uses lazy recovery (Line 11) to restore the nodes that use only the in-cache-line for a failed epoch. The lazy recovery rolls back the in cache line log state to the state at the start of the failed epoch.

The algorithm is as follows. When the program accesses the node after a restart, *InCLL* first checks if the `nodeEpoch` is from a previous execution (Line 12). If the node is from a previous execution, then the recovery algorithm brings the node to the beginning of the current execution (`currExecEpoch`) and recovers the node if possible.

One important issue is that multiple threads can be accessing the node concurrently. Locking is necessary to avoid data race conditions for the recovery code and to ensure that the recovery is correctly visible according to the Masstree design. However, *InCLL* cannot rely on the leaf node's lock. The reason is that the leaf node lock might be in a failed state after a failure. If *InCLL* tries to acquire a lock that is in the wrong state, the program can deadlock. The deadlock can also occur when a single thread tries to acquire the leaf node's lock.

InCLL uses its own separate locking mechanism for leaf nodes. It creates an array of transient locks residing in DRAM for recovery, hashes the address of the leaf node (Line 13), finds the appropriate bucket in the lock array, and acquires the lock (Line 14).

Once a node is locked, *InCLL* continues with the recovery code. *InCLL* checks the `nodeEpoch` again to ensure that the node is not recovered before by a concurrent thread (Line 15). *InCLL* then continues with the recovery code. *InCLL* checks if the last modifications to the node are from a failed epoch (Line 23). If the node is from the failed epoch, then *InCLL* recovers the `permutation` by using the value from `permutationInCLL` (Line 24). Afterward, *InCLL* checks the epoch numbers of $\text{InCLL}_{1,2}$ by combining the higher 16 bits of the `nodeEpoch` and the lower 16 bits of the `lowNodeEpoch` (Lines 27, 31). This is safe as the higher 16 bits are the same for the epoch that fails and the previous epoch. Otherwise, the node is logged (as shown in Line 5 figure 3.5).

For each $\text{InCLL}_{1,2}$, if the epoch number is from a failed epoch, then *InCLL* recovers the entry by restoring the pointer entry in the `vals` array using the `idx` field from the $\text{InCLL}_{1,2}$ and the `ptr` field (Lines 29, 33). Finally, *InCLL* completes node recovery by re-initializing the leaf node lock (Line 41) and fixing the epoch numbers (Lines 42, 46). By setting the `nodeEpoch` to the `curr-`

`ExecEpoch` (Line 46), the node does not need further recovery. The program can continue the execution normally and any subsequent use of an in-cache-line works as `currExecEpoch` < `curEpoch`.

If a node is recovered using the external log, then in-cache-line log might need to be properly recovered. The reason is that, during program execution, *InCLL* can log a node that uses an in-cache-line log. We do not roll back the in-cache-line log before using the external log for improved runtime performance. That is why, when using the external log, *InCLL* takes a snapshot of the entire log with in-cache-line log state. When the node is recovered from the external log, the recovery is not complete. The node is still not at the state at the beginning of the epoch as the in-cache-line log is not yet recovered. *InCLL* eagerly recovers the in-cache-line log during the recovery of the external log. The rest of the nodes that are not rolled back using an external log use lazy recovery.

Similar to the program execution algorithm, recovery does not require any write-back or fence instructions. If a power failure occurs before the recovery completes, recovery can be re-executed safely.

3.3.5 Persistent Memory Allocation

As we described in section 3.2.1, the value buffers are not stored inside a Masstree node. One important issue is that the allocation of the value buffers should be in NVM so they are available after a crash. Allocating and deallocating value buffers requires persistent memory allocation. For good runtime performance, it is essential to use high-performance persistent memory allocation and avoid costly write-back and fence instructions in the fast-path of the data structure.

Persistent memory allocation does not require any write-back or fence instructions. An allocator is essentially a data structure that keeps track of free chunks of memory. We only need to extend the in-cache-line logging approach to the data structure that records free chunks of memory and use periodic checkpointing to ensure persistence. *InCLL* can recover the state of the allocation data structure to its state at the beginning of an epoch.

The data structure that stores the free chunks of memory is a linked list. There are multiple linked lists for multiple size classes. During an object allocation, Masstree finds the correct size class and selects the appropriate linked list. Then Masstree pops an object from the free list and uses the memory region as a value buffer.

Deallocation is similar. Masstree reclaims the memory region by adding it to the appropriate linked list of free objects. The deallocated memory region becomes usable in the next epoch as Masstree uses epoch-based memory reclamation.

We observe that it is sufficient to use a single next pointer per free list node to implement the linked list of free objects. We use an in-cache-line structure, InCLL$_a$, to log the next pointer.

We first present a basic version of InCLL$_a$ to describe the high-level idea. Then we optimize InCLL$_a$ by compacting its size. For each object, the basic InCLL$_a$ has a header containing three fields that fit into a cache line. These fields are the next pointer, a backup location for the next pointer, and the epoch number.

The major drawback for *InCLL* persistent memory allocation is that the allocator requires header space for each object. The basic approach uses a 24-bytes header and can lead to 32-bytes space overhead due to 16 byte alignment constraints. We can compact the header using techniques introduced for InCLL$_{1,2}$.

Optimizing InCLL$_a$ Space Overhead

We compact the space occupied by the InCLL$_a$ and reduce the space overhead to 16 bytes for each object. The InCLL$_a$ fields are:

- `next`: The current next pointer.
- `nextInCLL`: The value of the `next` pointer at the beginning of the epoch.
- `allocEpoch`: 32-bit epoch number to keep track of the current epoch.

Both `next` and `nextInCLL` are pointer values. The upper 16 bits of both pointers provide 32 bits in total. 32 bits are sufficient for compacting the epoch number `allocEpoch`. We break the 32 bit epoch number `allocEpoch` into two 16 bit numbers and place the most significant 16 bits in the `next` and the least significant bits in the `nextInCLL`.

One problem with epoch number compaction is that we update a single word atomically. However, the epoch number is spread across two words. The update to the epoch number can see a partial write state after a crash. After recovery, *InCLL* can see a state where only half of the epoch number is updated. Seeing a partial state can lead to incorrect recovery.

We encode a small counter to avoid torn writes using the lower unused bits of the pointers. Specifically, *InCLL* uses the four least significant bits of both `next` and `nextInCLL` to implement a counter. *InCLL* increments the small counter when the program modifies the pointer for the first time in the epoch.

If both counters are the same, *InCLL* assumes that the pointer modification is complete and `nextInCLL` contains the backup value for `next`. *InCLL* constructs the correct epoch number by combining the 16 most significant bits of `next` with the most significant 16 bits of `next-InCLL` and checks the epoch number. If the epoch number is from a failed epoch, *InCLL* recovers the `next` using the value of `nextInCLL`. *InCLL* creates the epoch number using the 16 most significant bits of `next` and `nextInCLL` to form an epoch number. If both counters are different, then the difference between the counter values implies that a crash happened before during the pointer modification. *InCLL* recovers the value of the `next` field from the `nextInCLL` field.

3.4 Evaluation

3.4.1 Machine Configuration

We performed the experiments on an Intel server with Intel Optane Persistent Memory. The server had two sockets with Intel Xeon Platinum 8276L processors running at 2.20GHz each containing 28 physical cores (56 hyperthreads). It had 39.8MB L3 cache, 375GB DRAM, and 1.5TB NVM. The machine ran Fedora 30 with Linux Kernel version 5.0.9. We used gcc/g++ version 9.3.1 and make version 4.2.1 for compilation. We set the CPU governor to "performance", disabled Turbo Boost and huge pages, and pinned each worker thread to a core. We used Optane in App-Direct Mode with 6-way interleaved NVDIMMs with DAX enabled *ext4* file system as described in section 2.1.4.

3.4.2 Previous and Current Configuration

There are several differences between this evaluation and previous evaluations. This work was published in ASPLOS 2019 [38] based on software simulation. Research predating Optane relied on a variety of simulation strategies such as emulation [52, 53, 173], treating DRAM as Optane [23, 32, 51, 128], and adding artificial latencies to fences [40, 68, 185]. In the ASPLOS paper, we considered the effects of adding artifical latencies. The `clflushopt` instruction is asynchronous (Section 2.1.2) and the write-back instructions complete at `sfence` point. For this reason, the simulation approach introduced extra delay after each `sfence` instruction to simulate the effects of write-back and fence instructions, for which *InCLL* is optimized.

The artifical delay models increased the cost of flushing cache lines to NVM. The simulation methodology, however, did not model the speed of other memory accesses such as reads and writes. It also failed to address the low read/write bandwidth of Optane, especially under heavy concurrent accesses.

Another important factor was the cache sizes. Overall, the cache sizes in the current machine were larger than those in the ASPLOS paper. The L3 cache size for ASPLOS was 19.25MB and the current machine had 39.8MB (Section 3.4.1). The presentation made for PIRL 2020 [9] used Optane evaluation, however it had 32MB L3 cache. Large caches can lead to more time spent in `wbinvd`. Furthermore, filling a larger cache can take more time.

For the evaluation, in this dissertation, we did not use software simulation. Since we had access to Optane, we evaluated directly on Optane. We also treated DRAM as Optane and provided measurements for this case as well.

3.4.3 Baseline Configuration

We configured Masstree to use the *jemalloc* allocator. We present the results for the unmodified Masstree (*Mt-Unmod* in figure 3.9). We used an optimized version of Masstree as a baseline, which ran $1.0-1.3x$ faster than the original Masstree. In this version, we modified the Masstree allocator to place the node and value buffer allocation on a memory-mapped file using Masstree's pool allocator instead of relying on the jemalloc allocator. We left the rest of the structures in the program intact and transient. The NVM version of baseline Masstree (*Baseline-NVM*) is not crash-consistent and can be in an inconsistent state after recovery.

For the DRAM configuration (*Baseline-DRAM*), Masstree uses files stored in `/dev/shm`. The DRAM configuration did not use two separate copies, as this file system is based on *tempfs*. For the Optane configuration (*Baseline-NVM*), Masstree used files in a device backed by Optane in App-Direct Mode. We used DAX-enabled *ext4* file system for managing NVM. We did not introduce artificial delays. We populated page table entries for the memory-mapped files by writing idempotently into the address space initially. We pinned each Masstree thread to a separate core.

3.4.4 InCLL Configuration

We built *InCLL* on top of baseline Masstree (Section 3.4.3). DRAM and Optane configurations are similar to baseline Masstree (Section 3.4.3). We allocated the Masstree data structure, the value buffers, and the external logs in DRAM for *InCLL-DRAM* and in Optane for *InCLL-NVM*.

Each leaf node had up to 14 key-value pairs. It is less than non-*InCLL* version because of the in-cache-line log. The combination of the mechanisms allowed Masstree to be durable. *InCLL-NVM* configuration is completely durable, however, *InCLL-DRAM* would require battery-backed DRAM.

3.4.5 Logging Configuration

Logging is another durable version of Masstree. *Logging* is similar to the version of *InCLL* without the in-cache-line log, but with external logs. We allocated the Masstree data structure, the value buffers, and the external logs in DRAM for *Logging-DRAM* and in Optane for *Logging-NVM*.

Each leaf node had up to 15 key-value pairs as in the original. *Logging* implemented fine-grained checkpointing, external logging, and persistent memory allocation as described in section 3.3. We only introduced a `nodeEpoch` field for the leaf nodes similar to internal node logging. Persistent memory allocation still used an in-cache-line log (InCLL_a). The *Logging* design is useful to show that *InCLL* approach improves runtime performance for write-heavy workloads, as we demonstrate in section 3.4.7.

3.4.6 YCSB Configuration

We used the YCSB [42], a popular NoSQL benchmark, to evaluate Masstree performance. Unless otherwise noted, we initialized Masstree with 24 million key-value pairs and run the experiments using 6 threads. Keys were 8 bytes while value buffers were 24 bytes. The driver threads generated the workload on the same machine to avoid network interference. We used three different workloads:

YCSB A Write heavy, 50% puts, 50% gets

YCSB B Read heavy, 5% puts, 95% gets

YCSB C Read only, 100% gets

We used two types of memory access patterns, which were *uniform* access pattern and *Zipfian* access pattern. For uniform access, the workload driver generates keys uniformly, at random between 0 and 24M. For Zipfian access, the workload driver generates keys using a Zipfian distribution with a skew parameter of 0.99. The workload driver scrambled the keys by hashing their values to avoid frequent keys being in close proximity. Each workload thread ran 1 million operations. We reported throughput (operations per second) for Masstree, where higher is better. We ran the experiments 5 times and provided the average result. The standard deviation for the experiments ranged from 0.01% to 0.33%.

3.4.7 Measurements

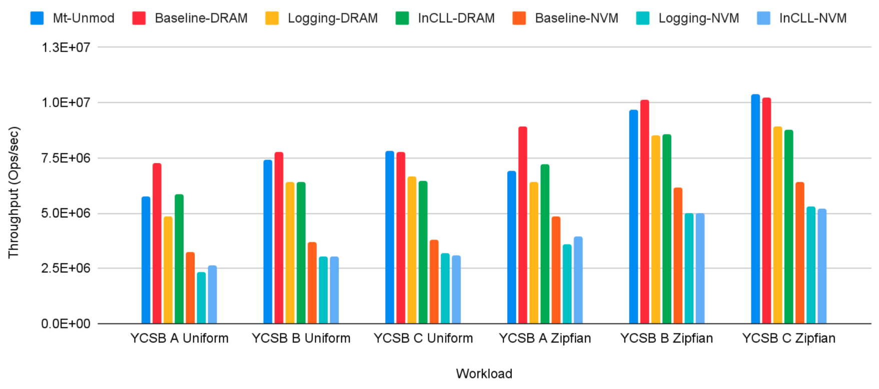


Figure 3.9 – Throughput of unmodified Masstree, baseline Masstree and *InCLL* with DRAM and Optane evaluations.

Figure 3.9 shows the main workload for baseline Masstree, *InCLL*, and *Logging* running on DRAM and Optane. Compared to *Baseline-DRAM*, the slowdown of *InCLL-DRAM* is 14.4 – 19.4% (YCSB A, B, C - Uniform, Zipfian). Specifically, *InCLL-DRAM* has 19.4% overhead for Uniform A, 19.2% for Zipfian A, 17.5% for Uniform B, 15.8% for Zipfian B, 16.5% for Uniform

C, and 14.4% for Zipfian C. These results are similar to the ASPLOS evaluation. The main difference is due to the cost of flushing larger caches.

Compared to *Baseline-NVM*, the slowdown of *InCLL-NVM* is between 18.6 – 19.0%, which is similar to the difference between *Baseline-DRAM* and *InCLL-DRAM*. Read-only cases have slightly more overhead due to cache misses to Optane after a `wbinvd`. This shows the effectiveness of the *InCLL* algorithm, which minimizes write-back and fence instructions. However, the full overhead of *InCLL* is exposed by the comparison between the baseline Masstree (*Baseline-DRAM*) and fully durable Masstree (*InCLL-NVM*). In this, *InCLL-NVM* has a slowdown of 49.4 – 63.9%. The difference is attributed to using Optane to hold the Masstree data structure. One way to see this is to compare baseline Masstree (*Baseline-DRAM*) with *Baseline-NVM*, which leads to a slowdown of 38.7 – 55.6%.

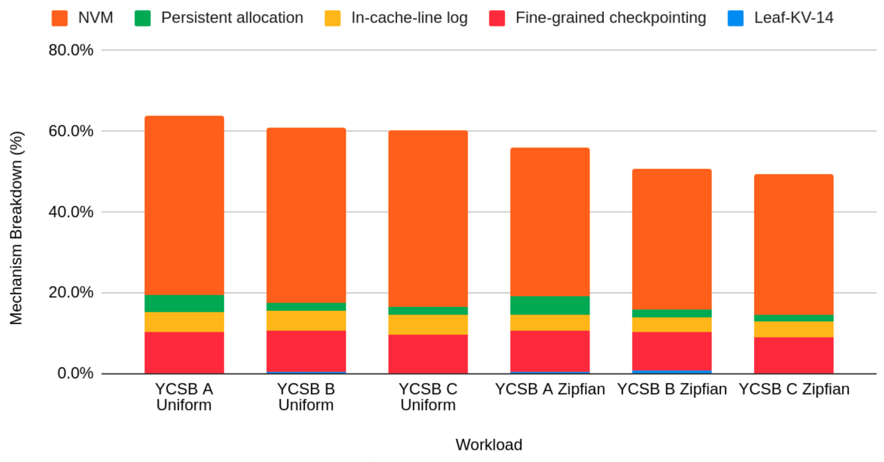


Figure 3.10 – Mechanism breakdown of *InCLL* features.

Figure 3.10 breaks out the cost of using *InCLL* mechanisms (lower is better). We measure the throughputs for reducing key-value pairs (Leaf-KV-14), fine-grained checkpointing, in-cache-line logging, persistent allocation, and *InCLL-NVM* successively and calculate the slowdown from the baseline for each configuration. We report the percentage differences for the slowdowns. The majority of the cost is due to using NVM (34.9 – 44.4%). *InCLL* allocated the entire Masstree data structure, value buffers, and external log in Optane, which has a significant access penalty.

Globally flushing the caches imposes the next largest cost (9.1 – 10.3%), due to the larger caches and cache invalidation. Running the `wbinvd` instruction takes approximately 4.3ms per epoch (6.7% overhead in a 64ms epoch) for both DRAM and Optane cases.

We measured the effect of flushes independently by running `wbinvd` (no *InCLL*) on top of baseline Masstree. Cache flushes increase the cost of cache misses from 10 – 20% and the cost of handling TLB misses up to 30% (VTune Profiler). This introduces 11 – 12% overhead in the DRAM case and 10 – 14% in the Optane case.

The persistent memory allocator changes the buffer layout and leads to additional slowdown in write-heavy cases of 4.2 – 4.5%, as allocation is heavily used. The slowdown due to reducing the number of key-value pairs in leaf nodes (*Leaf–KV–14*) is minimal (0.0 – 0.8%).

The cost of using in-cache-line logging combined with external logging is between 3.8 – 5.0%, which is low. The external log contains 181KB for Uniform A and 39KB for Zipfian A of data for *InCLL-DRAM*, and 44KB for Uniform A and 16KB for Zipfian A of data for *InCLL-NVM*. The main reason for fewer entries for the Optane case is that there are fewer operations within an epoch.

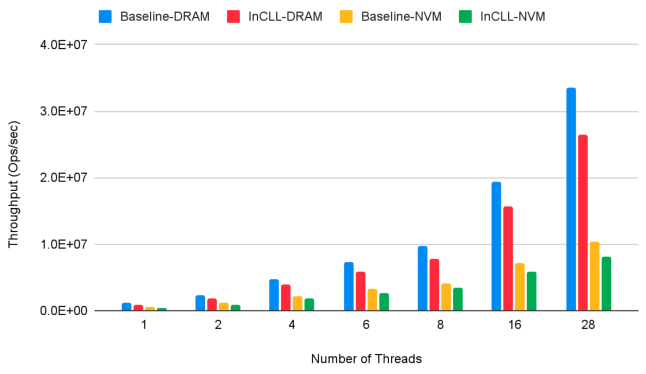
Figure 3.9 reports the throughput results for *Logging*. The slowdown of *Logging-DRAM* over *Baseline-DRAM* is between 12.7 – 33.6%. Specifically, *Logging-DRAM* suffers from external log's write-backs and fence instructions for fast-path operations (28.0 – 33.6% for YCSB A). However, *Logging-DRAM* performs better for the read-only case (12.7 – 14.3% for YCSB C) compared to *InCLL-DRAM*. This is expected as the layout of a leaf node is not modified except introducing `nodeEpoch` and the external log is not needed for the read-only cases.

Comparing *Logging-NVM* with *Baseline-DRAM* leads to a slowdown of 48.2 – 68.0%, where the bandwidth limitation of Optane dominates most of the cost. The external log contains 7.9MB for Uniform A and 5.1MB for Zipfian A of data for *Logging-DRAM*, and 3.8MB for Uniform A and 2.9MB for Zipfian A of data for *Logging-NVM*, which is at least an order of magnitude greater than *InCLL*. While both *Logging-NVM* and *InCLL-NVM* are slow due to using Optane, the performance difference is not as significant as in the case of *Logging-DRAM* and *InCLL-DRAM*.

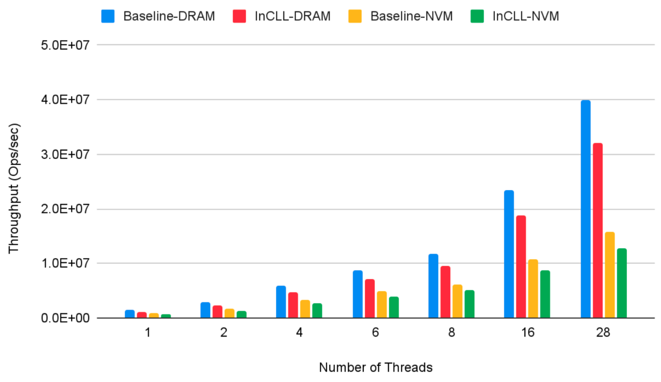
Figure 3.11 shows the scalability limitations of Optane for both baseline Masstree and *InCLL*. We vary the number of threads from 1 to 28 and report the results for the write-intensive YCSB A. Comparing with *Baseline-DRAM*, *InCLL-DRAM* slowdown is 19.0 – 21.5% for the Uniform A case and 18.8 – 20.3% for the Zipfian A case. For the DRAM case, *InCLL-DRAM* scalability is not affected significantly by increasing the number of concurrent worker threads. On the other hand, the Optane case is different. Comparing with *Baseline-DRAM*, *InCLL-NVM* consistently slows down as we increase the number of threads. The overhead is 61.9 – 75.5% for the uniform case and 55.0 – 68.1% for the Zipfian case. The main performance bottleneck is due to Optane.

If we normalize the performance for each Masstree version with respect to a single thread performance, the limitations of Optane become clear. For 28 threads normalized to a single thread, the performance of *Baseline-DRAM* is 26.7 – 28.2x and *InCLL-DRAM* is 26.8 – 28.4x. *InCLL* is a scalable approach. However, overall, the scalability of both baseline Masstree and *InCLL* is hampered by Optane. Running with 28 threads leads to a speedup of 17.2 – 17.8x for *Baseline-NVM* and 18.1 – 18.9x for *InCLL-NVM*.

Figure 3.12 shows the effect of the initial size of the tree on YCSB performance. We again use YCSB A workload and vary the number of initial key-value pair entries from 6M to 96M. The dominating trend for performance is tree traversal, which reduces Masstree throughput. Compared to *Baseline-DRAM*, the slowdown of *InCLL-NVM* is between 61 – 64% for Uniform



(a) Uniform A workload.



(b) Zipfian A workload.

Figure 3.11 – Throughput of baseline Masstree and *InCLL* with DRAM and Optane evaluations for different number of threads.

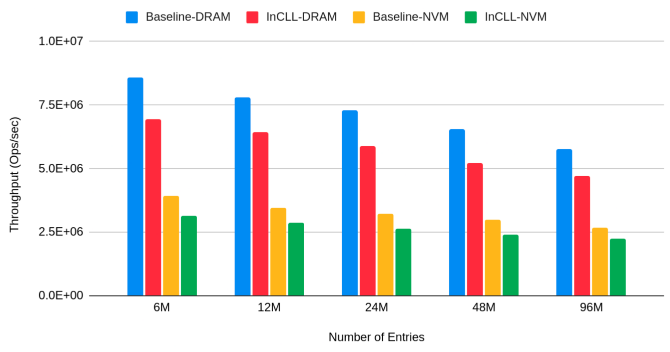
A and 53 – 54% for Zipfian A.

Figure 3.13 shows the effect of changing the checkpointing interval from 16ms to 256ms. The overheads for the baseline Masstree are mostly consistent (<0.8%). For *InCLL*, flushing the caches more frequently has a negative effect on performance. If we compare the same *InCLL* version with 16ms intervals and 256ms intervals, the performance for the *InCLL* version with 256ms intervals is always faster. *InCLL-DRAM* is 1.4x faster and *InCLL-NVM* is 1.3x faster.

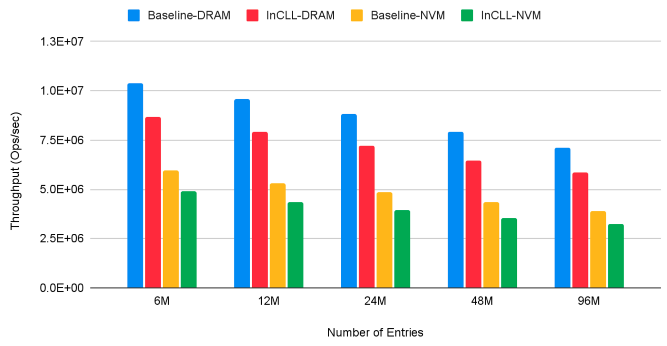
Correctness and Recovery

We use testing to check the correctness of *InCLL*. We crashed *InCLL* at random points, launched a new process and checked that the system state matches with the state at the beginning of the failed epoch. We used many unit tests to check that in-cache-line log and external log recovery can correctly restore the state.

We measured the time to recover the log for *InCLL-NVM* with a 24M initial tree size running YCSB A Uniform workload. We crashed at the end of the 5^{th} epoch before starting a new epoch.



(a) Uniform A workload.



(b) Zipfian A workload.

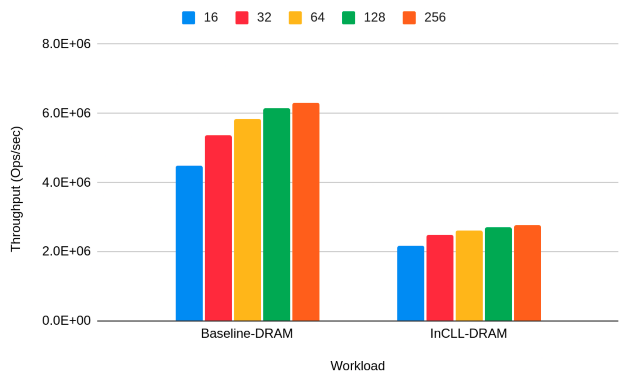
Figure 3.12 – Throughput of baseline Masstree and *InCLL* with DRAM and Optane evaluations for different initial tree sizes.

For each thread, the log contained 44KB of data on average and approximately took 0.18ms to process the log, which included in-cache-line rollback of the logged entry. The program performance returns to 2.6MOps/sec after 4 epochs of 64ms.

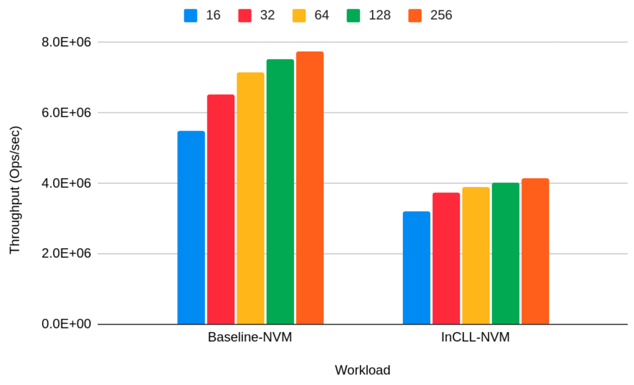
3.5 Summary

The chapter presented *InCLL* as a technique for building durable data structures. Overall, *InCLL* places the entire data structure solely in NVM and periodically flushes modifications from the caches to NVM. *InCLL* uses in-cache-line logging to minimize write back and fence operations for fast-path data structure modifications.

We modified Masstree with *InCLL* technique and made Masstree durable. We persisted the linked list allocator and provided persistent allocation. The new evaluation on DRAM is similar to the previously published results in ASPLOS and showed that *InCLL* can be feasible using battery-backed DRAM. We should note that the design of *InCLL* was specific to the simulation strategy that we used to evaluate it. The simulation strategy introduced delays after each `sfence` instruction and measured the costs related to write-back instructions and fences similar to the previously published evaluation methods.



(a) Uniform A workload.



(b) Zipfian A workload.

Figure 3.13 – Throughput of baseline Masstree and *InCLL* with DRAM and Optane evaluations for different number of checkpointing intervals.

As the saying goes, we got what we measured by solely evaluating the effects of write-back instructions. On the other hand, Optane in its current form is still slower. Optane has higher latency and lower bandwidth compared to DRAM. The simulation strategy did not take costs incurred by memory accesses and Optane characteristics into account. Not accounting for performance bottlenecks from Optane leads to unsimulated overheads of up to 64% due to additional memory access overheads. We acknowledge that this is a shortcoming of *InCLL* design. We propose a new checkpointing design, *CpNvm*, to mitigate the costs related to Optane in the next chapter.

4 NVM Checkpointing with *CpNvm*

Programs must be written for people to read, and only incidentally for machines to execute.
— Abelson and Sussman

We expect Intel Optane to be used in a data center environment and be widely available as durable memory as we explained in chapter 1. This implies that, *InCLL*, which is described in chapter 3, will likely be using Optane as an NVM device. While *InCLL-DRAM* incurs overheads less than 20%, the overheads for Optane vary between 49 – 64%. This cost is very significant and the performance mismatch between the current Optane and DRAM devices is too large to be practical.

DRAM is faster than Optane and has a higher read/write bandwidth. Exploiting the performance of DRAM could surely improve performance. However, *InCLL* is already a complicated design. There is no automated technique to apply *InCLL* to different data structures and it is rather a tool for an expert library developer. Due to *InCLL*'s use of an undo logging recovery scheme, it is difficult to move the data structure from NVM to DRAM. Furthermore, Intel does not currently provide a hardware mechanism to cache persistent modifications in DRAM.

In this chapter, to address these problems, we introduce *CpNvm*, a runtime system for periodically checkpointing data structures. *CpNvm* uses write-combining in DRAM to buffer modifications and periodically (on the order of milliseconds) propagates these modifications to an NVM copy of the data structure using redo logging. Heavily read or written data in DRAM are accessed at a fraction of the cost. We implement *CpNvm* as an easy-to-use application-level checkpointing API.

This chapter is organized into the following sections:

- Section 4.1 motivates *CpNvm* by identifying the challenges related to Optane.
- Section 4.2 describes the high-level design of *CpNvm*.
- Section 4.3 explains the implementation details and the checkpointing interface.

- Section 4.4 evaluates *CpNvm* and demonstrates its low overhead for both the Masstree data structure and the Memcached program.

- Section 4.5 compares *CpNvm* to related work.

4.1 Introduction

Byte-level addressability does not make NVM a substitute for DRAM. Optane is slower than DRAM as it has lower bandwidth. Yang et al. [188] reports that NVM read latency is $2-3x$ that of DRAM, and writes can occasionally execute $100x$ slower (possibly due to page remapping for wear leveling). Non-interleaved NVM has a peak read bandwidth of approximately 6% of DRAM and peak write bandwidth of 3% of DRAM. NVM's bandwidth degrades severely as the number of threads increases. Furthermore, NVM has other unfortunate characteristics, which further lower the bandwidth. Mixed read/write workloads to Optane can perform poorly [188]. Sequential access to NVM can be $4x$ faster than random accesses to NVM [91] (due to NVM's internal caches). A high-performance crash-consistency mechanism needs to account for these characteristics.

As chapter 3 demonstrates, a program can incur a significant overhead by using Optane. The Masstree data structure has a $39-56\%$ slowdown when using Optane compared to DRAM as shown in figure 4.1. This large overhead does not include any crash-consistency mechanism nor a persistent memory allocator. After a restart, the Masstree data structure would be in an inconsistent state, and the allocator could have dangling pointers and memory leaks.

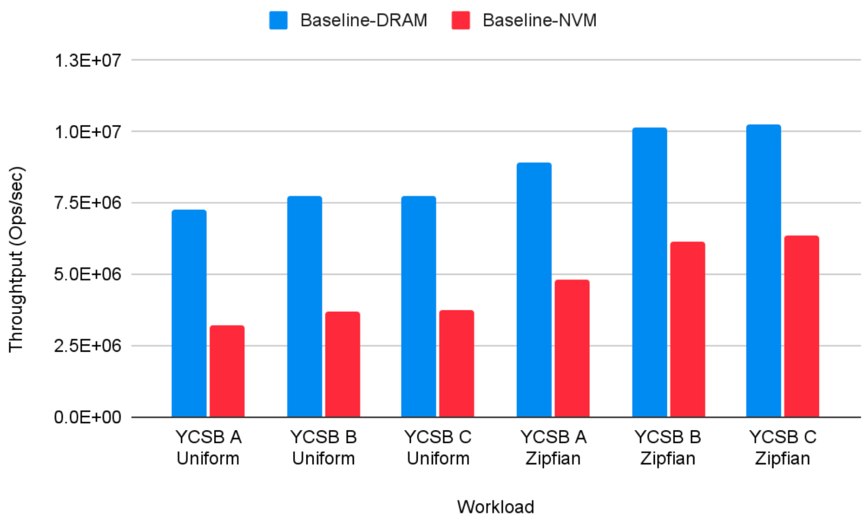


Figure 4.1 – Throughput of Masstree for DRAM and Optane evaluations.

The majority of the overhead is due to memory accesses. Furthermore, carefully examining the overheads in Masstree identifies the locking mechanism, which is a major source of overhead. Masstree uses a spinlock to lock a Masstree node. If the program allocates the node in NVM, then the spin operation will cause NVM memory accesses.

Recipe [108] avoids implementing the lock in the Masstree node and instead uses the standard `mutex` locks residing in DRAM. Recipe revamps the Masstree data structure design completely to optimize performance for NVM. After a restart, Recipe reinitializes the locks for each node. Our measurements indicate that even an optimized version of Masstree still suffers from the cost of using Optane. Recipe suffers 38 – 51% slowdown because of Optane, not including a persistent memory allocator.

We observe that if we move the runtime data structure from Optane to DRAM, we can improve performance. There are several design alternatives that we can follow. We can implement a hybrid model [81, 134, 181, 186, 189] where a part of the data structure is in DRAM and another part is in NVM. We can make value buffers durable with its key and keep the rest of the Masstree data structure as volatile, effectively eliminating in-cache-line logging. Other schemes, such as keeping only internal nodes volatile, are also possible. However, these complicate Masstree recovery and can introduce a significant overhead after a restart due to reconstructing the entire traversal data structure.

InCLL also has other problems. It is not a general technique as the implementation requires an expert library programmer. It requires modifications to the internal layouts of data structures. It also requires the execution of a privileged instruction (`wbinvd`) to flush the caches to NVM, which can create security issues in multi-tenant systems.

Our goal is to implement a general checkpointing solution that works for existing programs and can support relaxed crash-consistency models while avoiding the cost of Optane. Our approach stands in stark contrast with the previous work [32, 37, 108, 160], which used a write-through mechanism to NVM to implement a strong consistency model such as atomic transactions. The consequence of write-through is that only processor caches are used to reduce memory latency. Bypassing DRAM, which offers high capacity and performance, can severely hinder program performance. Newer systems [112, 185] use a write-back model, in which a data structure resides in DRAM and a redo log propagates modifications to a copy in NVM. The main issue with these systems is that the (simulated) performance is too slow for general use. For example, DudeTM [112] reports 7.4 – 24.6% overhead on top of TinySTM's undocumented cost, while PMThreads [185] report slowdowns between 43 – 122%.

CpNvm is a runtime library that provides checkpointing functionality to store a recoverable copy of the program data in NVM. *CpNvm* breaks the program execution into epochs and periodically persists the modifications to NVM. *CpNvm* uses DRAM to buffer modifications during an epoch and allow fast memory accesses. After a restart, *CpNvm* can restore program data stored in the durable heap to its state at the last committed epoch.

CpNvm provides the checkpointing functionality with a general, easy-to-use API, which is inspired by application-level checkpointing [26, 138]. The programmer inserts calls to *CpNvm* API that can provide durability for the program data structures periodically.

A program using the *CpNvm* library has four distinct stages: *execution*, *checkpointing*, *background replay*, and *recovery*. In execution, *CpNvm* keeps track of modified memory regions. The tracking mechanism requires additional calls by the programmer.

The second stage is checkpointing. *CpNvm* persists the modifications to memory regions by saving the changes in redo logs in NVM. Unlike *InCLL*, *CpNvm* does not flush the cache hierarchy, as all the modifications are on DRAM data structures.

The third stage is background replay, which compacts the log by applying modifications to an NVM copy of the data structure. Background replay runs concurrently with program execution.

The fourth stage is recovery, which runs after a failure. *CpNvm* restarts the program, finishes log replay and lazily faults in data pages from the shadow copy in NVM.

CpNvm performs well with low overhead (< 15%) for Masstree [117] and (< 6%) for Memcached [61]. For read-only workloads *CpNvm* incurs almost no overhead for either Masstree or Memcached. These results are achieved by 25 *CpNvm* library calls in 27K LoC of Masstree and 29 LoC modifications in 4.5K LoC of Memcached.

Checkpointing is well-known technique with a lot of past work [1, 21, 29, 55, 99, 111, 130, 139, 140, 153, 161, 171, 176]. The new result from this work is that existing and new techniques can be combined to form a usable application-level checkpointing API that can be implemented with a low overhead for building durable data structures in NVM with performance near DRAM execution speed.

The main contributions of this work are:

- The design of a new checkpointing library allowing periodic checkpointing intervals on the order of milliseconds by relying on redo logging and shadow memory.
- A new programming model (API) for checkpointing for NVM based on identifying writes to persistent data structures and program idleness.
- The implementation of *CpNvm*, a new C/C++ library that implements periodic checkpointing as an application-level checkpointing library at low cost with minimal developer effort for NVM.
- Evaluation of *CpNvm* using Masstree and Memcached on Intel's Optane DC Persistent Memory, demonstrating its low run-time overhead.

4.2 *CpNvm* Design

As we explained in section 4.1, the goal of *CpNvm* is to provide an efficient checkpointing mechanism by minimizing unnecessary operations on NVM. The main design constraint for

CpNvm is the low bandwidth and high latency of NVM, compared to DRAM. For this reason, we minimize operations in NVM by using DRAM as a last-layer cache.

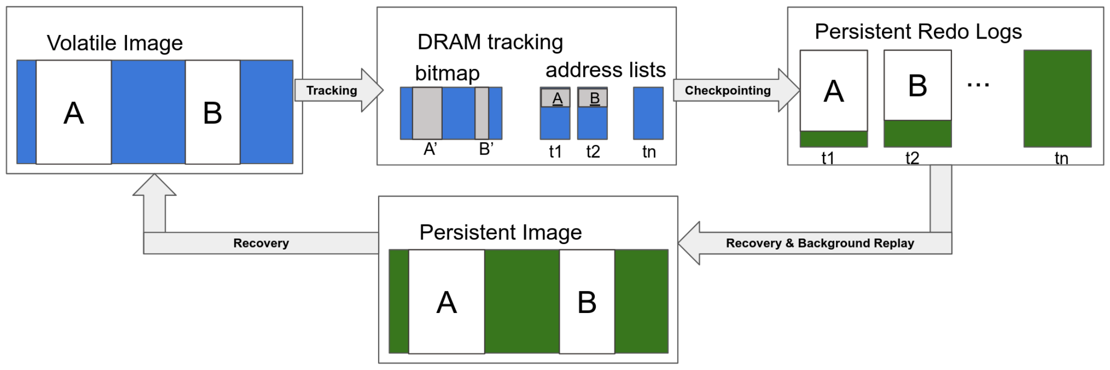


Figure 4.2 – *CpNvm* design.

Figure 4.2 depicts the overall design for *CpNvm*. There are two copies of a data structure: the *volatile image* in DRAM and the *persistent image* in NVM. The *volatile image* resides in a memory-mapped file in DRAM. The second copy, *persistent image*, is a memory-mapped file in NVM. The execution operates on *volatile image* and cannot directly modify the *persistent image*. *CpNvm* coalesces multiple writes to a cache line to avoid unnecessary data transfer to slow, possibly wear-sensitive NVM. We describe the memory capacity of a system in detail in section 4.3.7.

In execution, *CpNvm* tracks the *volatile image*. The *CpNvm* design requires programmers to explicitly identify modifications. The programmer inserts calls to the *CpNvm* library to track memory regions that are modified within an epoch. Once a memory region is tracked, tracking the same region within an epoch is not necessary. *CpNvm* uses a lightweight tracking mechanism with little overhead (3.4% worst-case - Figure 4.8) that is negligible for read-only workloads. *CpNvm* can track memory writes at word (8-bytes) granularity or cache-line (64-bytes) granularity.

There are several ways to minimize programmer effort for implementing a tracking mechanism. It is possible to use compiler-techniques [21, 44, 57, 185] to minimize the programmer effort. The programs that we examine do not require many changes, and most modifications are local and in close proximity to each other. For this reason, we did not explore compiler-techniques. Another alternative is to rely on VM page protection to detect memory writes [112, 185]. Page protection allows the system to detect the first write to a page. The system then assumes all data on the page is modified, which causes unnecessary logging. PMThreads [185] avoids copying unmodified locations by performing a byte-by-byte comparison between the DRAM page and its NVM copy. However, the comparison is expensive and throttled by the low NVM bandwidth.

During checkpointing, *CpNvm* stores a snapshot of the modified memory regions in *persistent logs*. We prefer an append-only log, because of NVM's higher sequential access pattern performance [91]. *CpNvm* uses intervals on the order of milliseconds for checkpointing.

There are many issues in using a redo log design. One major concern is that the redo log can grow unbounded. It is necessary to compact the log. It is possible to use standard compression [140] techniques, which trade-off the time spent in compression and log size reduction. Moreover, traditional redo logging redirects reads and writes to the redo log for the latest values. This redirection can require significant modifications to a codebase. Finally, after a restart, the entire redo log must be replayed to regenerate the data structure, which can take a significant amount of time depending on the log size.

To simplify the API, avoid read/write redirection, reduce time spent in restarting, and compact the log size, we use shadow memory from DudeTM [112]. The *persistent image* is a loosely synchronized copy of the *volatile image* and *persistent logs* act as an intermediate durable format for the *persistent image*.

CpNvm uses background replay threads concurrent with execution to apply the log to the *persistent image*. *CpNvm* discards log entries for an epoch once the snapshots are replayed, compacting the log.

Recovery is similar. Recovery code in *CpNvm* first finishes replaying the log to update the *persistent image* to the program state at the last committed epoch. Since the *volatile image* is transient, after a restart, it is entirely lost. *CpNvm* faults in pages of the *volatile image* from the updated copy in the *persistent image* lazily. It can also eagerly load all the pages.

4.2.1 Crash-Recovery Model

In this section, we discuss the implications of breaking the program execution into epochs and using checkpointing for crash consistency. *CpNvm* breaks the program into epochs similar to *InCLL* and takes a checkpoint at well-defined intervals. The duration of the checkpointing interval creates a window of vulnerability. This is not a semantic problem, but a practical problem depending on the program's specification and needs. Increasing the frequency of checkpoints can reduce the size of the window of vulnerability, at the cost of increasing additional overhead costs.

If the system cannot tolerate data loss, it is possible to use another mechanism in addition to *CpNvm*. Operational logging [122] can be useful for recording the operations within an epoch. After a restart and completing *CpNvm*'s log recovery, all the operations committed within the failed epoch can be re-executed using the operational log.

4.3 *CpNvm* Implementation

In this section, we describe the implementation of *CpNvm* checkpointing library. *CpNvm* is a C/C++ header-only library that allows programs to periodically persist their data structures.

Figure 4.3 depicts *CpNvm* API. The programmer needs to insert the calls to *CpNvm* API to make selected program data structures durable.

```
void cpnvmInit(opt_t *options);
void cpnvmThreadInit();
void *cpnvmRoot(char *name, void *addr);
void *cpnvmAlloc(size_t size, bool mark=true);
void cpnvmFree(void *ptr, size_t size);
void cpnvmMark(void *ptr, size_t size);
void cpnvmCheckpoint();
bool cpnvmInRecovery();
```

Figure 4.3 – *CpNvm* API

The API operates follows:

1. Initialize *CpNvm* before accessing durable data structures (`cpnvmInit`).
2. Initialize *CpNvm* for each worker thread (`cpnvmThreadInit`).
3. Use *CpNvm* allocation functions to obtain and release memory for durable program data structures (`cpnvmAlloc`, `cpnvmFree`).
4. Call the marking function to track modifications to memory regions (`cpnvmMark`).
5. Call the checkpointing function to initiate checkpointing (`cpnvmCheckpoint`).

That is it.

4.3.1 Execution

Before starting execution, a program calls `cpnvmInit` to initialize *CpNvm* data structures (e.g., *persistent logs*, *address lists*, *bitmap*) and background threads. `cpnvmInit` takes an *options* argument, which configures the *CpNvm* library. The options determine the interval between checkpoints in milliseconds (e.g., $64ms$), the granularity of tracking (e.g., word or cache-line granularity), allocator information, recovery information (e.g., lazy or eager), and the number of background threads. When the program creates a thread, it must call `cpnvmThreadInit` to initialize thread specific data structures (e.g., *persistent logs* and *address lists*). *CpNvm* maintains these data structures: *volatile image*, *persistent image*, and *persistent logs*. The *volatile image* files are in DRAM (using *tempfs*), while the *persistent image* and *persistent logs* files are stored in Optane (using DAX in *ext4*).

In *CpNvm*, a *root* pointer keeps an address for a known location in the *volatile image* and semantically is the root of the data structure. The reason is that, the program needs to access the same location after a restart and execution operates solely on the *volatile image*. `cpnvm-Root` takes two arguments, a unique name for the root pointer and the memory location. If the location is not specified, `cpnvmRoot` returns the address depending on the name if it exists. If the location is specified, `cpnvmRoot` binds the address to the name durably and the location is accessible given the name after a restart.

CpNvm requires the volatile and persistent images to be mapped to the same virtual addresses after a restart. One possible technique to allow different mappings after a restart would be to use pointer swizzling [39], fat pointers [141], relative pointers, or OS support [24].

CpNvm handles the recovery of the program data structures. However, after a restart, a program might need further recovery, such as re-initializing locks and files [39]. We provide `cpnvmInRecovery` to allow a programmer to implement recovery code specific to the program, if *CpNvm* runs recovery after a restart. `cpnvmInRecovery` always returns `true` if the persistent data structures (*persistent logs*, *persistent image*) exist.

The user of *CpNvm* library can erase the persistent state by deleting all the memory-mapped files in NVM, which allows the program to start afresh.

Allocation

We use Masstree's pool allocator. We discussed Masstree's allocator in detail in section 3.3.5. The main benefit of the Masstree pool allocation is that its data structures reside in the memory-mapped file, instead of an external location, making it straightforward to make the allocator durable.

We modify the allocator to use *CpNvm*'s tracking and checkpointing mechanisms to persist the allocator structures, similar to any other data structure. The Masstree allocator has a pointer to each free list for an unallocated block. *CpNvm* tracks the pointers if modified. *CpNvm* can allocate (`cpnvmAlloc`) and free (`cpnvmFree`) 8-byte-aligned blocks of uninitialized memory in DRAM, backed by NVM. Programs frequently modify the allocated memory regions after allocation. That is why, to minimize programmer effort, *CpNvm* marks the memory region by default before returning the allocation to the program. It is possible to disable the marking by setting the `mark` argument to `false`. At deallocation, *CpNvm* only needs to mark the `next` pointer.

4.3.2 Tracking

CpNvm needs to track modifications to data structures for checkpointing. *CpNvm* implements a lightweight tracking mechanism. *CpNvm* provides two granularities for tracking, word (8-byte) and cache-line (64-byte).

Figure 4.4 shows the high-level algorithm for tracking, which requires the use of the the `cpnvm-`

```
class bitmap_t; // for globally shared bitmap
class addrlist_t; // for thread−local address list
class plog_t; // for thread−local persistent logs

bool setBits(bitmap_t∗,void∗,size_t); // sets bits in the bitmap
size_t resetBits(bitmap_t∗, void∗); // resets adjacent bits in the bitmap
void addToList(addrlist_t∗,void∗); // appends the location to address list
void logModification(plog_t∗, void∗, size_t); // appends the modification to a redo log
void finalize(plog_t∗,bool discard=false); // make durable, discard epoch entries if true
void threadBarrier(); // spinlock thread barrier

void cpnvmMark(void ∗addr, size_t size) {
  bool isSet = setBits(bitmap, addr, size);
  if (isSet) {
    addToList(addrlist, addr);
  }
}

void cpnvmCheckpoint() {
  if (passed_time > interval) {
    threadBarrier();
    snapshot();
    threadBarrier();
  }
}

void snapshot() {
  for (void ∗addr : addrlist) {
    size_t size = resetBits(addr);
    if (size == 0) continue;
    logModification(plog, addr, size);
  }
  finalize(plog);
}
```

Figure 4.4 – Pseudocode for *CpNvm* tracking and checkpointing.

`Mark` function (Line 12). The programmer inserts calls to `cpnvmMark` to track memory regions, by passing the memory region address and size. Tracking only marks modified memory regions in a globally shared *bitmap* (Line 13) and does not record the contents of these regions within an epoch. A single bit in the bitmap corresponds to an 8-byte word in the memory region. It is important to note that, the tracking mechanism is not a log. *CpNvm* also maintains thread-local *address lists* to avoid traversing the entire bitmap to find marked locations. Address lists hold the first address in a marked region of memory. *CpNvm* adds an address if the memory region was not marked before (Line 15). The bitmap and the address lists are both transient.

CpNvm needs to be invoked only at the first write to a memory region in an epoch. Repeated calls on `cpnvmMark` with the same location are harmless, but can lead to performance degradation.

Both word and cache-line granularity treat a single bit as an 8-byte word. However, there are several differences between the granularities. Word granularity synchronizes accesses to the bitmap to avoid data races in setting bits in byte. Different program threads can access different bits in the same byte. Without synchronization, it is possible to unmark an already marked bit in a byte due to a data race. Word granularity atomically reads-and-conditionally-updates bits in the bitmap using compare-and-swap instructions. Word granularity maintains the invariant that an address is in at most one address list within an epoch. However, this synchronization on the bitmap incurs overhead.

We also provide cache-line granularity for better performance. Cache-line granularity uses normal read and write instructions for the bitmap. We assume writing a single byte is atomic, which is the case for *x86-64* and ARM chips. Cache-line granularity records the entire cache line by marking the entire byte instead of setting individual bits. This leads to data races to the bitmap. Setting two different bits in a byte by concurrent threads can erase the work of one of the threads. However, loading an entire byte and setting it is idempotent across two concurrent threads, as these two threads are executing the same operation, which means that recorded work cannot be "unmarked". Another issue is duplicate logging, which we explain in the next section (Section 4.3.3). Cache-line granularity has better performance as shown in section 4.4.2.

4.3.3 Checkpointing

CpNvm breaks the execution into epochs and executes checkpointing at the end of an epoch, in contrast to *InCLL*, which checkpoints at the beginning of the epoch. It is a minor semantic difference in the implementation.

Figure 4.4 shows the high-level algorithm for checkpointing, which requires the use of the `cpnvmCheckpoint` function (Line 19) to initiate a checkpoint. `cpnvmCheckpoint` checks if a sufficient amount of time has passed since its previous invocation (Line 20). The programmer specifies the checkpointing interval between epochs as a configuration parameter to `cpnvm-`

`Init`. If a sufficient amount of time has passed, *CpNvm* takes a checkpoint, otherwise, the call returns. During checkpointing, *CpNvm* takes a snapshot (Lines 22, 27) of all the tracked memory regions and makes this record of modifications durable in a persistent redo log.

The programmer must identify when to checkpoint in the program code. This is to ensure the consistency of checkpointing. If we used a timer to initiate checkpointing periodically, at the specified intervals, there might be an in-progress operation that puts the data structure into an inconsistent state. After a restart, it would be difficult to make the state consistent or complete the operation. We instead opt for allowing the programmer to decide. The points of consistency are program-specific and require programmer knowledge. We insert the call to `cpnvmCheckpoint` in the event loop of both Masstree and Memcached to divide the execution into periodic intervals on the order of milliseconds.

Each thread can run in parallel without synchronization (except for word granularity access to the bitmap). The thread pops an address from its address list (Line 28) and locates the position in the bitmap. *CpNvm* then resets the region's bits in the bitmap starting from the address (Line 29) until an unmarked bit to infer memory region size. If the memory region was already unmarked, then *CpNvm* tries a different entry (Line 30). If unmarking is successful, then *CpNvm* takes a snapshot of the entire unmarked memory region.

Again, there are minor implementation differences between the word granularity and the cache-line granularity. Word granularity avoids duplicate work, as an address can be in at most one of the address lists for an epoch. Similarly, resetting is done using synchronization.

Cache-line granularity allows duplicate work if there is a data race, which is rare as adjacent regions are usually accessed under a critical section. Duplicate entries can be logged twice, which is wasteful but not harmful. Checkpointing is idempotent for an epoch and always snapshots the same version of the data structure. In practice, we observe better performance for cache-line granularity as shown in section 4.4.2.

When checkpointing completes, the entire bitmap is clear and ready for the next epoch.

4.3.4 Background Replay

Figure 4.5 shows the high-level algorithm for background replay (Line 8). The `cpnvmInit` function creates background replay threads separate from the program execution threads, where the number of background threads is specified in the options. The background threads call the `backgroundReplay` function if there are log entries that can be replayed within an epoch.

CpNvm replays the logs an epoch at a time. The background replay threads can run independently for a single epoch. Replaying a single epoch is idempotent as log entries are snapshots of the data structure at the time of checkpointing and always produces the same result in the persistent image even with duplicate work. After a background thread finishes replay, the

```
void *toNVM(void*); // return the persistent image address
void threadBarrierB(); // spinlock thread barrier for background replay
void threadBarrierR(); // spinlock thread barrier for recovery replay
void protect(); // protects the volatile image from the previous execution
void load(); // loads the volatile image from the previous execution
void canReplayEpoch(); // returns true if there are epochs that can be replayed

void backgroundReplay() {
  threadBarrierB();
  logReplay();
  threadBarrierB();
  finalize(plog, discard=true);
}

void logReplay(){
  entries = getEpochEntries(plog);
  for log entry L in entries do{
    void* nvmAddr = toNVM(L→addr);
    memcpy(nvmAddr, L→content, L→size);
  }
}

void recover(){
  while(canReplayEpoch())
    backgroundReplay();

  if(lazy_recovery)
    protect();
  else
    load();

  threadBarrierR();
}
```

Figure 4.5 – Pseudocode for *CpNvm* background replay and recovery.

background thread has to wait for the other background threads to finish. Without a thread barrier, a modification to the persistent image from an earlier epoch could overwrite the modifications done in the current epoch and lead to an inconsistent state. The thread barrier ensures that data race does not occur across epochs.

4.3.5 Recovery

At a restart, when the program calls `cpnvmInit`, *CpNvm* runs the recovery procedure. Figure 4.5 shows the high-level algorithm for recovery (Line 23). Each background recovery thread calls `cpnvmRecover` to recover the state of the persistent image to last committed checkpoint. *CpNvm* first completes background replay and ensures that all the redo logs are replayed to the persistent image.

There are two modes for recovery, lazy and eager recovery. Eager recovery loads the pages directly from *persistent image* to *volatile image* before execution begins (Line 30). Lazy recovery uses the virtual-memory page protection to trap a read/write access to the entire address range of the volatile image from the previous execution (Line 28). Afterward, the execution continues.

Any read and write access by the program to the volatile image triggers a page fault. *CpNvm* uses the `SIGSEGV` handler to lazily bring pages into the volatile image from the persistent image. Since the persistent image version is consistent, the volatile image will start with the last committed checkpoint. Moreover, *CpNvm* only pays the cost of a trap upon first access to a page. It grants full access to other threads once the page is copied in the volatile image.

4.3.6 Masstree Example

Figure 4.6 depicts tracking for *CpNvm* API for Masstree. We rely on the same pseudocode, which we used in section 3.2. Our tracking methodology is similar to the logging methodology described in *InCLL*.

```
class basenode; // lock, version, meta information
template <int width=15>
class leafnode : public basenode{
    basenode *parent, *prev, *next;
    uint64_t permutation; // which key/vals are active
    keytype keys[width];
    valuetype *vals[width];
    void remove(keytype key){
        int idx = find_idx(key);
        remove_idx(&permutation, idx);
        cpnvmMark(&permutation, sizeof(uint64_t));
    }
    void insert(keytype key, valuetype *val){
        //value buffer pointer by val is marked
        int idx = find_idx(&permutation);
        keys[idx] = key;
        cpnvmMark(&keys[idx], sizeof(keytype));

        vals[idx] = val;
        cpnvmMark(&vals[idx], sizeof(valuetype*));

        add_idx(&permutation,idx);
        cpnvmMark(&permutation, sizeof(uint64_t));
    }
    void update(int idx, valuetype *val){
        //value buffer pointer by val is marked
        vals[idx] = val;
        cpnvmMark(&vals[idx], sizeof(valuetype*));
    }
};

void workload::operationHandler(basenode *root, op_t &op){
  //workload driver
  executeOp(root, op); //put or get
  cpnvmCheckpoint();
}
```

Figure 4.6 – *CpNvm* Masstree example.

We use coarse-grained tracking by marking the entire node for internal nodes, leaf node splits, and leaf node merges. We use fine-grained tracking for insertions, deletions, and updates, which are the common operations. For deletions, it is sufficient to track the `permutation` field as the key-value pair is removed. Unlike, *InCLL* which uses an undo log, *CpNvm* uses a redo log approach and needs to track all modifications. That is why, for insertions, *CpNvm* tracks the `permutation` field along with the slots in the `keys` and the `vals` arrays. For updates, it is sufficient to track the pointer to the value buffer. *CpNvm* keeps track of the entire *value buffer* upon allocation.

4.3.7 Memory Capacity

The initial implementation of *CpNvm* limits the size of persistent structures to the size of DRAM. The reason is that *CpNvm* allocates durable memory directly in DRAM. Fortunately, Intel Optane has Mixed Mode (Section 2.1.4).

CpNvm takes advantage of the existing Mixed Mode to expand the DRAM address space by using almost half of NVM and leaving the rest of it for persistence (*persistent image*, *persistent logs*, *CpNvm* metadata such as epoch number). One issue of this approach is that there will be two copies of the persistent data in NVM, one as the NVM copy (*persistent image*) and the other one as the temporary backing store (*volatile image*) in NVM. Keeping a single copy would be interesting, but writes to the *volatile image* do not leave a consistent image that can be used in recovery (see Future Work section 6.1).

4.4 Evaluation

This section evaluates *CpNvm* by using standard benchmarks to measure its performance as well as to compare it against recent systems (Section 4.5). We used the machine configuration from section 3.4.1. We used the Masstree data structure from section 3.4.3.

4.4.1 CpNvm Configurations

We inserted 25 lines to call *CpNvm* functions (in a codebase of 27K LoC) to make Masstree durable and recoverable with 0 – 14.4% overhead. This version used *CpNvm*'s allocation and we persist only the value buffers and the node structures for Masstree. We inserted *CpNvm*'s tracking calls to Masstree's insert and remove operations.

We implemented the entire *CpNvm* system in *CpNvm-B2-NVM* as described in section 4.3. *CpNvm-B2-NVM* used *CpNvm* with 2 background replay threads (B2) and cache-line granularity (64-bytes) tracking. *CpNvm-B2-NVM* provided crash consistency, avoided memory leaks, and dangling pointers. We also report results for *CpNvm-Word-B2-NVM*, which implemented *CpNvm* with word-granularity tracking as described in section 4.3.

CpNvm-B2-MixedMode used the cache-line granularity version of *CpNvm* under Intel Mixed Mode as explained in section 4.3.7. We configured the Optane operation to 50% Mixed Mode, which used 366GB of NVM for App Direct Mode and the rest as backing store for *CpNvm-B2-MixedMode* running on socket 1.

4.4.2 Measurements

Measurements used the same YCSB evaluation default configurations from section 3.4.6 for Masstree.

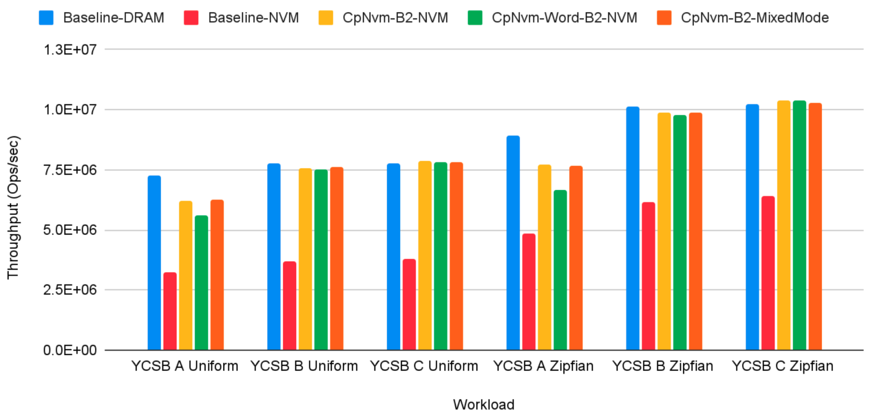


Figure 4.7 – Throughput of Masstree for different implementations.

Figure 4.7 reports the throughput for the *CpNvm* implementations (higher is better) for 6 worker threads. *CpNvm-B2-NVM* is the principle *CpNvm* approach. It uses cache-line granularity tracking and leaves the primary structure in DRAM and checkpoints modifications. *CpNvm-B2-NVM*'s slowdown is less than 15%. Specifically, for write-intensive YCSB A, *CpNvm-B2-NVM*'s slowdown is 14.4% for Uniform and 13.3% for Zipfian.

Read-heavy workloads have minimal overhead. For read-dominant YCSB B, the slowdown is 2.6% for Uniform and 2.7% for Zipfian. In fact, each thread needs to synchronize for checkpointing, which improves Masstree execution performance for YCSB C performance by somewhat less than 1%.

Figure 4.8 shows the breakdown cost for each *CpNvm* mechanism. Lower is better. The majority of the cost is due to checkpointing, which writes the persistent redo logs in NVM. Its cost is 7.6% of execution for Uniform A and 7.5% for Zipfian A workloads. A single thread logs 5.4MB in 6.2ms for Uniform A and 4.8MB in 5.4ms for Zipfian A, on average per epoch. For YCSB B, checkpointing costs 0.9% for Uniform B and 1.1% for Zipfian B, since read-only workloads do not require the tracking or the checkpointing mechanisms.

The second largest cost is the tracking mechanism that registers modified memory regions in the bitmap and address lists. Its is 3.4% for Uniform A and 3.2% for Zipfian A, and 1.3% for Uniform B and 1.2% for Zipfian B workloads.

The remaining cost is due to using Optane and a minimal cost from background threads. The cost of using Optane for memory-mapped files is 2.7% for Uniform A and 2.2% for Zipfian A. The cost of background threads is less than 1%.

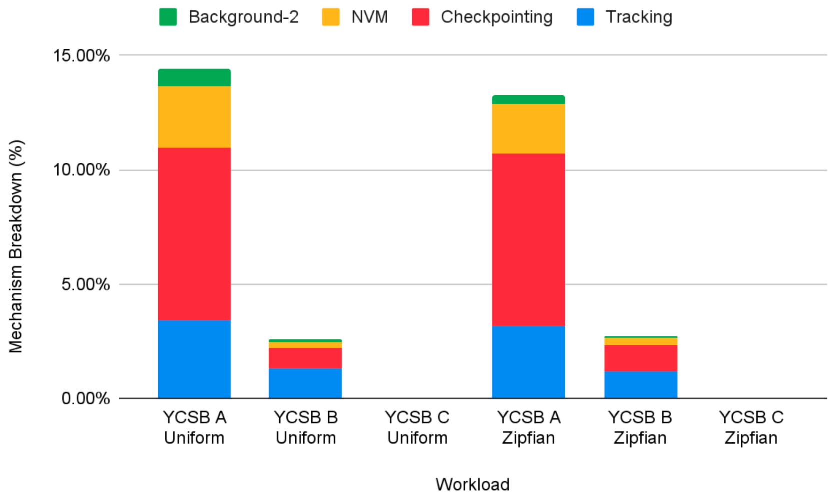


Figure 4.8 – Mechanism breakdown for *CpNvm-B2-NVM*.

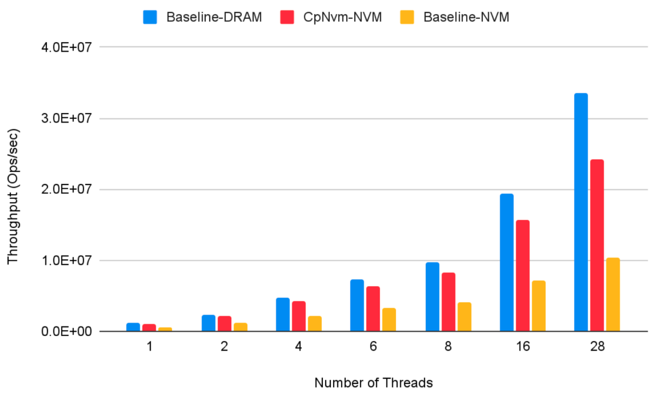
We also measured the time spent replaying the logs to NVM copy for an epoch (`logReplay` function in figure 4.5). A single background thread replayed 16.2MB in 30.6ms for Uniform A and 14.4MB in 26.6ms for Zipfian A, on average for a 64ms epoch. Background replay does not affect program performance if the memory bandwidth is not saturated. However, background replay is slower than checkpointing as the copying is from the persistent log in NVM to the persistent image in NVM (random access), while checkpointing sequentially copies from DRAM to the persistent log.

CpNvm-Word-B2-NVM is slower than *CpNvm-B2-NVM* due to the cost of synchronization, which is more significant in the Zipfian case. The slowdown of *CpNvm-Word-B2-NVM* over *Baseline-DRAM* is 22.9% for Uniform A and 25.3% for Zipfian A.

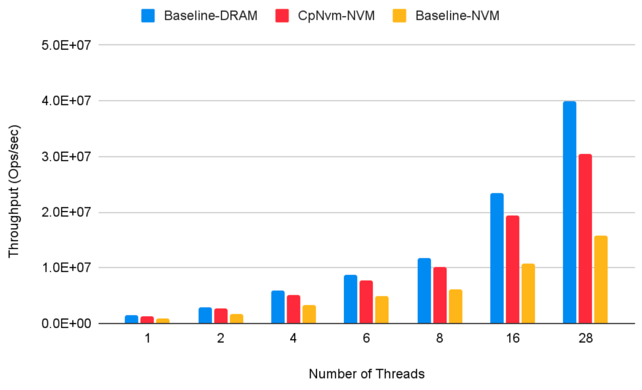
We also evaluated *CpNvm-B2-MixedMode*, which uses Mixed Mode Memory to increase the volatile memory capacity. The goal of the experiment to observe the effects of using Mixed Mode Memory and whether it is a viable configuration option. The difference between *CpNvm-B2-MixedMode* and *CpNvm-B2-NVM* is less than 1%. Since the Masstree data structure easily fits in DRAM, this measurement does not include the cost of paging to NVM. We ran additional micro-benchmarks, which bounded this overhead at 1.5—2.6x for memory-operation-intensive kernels.

We also varied the number of background threads from 0 to 6 for *CpNvm-B2-NVM* from figure 4.7. We observed that the performance effect of background threads is minimal, less than 1% across all the threads.

Figure 4.9 shows the scalability of YCSB A with different numbers of worker threads. For *CpNvm*, we did not use any background threads for this experiment. As we increase the parallelism, the throughput of YCSB A increases from 1.1E6 Ops/sec to 2.4E7 Ops/sec for the uniform workload and 1.3E6 Ops/sec to 3.1E7 Ops/sec for the Zipfian workload. This



(a) Uniform A workload.



(b) Zipfian A workload.

Figure 4.9 – Throughput of the *CpNvm* implementation of Masstree for different numbers of threads.

improvement is expected as Masstree was designed to achieve low memory contention and good parallel performance. However, Optane limits its scalability. The throughput overhead increases from 11% to 28% for Uniform and 11% to 24% for Zipfian. *Baseline-NVM* throughput overhead increases from 51% to 69% for Uniform and 39% to 61% for Zipfian. Zipfian performs better because of the increased locality with a skewed distribution.

Masstree is scalable and achieves 28*x* speedup for Uniform A and 27*x* for Zipfian A workloads over a single thread. However, as we increase the number of threads, the limited bandwidth of Optane becomes significant even for the *CpNvm* design. The speedup of *CpNvm* drops to 23*x* for both Uniform A and Zipfian A.

Figure 4.10 shows the effect of varying the number of background replay threads for *CpNvm* (B-<number of background threads>) when we saturate performance with 28 worker threads. The performance overhead increases to 53.7% for Uniform A and 48.3 for Zipfian A as we increase the number of background threads to 28. By comparison, if we put the Masstree data structure in NVM, without even performing cache flushes, the throughput degradation

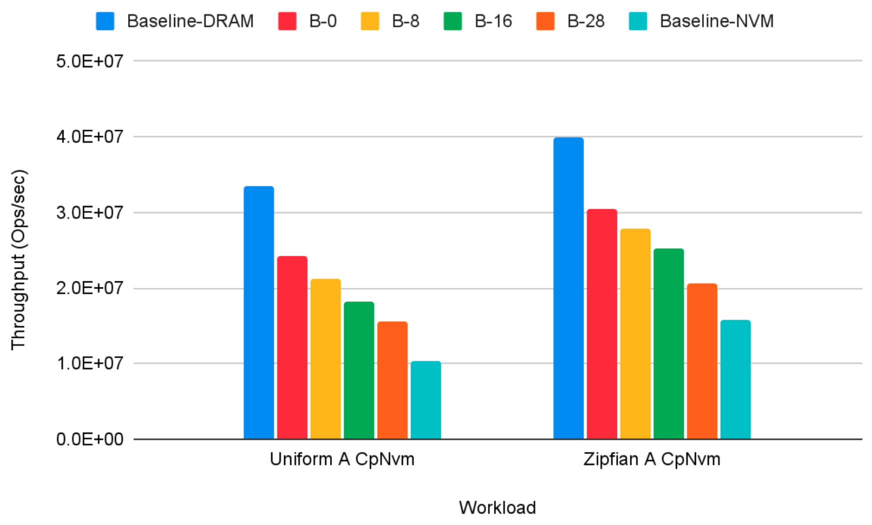


Figure 4.10 – Throughput of the *CpNvm* implementation of Masstree with 28 worker threads and different numbers of background replay threads.

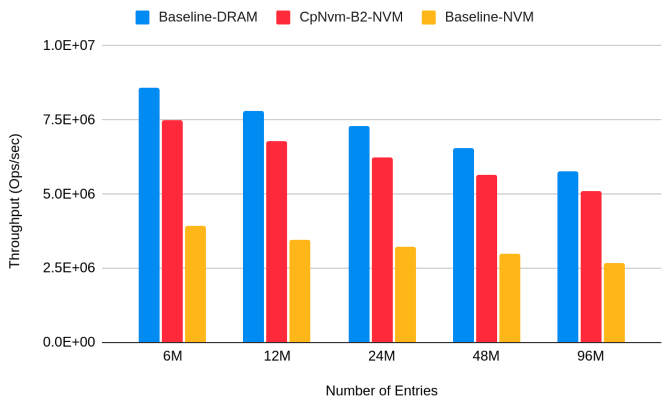
for 28 threads is 68.8% for Uniform A and 60.7% for Zipfian A. Baseline Masstree is a mixed read/write workload, randomly accesses memory, and runs concurrently, all of which are known to be detrimental to NVM performance [188, 91, 120].

We varied the checkpointing interval from 16ms to 256ms. For *Baseline-DRAM*, the difference between the intervals is less than 1%. Overall, the performance improves for *CpNvm-B2-NVM* as we increase the time between checkpoints and reduce the number of synchronization barriers. The change is not large for Uniform A, going down from 15.5% to 13.4% overhead. The change is more significant for Zipfian A as the overhead goes from 15.2% to 10.1%. On average per epoch per application thread, checkpointing for 16ms logs 1.4MB in 1.6ms for Uniform A and 1.3MB in 1.4ms for Zipfian, and for 256ms logs 20.9MB in 23.8ms for Uniform A and 16.9MB in 18.9ms for Zipfian A. On average per epoch per background replay thread for 16ms replays 4.1MB in 8.3ms for Uniform A and 3.9MB in 7.2ms for Zipfian A and for 256ms replays 62.8MB in 118.4ms for Uniform A and 50.6MB in 91.0ms for Zipfian A.

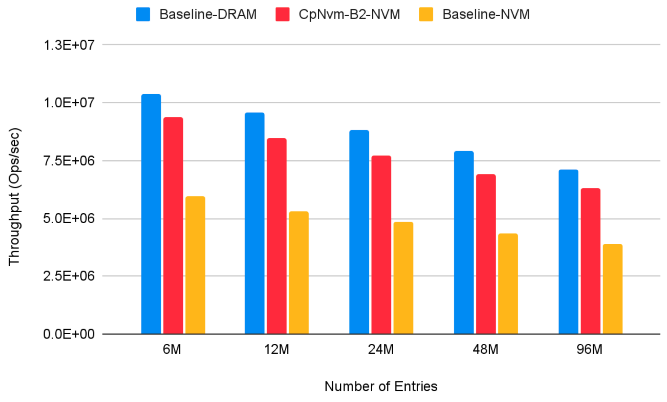
Figure 4.11 shows the effect of the initial tree size, where the cost of data structure traversal dominates throughput. The default number of entries is 24M across the experiments. Compared to *Baseline-DRAM*, the slowdown of *CpNvm-B2-NVM* is between 11 – 14% for Uniform A and 10 – 13% for Zipfian A.

4.4.3 Recipe

We compared both *CpNvm* and *InCLL* implementations of Masstree to the published Recipe [108, 7] (published after *InCLL*) version of Masstree, which is optimized for Optane. Recipe, however, revamped Masstree's structure. Recipe removed internal nodes and used leaf nodes instead,



(a) Uniform A workload.



(b) Zipfian A workload.

Figure 4.11 – Throughput of baseline Masstree and *CpNvm* with DRAM and Optane evaluations for different initial tree sizes.

modified the node versioning, and used external locks (`std::mutex`).

The baseline *Recipe-DRAM* placed the Masstree data structure in DRAM. *Recipe-NVM* placed the Masstree data structure in NVM using libvmmalloc [3] and followed the published approach that Recipe used for its evaluation. We note that *Recipe-NVM* with *libvmmalloc* is not entirely persistent as libvmmalloc is not a persistent allocator. libvmmalloc only redirected allocation calls to a memory-mapped file in NVM. Recipe also used pre-allocated value buffers, which lowered its runtime cost. Unfortunately, Recipe's Masstree implementation on PMDK was not complete [7] and we do not provide PMDK comparison.

Figure 4.12 compares *CpNvm-B2-NVM* against the published version of Recipe and our *InCLL* implementation. For YCSB A, *Recipe-NVM* has a 49.9% slowdown (1.9*x CpNvm*) for Uniform and 39.8% slowdown (1.7*x CpNvm*) for Zipfian compared to *Recipe-DRAM*. For YCSB C, Recipe pays the cost of keeping its data structure in NVM with a slowdown of 51.1% (2.1*x CpNvm*) for Uniform and 38.0% (1.7*x CpNvm*) for Zipfian.

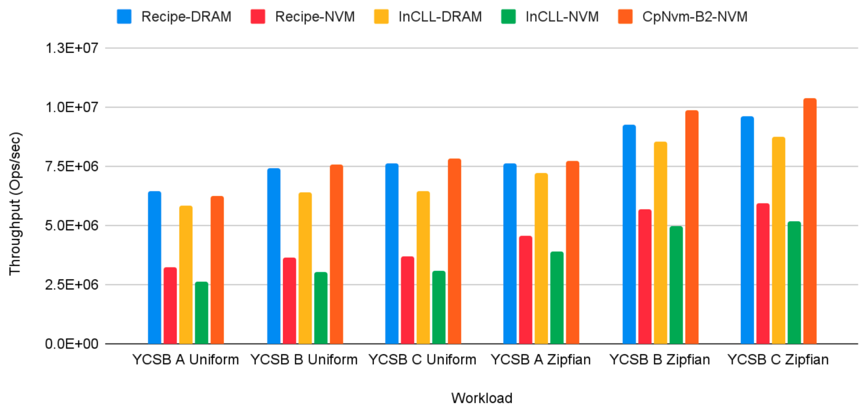


Figure 4.12 – Throughput of Recipe and InCLL compared to different Masstree implementations.

4.4.4 InCLL

We compare our *CpNvm-B2-NVM* Masstree implementation to our previous *InCLL-NVM* Masstree implementation. For YCSB A, *CpNvm-B2-NVM* is 2.4x faster for Uniform and 2.0x faster for Zipfian compared to *InCLL-NVM*. For YCSB C, *CpNvm-B2-NVM* is 2.5x faster for Uniform and 2.0x faster for Zipfian compared to *InCLL-NVM*.

4.4.5 PMThreads

We also compared *CpNvm* to PMThreads, another fine-grained checkpointing system. PMThreads leveraged locking in a concurrent program to identify persist durable regions. PMThreads operated on DRAM pages and propagated the modifications to NVM when the program is quiescent (no active critical sections) to avoid checkpointing data structures in an inconsistent state.

PMThreads required the program to use `pthread_mutex_lock`. Masstree did not use mutex lock from pthreads and instead implemented its own locking mechanism. We introduced a thread-local lock that we use for the entirety of an insertion or a deletion. Since the lock is thread-local, operations do not interact with other concurrently running threads but are sufficient to trigger the PMThreads's checkpointing mechanism. We also applied the thread-local locking mechanism to Masstree's memory reclamation, which did deallocations based on the epoch. To be fair, we evaluated the *CpNvm* version using these unnecessary locks. The performance results would be even further skewed in favor of *CpNvm* without them.

For the PMThreads experiments, we used PMThreads's allocator and do not modify anything. We disabled the artificial delay for PMThreads [185] by setting it to 0. We reported two versions using PMThreads, which were *PM-I-DRAM* and *PM-P-DRAM*. *PM-I-DRAM* used compiler instrumentation to track writes at word granularity. *PM-P-DRAM* used virtual memory protection to track accesses at page granularity using *mprotect*. However, both versions marked an entire page as dirty during execution and use page-diffing to identify modified bytes to be copied to NVM. We used 6M initial key-value pairs to populate the Masstree. For this experiment, we compiled Masstree, *CpNvm*, and PMThreads with clang-7.0.1.

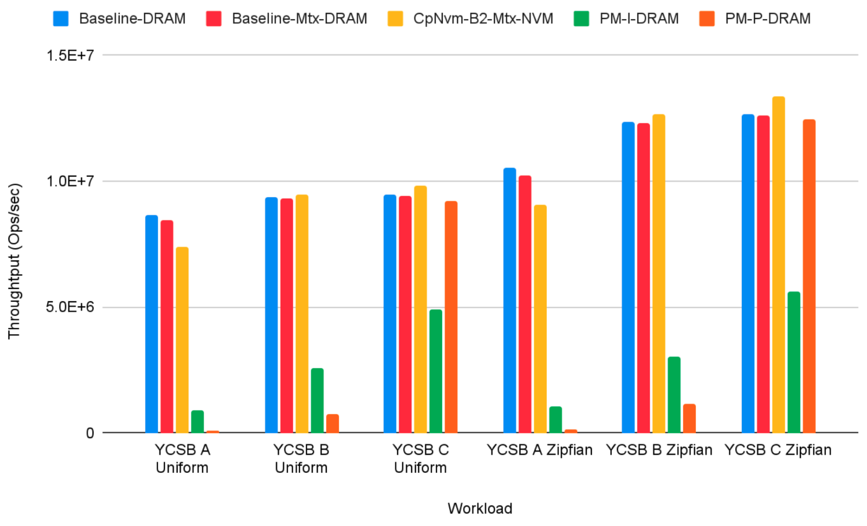


Figure 4.13 – Throughput of *CpNvm* and PMThreads implementation of `pthread_mutex-_lock` Masstree.

Figure 4.13 shows the `pthread_mutex_lock` Masstree running with *CpNvm* and PMThreads. Overall the thread-local locking mechanism introduces 2.1% overhead for Uniform A and 2.9% overhead for Zipfian A, while for the rest of the benchmarks the overhead is less than 0.5%. *CpNvm*'s slowdown is 14.5% for Uniform A and 14.2% for Zipfian A and the read-dominant workloads perform better due to synchronization for checkpointing. For write-heavy workloads, *PM-I-DRAM* is 94.1% slower for Uniform A and 93.2% slower for Zipfian A. For read-only workloads, *PM-I-DRAM* is 47.9% slower for Uniform C and 55.5% slower for Zipfian C. The poor performance is mainly from the cost of instrumenting each store in Masstree and logging entire pages. On the other hand, *PM-P-DRAM* has lower overheads as the virtual memory mechanism is not invoked for read-only workloads which leads to a slowdown of 2.4% for Uniform C and 1.4% for Zipfian C. For Uniform A, *PM-P-DRAM* increases page faults by 26x (perf profiler). However, *PM-P-DRAM* pays a huge cost for write-heavy workloads due to page-level tracking which results in 99.1% overhead for Uniform A and 98.9% for Zipfian A.

4.4.6 Recovery

We tested *CpNvm* by crashing the system at random points in an epoch, restarting the system, doing log recovery, and verifying that the persistent image state after replay is the same as the volatile image state before the crash.

To measure log recovery, we ran Masstree with the Uniform A workload with 24M initial entries.

We crashed at the end of the 5^{th} epoch after committing the redo logs. As Masstree replayed background threads for the previous epoch in the current epoch, the recovery code only needed to replay the committed epoch. Recovery replayed with 6 threads and each thread on average spent 14.1ms replaying $5.4MB$.

After a restart, we continued the Uniform A workload and the throughput normalized after 4s as shown in figure 4.14. *CpNvm* creates a separate thread to lazily bring in pages on top of the worker threads. *CpNvm* loads 440K pages from NVM in total.

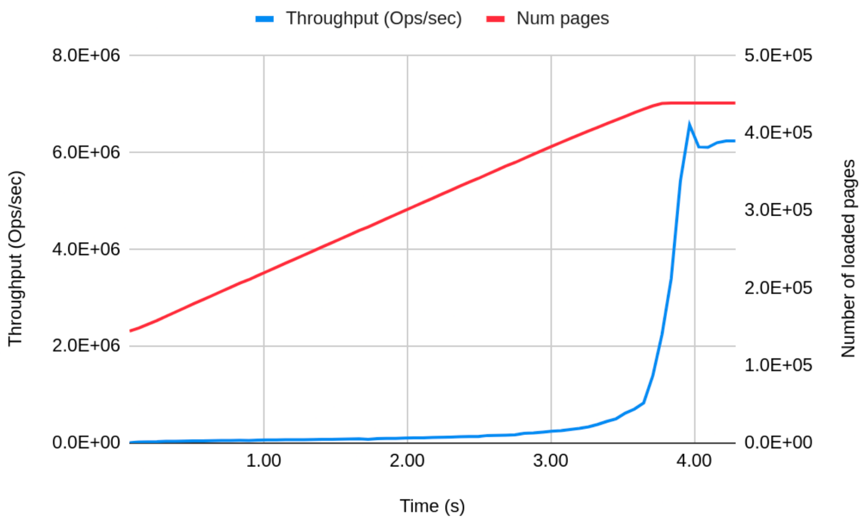


Figure 4.14 – Execution of *CpNvm* after a restart.

For this program, it is possible to use `memcpy` and load the entire persistent image more efficiently before starting the execution. This avoids paying the cost of page protection. Eagerly loading the NVM pages to DRAM takes around 145ms with 6 threads (760ms with a single thread). The throughput normalizes after 4 epochs. Different data structures with different access patterns might perform better with eager or lazy loading. The choice is left to the programmer.

4.4.7 Memcached

Finally, we evaluated the performance of *CpNvm* for the data center application Memcached-1.2.4 [61], an in-memory key-value store, using the Memtier [2] benchmark. We enabled processor socket 2 and ran memtier benchmark from socket 2. Memtier performed set and get operations on a Memcached server running on socket 1. The evaluation used 50 clients, 8 threads, 128 byte data, 200 ms epochs running for 2 seconds. We persisted the heap allocations and the hash table related data structures by adding 29 lines of *CpNvm* modifications in a 4.5K LoC application. We did not make the LRU cache of Memcached durable, and the entire system was not durable, which is a challenging endeavor [119]. After a restart, the hash table data structure and its contents are durable, however, recovery needs to handle LRU cache explicitly. *CpNvm* used 4 background threads.

We also compared it with PMThreads 4.4.5 and Atlas running Memcached. Atlas is similar to PMThreads in that it requires lock-based concurrent applications. However, Atlas uses transactions instead of checkpointing and all its log and the program data structures reside in NVM. We used DRAM evaluation for Atlas as described in [32] without any added latencies. Similar to *CpNvm*, we did not persist the LRU cache for either PMThreads and Atlas.

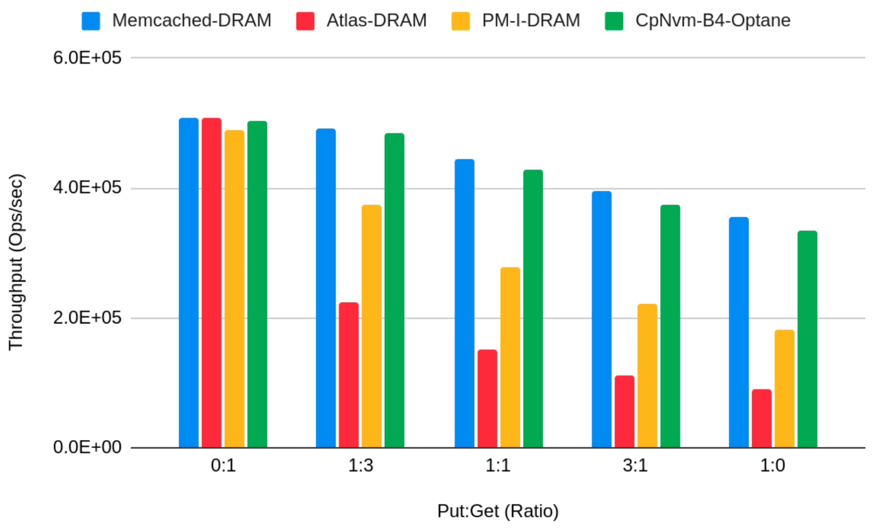


Figure 4.15 – Throughput for different versions of Memcached.

Figure 4.15 evaluates the *CpNvm*-annotated version of Memcached with a varying ratio of gets and puts. The worst-case for *CpNvm* is the 1:0 put:get ratio (write-only), which incurs 5.6% overhead. The more balanced 1:1 ratio incurs 3.3% overhead. In the worst case, PM-Threads with compiler instrumentation has 48.6% overhead due to its tracking mechanism and checkpointing. Atlas has 74.6% due to transactional dependencies between logs and tracking.

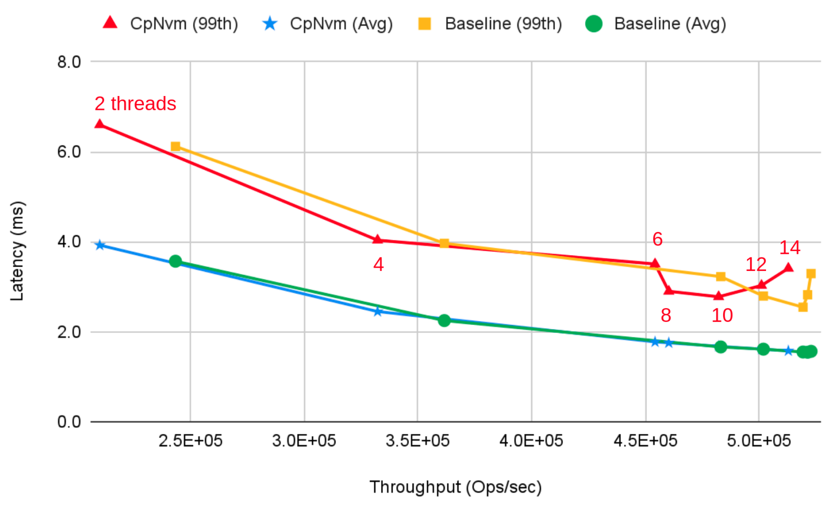


Figure 4.16 – Latency vs. throughput measurements for Memcached and *CpNvm* for 1:1 (Put:Get).

Figure 4.16 reports the latency vs. throughput measurements for baseline Memcached and *CpNvm*. We present the average latency and the 99^{th} percentile latencies. We increased the number of client threads to 14 and measured the 1:1 (Put:Get) case, as the overhead of *CpNvm* for read-intensive workloads are minimal. We varied the number of Memcached worker threads from 2 to 14 in increments of 2, using half the number of background replay threads as worker threads. The drop at 8 threads for *CpNvm* 99^{th} percentile latency is due to saturation. Overall, the curves for *CpNvm* and baseline Memcached are very similar. The baseline has better throughput and latency for each point, however, the difference is not large. It shows that pausing for checkpointing does not affect in this benchmark very much.

4.5 Related Work

Both *CpNvm* and *InCLL* are checkpointing systems with intervals in the order of milliseconds. Chapter 2 provided more details about crash-consistency and checkpointing in general. In this section, we discuss related work for both of these systems and consider only the relevant subset of the work.

4.5.1 NVM Programming Systems

Prior work builds on ideas from transactional memory [76]. There are many designs that rely on software transactional memory (STM) [33, 37, 43, 68, 72, 79, 80, 98, 112, 141, 149, 175] and HTM [18, 31, 65]. The two foundational works in transactional design are Mnemosyne [175] and NV-Heaps [37]. Both use write-ahead logging for crash-consistency. Mnemosyne relies on a redo log and uses a word-based transactional memory system based on TinySTM. For

Mnemosyne, the programmer needs to modify each read and write operation within a transactional region to use the redo log. NV-Heaps relies on an undo log and uses object granularity. For NV-Heaps, the programmer needs to insert calls to the logging to track memory regions.

CpNvm and *InCLL* persist the data structure using checkpointing at periodic intervals instead of using transactions. The only requirement of *CpNvm* and *InCLL* is to break the program execution into epochs. *CpNvm*, like Mnemosyne, relies on a redo log approach. The full redo log contains information to recreate the entire data structure from scratch. The *CpNvm* redo log contains the latest value for a modified memory region at the end of the epoch before checkpointing. *InCLL* uses an undo log approach similar to NV-Heaps. *InCLL* implements in-cache line logging and an external log for undo logging. *InCLL* undo log mechanism contains modifications done after the start of a new epoch.

Kamino-Tx [121] optimizes data copying for transactions. There are two heaps, the main heap and the backup heap, that are stored durably in NVM. The backup heap asynchronously synchronizes with the main heap, however, Kamino-Tx does not allow dependent transactions to execute before the completion of asynchronous transfers. Kamino-Tx relies solely on NVM similar to *InCLL*, however, in *CpNvm* the operations occur in DRAM.

DudeTM [112] is a transactional system based on redo logging and shadow memory to reduce the redo log programming effort and reduce persistent operations on the critical path of the program. Similar to *CpNvm*, the program operates on DRAM data structures. DudeTM brings in the page from NVM upon access and keeps it until eviction is necessary. For each transaction, DudeTM creates a volatile redo log that acts as a backup and asynchronously persists the redo log to NVM. Similar to *CpNvm*, the persistent redo logs update an NVM copy. DudeTM relies on page-level protection to detect writes and to create redo logs for transactions, which can lead to write amplification.

Compared to DudeTM, *InCLL* requires the programmer to modify data structure fields and methods to use in-cache-line logging. *CpNvm* requires the programmer to annotate the code with calls to tracking for checkpointing the modified memory regions. *CpNvm* only uses page protection to bring in pages from the previous execution. *CpNvm* and *InCLL*'s approach require programmer effort, which is better for performance instead of relying on costly instrumentation or protection by an STM system. DudeTM does not report the overheads for the STM system, does not use Intel Optane for evaluation, and simulates execution with introduced delays at each fence instruction. Despite the lack of overheads from the DudeTM, *CpNvm* overheads reported in section 3.4 are comparable ($< 15\%$)[1].

There are transactional approaches that either rely on hardware transactional memory [18, 31, 65], or hardware modifications [73, 93]. Similarly, several checkpointing systems [67, 150, 182] require hardware modifications or require HTM support. Both *InCLL* and *CpNvm* are pure software approaches and do not require any HTM support. Both systems require support only

[1]At the time of working on *CpNvm*, we could not obtain a copy of DudeTM to conduct the experiments

for NVM.

Pronto [122] is an operational logging approach and records data structure operations instead of memory regions. Pronto uses background threads to asynchronously log the data structure operation before the operation completes. After a restart, Pronto re-executes the data structure operations to recreate the data structure. Pronto implements a naive checkpointing model that takes a snapshot of the entire data structure. Both *InCLL* and *CpNvm* might benefit from the operational logging approach by replacing the checkpointing stage and data structure design in Pronto. Using Pronto, both *InCLL* and *CpNvm* can avoid lost work in a failed epoch.

Lock-delimited NVM systems [32, 78, 89, 113, 185] are an alternative to transactional systems. These systems rely on a compiler infrastructure to convert existing locked-based concurrent programs into durable equivalents. The foundational work in this area is Atlas [32], which uses an undo log for crash-consistency. Similar to *InCLL*, both the data structure and the logs are in NVM. NVthreads [78] use a redo logging approach, spawns a process for each different thread to provide a separate address space, and merges modifications to a single NVM store using synchronization points. JUSTDO logging [89] executes the FASE to completion after a restart. iDO [113] improves JUSTDO logging efficiency by further dividing each FASE into idempotent regions to avoid excessive logging.

Another important work is failure-atomic msync [135] that modifies the `msync` system call to persist all writes to a memory-mapped region. Failure-atomic msync uses the journaling mechanism of a file system to track modifications and provide crash consistency. The execution of the `msync` is similar to calling a checkpoint as all the modifications become visible in NVM.

4.5.2 Checkpointing

As we explained in section 2.3, checkpointing is a well-established technique with considerable prior work [1, 55, 99, 139, 153, 171, 176]. The novelty of *InCLL* is its mechanism to remove explicit write-back instructions from the fast-path data structure modifications. The novelty of *CpNvm* is the novel combination of existing mechanisms to checkpoint efficiently using byte-addressable Optane. Both *InCLL* and *CpNvm* only persist program data structures. They do not checkpoint the entire state of the process [70].

Compared to traditional long-running HPC programs, which rely on page-granularity tracking and long time intervals from seconds to minutes, both *InCLL* and *CpNvm* use fine-grained tracking with fine-grained intervals on the order of milliseconds.

Both *InCLL* and *CpNvm* are single-node, single-device checkpointing systems unlike multi-level and distributed checkpointing systems [29, 46, 74, 125, 130]. One major disadvantage of relying on local storage is node-local NVM failure, which is susceptible to data corruption.

Compared to ARIES [124] based databases [77, 161], both *InCLL* and *CpNvm* track data structure modifications directly due to the lack of an intermediate layer between data and the program. In-memory databases [30, 156] and virtual machines [41, 168] rely on copy-on-write (COW) to implement checkpointing. These systems use `fork` or `mprotect` to protect the address space that needs checkpointing. Page protection mechanism need to set each page entry underlying a data structure. For example, forking [30] copies the entire page table, which is dependent on the program's address space. This operation can be costly. Cache size for *InCLL* and the redo log size for *CpNvm* bound the time spent in checkpointing. The time is independent of data structure size. Moreover, *InCLL* and *CpNvm* do not introduce page protection overhead to the normal execution. *CpNvm* introduces page protection overhead to bring in pages from a previous execution.

There are many new designs that use checkpointing with NVM [38, 60, 67, 95, 123, 128, 150, 181, 190, 182, 185]. Most of the designs use NVM as a high bandwidth device [123]. Other designs specifically exploit the byte-addressable interface of NVM [60, 128, 185] and also support high-frequency checkpointing.

PMThreads [185] is a checkpointing system for lock-based concurrent programs. PMThreads uses lock-delimited regions to identify modifications to the memory regions. The system assumes that lock-delimited critical sections lead to a data structure consistent state. The implementation of PMThreads uses 2 NVM pages for a single DRAM page and allocates the pages eagerly. PMThreads uses a dual-version redo log approach to capture and store modifications. Similar to *CpNvm*, the program operates on the DRAM data structure and treats one of the NVM pages as a working copy and the other one as the consistent copy. PMThreads tracks modifications in two ways. One uses the compiler infrastructure to track stores at a page granularity and the other version uses page protection. Both versions commit the modifications in DRAM to the working NVM copy using byte-diffing to avoid NVM writes at the cost of extra NVM reads. Both *InCLL* and *CpNvm* use fine-grained tracking and do not require byte-diffing for tracking changes. PMThreads evaluation only introduces latencies at fence instructions (Section 4.4.5).

InCLL does not keep two copies of the same data structure and does not use Mixed Mode. There is only one persistent version that resides in NVM in App-Direct Mode. *InCLL* incurs space overhead due to the use of in-cache-line logs and bounded external logs. *CpNvm* keeps a single NVM page for a DRAM page and the bounded logs. PMThreads minimizes programmer effort at the cost of program performance, as the programmer only needs to use the PMThreads allocator for persistent data structures.

Dalí [128] is a periodically persistent hash table that persists modifications periodically by flushing the entire cache hierarchy. Dalí hash table uses a multi-versioning scheme, where updates prepend the new version to the appropriate bucket in the hash table and reads return the latest valid version. Dalí implements garbage collection to remove failed modifications. Both *InCLL* and Dalí periodically flush the entire cache hierarchy with the `wbinvd` instruction to ensure persistence. However, the multi-versioning strategy leads to prepending new data instead of reusing the existing structures leading to inefficient cache usage. In *InCLL* there is a single version of the data structure, which is modified by in-place updates and the undo log reverts the modifications. Both systems store the data structure directly in NVM, are program-specific, and hard to generalize. *CpNvm* operates on DRAM data structure and does not use `wbinvd`, however provides a general application-level checkpointing API.

Montage [181] divides the execution into epochs as well, however, it allows the execution to continue during checkpointing. However, a failure can lose two epochs worth of modification. Montage only persists value buffers instead of the full structure and requires recovery code for a data structure. Both *InCLL* and *CpNvm* lose a single epoch.

Phoenix [60] implements a multi-level checkpointing design and uses a log-structured file system on NVM. Phoenix relies on an object-granularity while *CpNvm* and *InCLL* use fine-grained tracking. Algorithm-Directed Crash Consistence [190] uses algorithmic information to determine when to evict cache lines during the program execution. It requires careful analysis of the program code and is non-trivial to reason about. Mona [64] and NVM-checkpoints [95] focus on optimizing pre-copy mechanism for speculative checkpointing, which is a different technique than ours. NV-Checkpoint [44] uses compiler instrumentation to create multi-versioned data structures depending on developer annotations and optimizes the timing of the checkpoints using machine learning models. The program can operate both on DRAM and NVM data structures and at least one version of the data structure is in NVM. NV-Checkpoint focuses on the domain of long-running HPC applications with long intervals between checkpoints instead of durable data structures with fine-grained epochs. NV-Checkpoint only provides examples for multi-versioning arrays, quadtrees, and graphs and persists the data structure by copying the DRAM version to the version in NVM.

Hardware checkpointing designs [150, 182] rely on hardware modifications such as extensions to the memory controller. *InCLL* and *CpNvm* do not require any architectural changes and do not require any ISA modifications.

4.5.3 Durable Data structures

Most of the prior work focuses on reducing the number of writes to NVM [40]. wB+-Trees [35] minimize writes to NVM with the implementation of a field similar to `permutation` in Masstree. NV-Tree [189] relies on append-only insertions to log to reduce the NVM write overhead. WORT [107] uses two explicit write-back and fence instructions for updates to an NVM radix tree. BzTree [17] relies on durable lock-free multi-word CAS implementation PMwCAS to implement a B+tree. They do not report the number of explicit write-back and fence instructions, however, at least two are required for each PMwCAS. Each insertion uses two PMwCAS, which requires four write-back and fence instructions in total.

The Log-Free data structures work [45] uses link-and-persist and link-cache to benefit from atomic updates and makes lock-free data structures durable. CCEH [127] uses buckets of cache-line sizes to find an entry in the table and only requires a maximum of two cache line accesses. The work by Cohen et al. [40] reduces explicit write-back instructions using writes to the same cache line to provide an optimal logging algorithm. The Efficient Lock-Free Durable Sets work [193] minimizes the write-back instructions and fences by avoiding making pointers durable and persisting only value buffers, which leads to a single fence for updates and no fence for reads. MOD [75] exploits the properties of functional data structures to minimize ordering.

Similar to *CpNvm*, several data structure [81, 186, 189] designs exploit DRAM for performance. Echo [19] is a key-value store that stores incoming operations in a thread-local DRAM store and propagates these items to a singular master NVM store upon a commit. NV-Tree [189] stores internal nodes in DRAM. FPTree [134] also places the internal nodes in NVM and places the leaf nodes in NVM. HiKV [186] uses a NVM hash table for fast operations and B+ tree in DRAM to support slow scan operations. Bullet [81] uses a DRAM hash table to take the load off from the slow NVM hash table and propagates changes to the NVM version epoch-based. NVTraverse [62] reduces flushes for the traversal part of lock-free traversal data structures. Efficient Lock-Free Durable Sets [193] only persist value buffers in NVM.

Recipe [108] describes a set of methods to convert existing DRAM-optimized data structures. Recipe requires the entire data structure to be in NVM similar to *InCLL*.

4.5.4 Persistent Memory Allocation

It is essential to have persistent memory allocation to avoid memory leaks and dangling pointers after a restart. For this reason, NVM programming requires the use of persistent memory allocation.

The nvm_alloc [158] breaks the allocation into two stages, which are reserve and activate. After a restart, nvm_alloc rolls back reserved allocations, and each stage flushes data to NVM. NV-Heaps [37] uses redo logs for implementing allocation with garbage collection and reference counting. Makalu [23] uses garbage collection to avoid write-back and fence instructions. However, Makalu has to traverse the memory for garbage collection after a restart, which increases program restart time.

PAllocator [133] keeps per core big and small allocators and provides defragmentation. Poseidon [48] provides metadata protection using Intel MPK. Ralloc [28] minimizes writes during allocation using a garbage collector after a restart similar to Makalu. However, Ralloc improves Makalu's conservative garbage collector using filter functions to optimize pointer traversal. *InCLL* uses fine-grained checkpointing and avoids write-back and fence instructions for memory allocation, while *CpNvm* uses the tracking infrastructure to mark changes to the allocator. After a restart, *InCLL* lazily recovers the pointers, while *CpNvm* pointers are at the last committed checkpoint upon the completion of log replay.

4.6 Summary

The chapter presented *CpNvm*, which is an application-level checkpointing library for C/C++ programs that can persist application data structures efficiently in NVM. *CpNvm* has low run-time overhead. Masstree evaluation shows overheads less than 15% for the YCSB benchmarks and Memcached evaluation show less than 6%. The relaxed crash-consistency mechanism of periodic checkpointing is suitable for cloud workloads where the program can be suspended between tasks to capture a consistent image.

CpNvm also provides a simple checkpointing programming model, which minimizes developer effort for modifying the program. We modify less than 30 lOC individually in both Masstree and Memcached.

CpNvm's novelty is its simple checkpointing model, which requires minor changes in the program codebase, and its efficient implementation, which provides an effective crash-recovery mechanism for persistent data.

5 Using Dataflow Analysis to Find NVM Program Bugs with FlowNvm

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
— Donald E. Knuth

The byte-addressable interface provides convenience for NVM programming. A programmer, however, needs to carefully reason about the persistence of write operations due to the volatility of the processor caches. Reasoning about cache line write-back instructions, object layouts, and ordering persistent writes is not straightforward. Mistakes in programming can lead to hard-to-find and subtle bugs that can corrupt program data after a restart or result in performance slowdown.

In this chapter, we introduce *FlowNvm*, which is an inter-procedural data-flow analysis tool to find NVM programming bugs. We identify three programming patterns and two anti-patterns that are commonly used or abused across NVM programs. *FlowNvm* can identify instances of the programming patterns and anti-patterns in program code. We demonstrate that *FlowNvm* can find previously known bugs and can be useful for finding real bugs in NVM programs.

This chapter is organized into the following sections:

- Section 5.1 describes the design goals and the motivation.
- Section 5.2 describes the three programming patterns and two anti-patterns that are identifiable by static analysis and are important for crash consistency and program performance.
- Section 5.3 describes the high-level design and the dataflow framework.
- Section 5.4 explains the implementation.
- Section 5.5 provides an evaluation.
- Section 5.6 compares *FlowNvm* to other NVM programming bug finding tools.

5.1 Introduction

NVM programming is a complicated task. A missing or misplaced flush instruction or reordered pair of writes might not affect a program's normal execution. Moreover, the lack of an expensive write-back and fence instructions might even improve performance. However, NVM programming errors can become apparent when the program fails at a specific point in the program's execution and state needs to be recovered. Failure at such a location can lead to situations where execution cannot be resumed because the NVM state is inconsistent. This type of programming error, similar to a data race [25], is execution-dependent, hard to reproduce, and difficult to detect.

Eliminating crash-consistency bugs in a program, by conservatively putting flush and fence instructions in the code, introduces runtime overhead to the program. Unnecessary flush and fence instructions may degrade performance [118, 181].

A programmer can avoid programming with these primitives. Databases with NVM support [91, 188], durable data structures [17, 81, 108], file systems optimized for NVM [94, 101, 174], or higher level NVM programming libraries based on transactions [121, 141, 149, 175] or checkpointing [60, 185] all can abstract away these primitives. However, the implementation of these libraries and systems still need to be crash-consistent. Furthermore, it is possible for programmers to incorrectly use the libraries.

NVM programming bugs are programming mistakes that either lead to inconsist program data in NVM after a crash or cause performance degradation due to the unnecessary use of durability primitives such as cache line flush and fence instructions. Previous work [63, 82, 104, 115, 132] relies on runtime profiling and testing to find NVM programming bugs. Using these tools requires many test cases with many different inputs to achieve good test coverage.

We make two key observations:

1. Programmers use a small number of programming patterns that provide durability across different NVM code bases.
2. We can use static program analysis at compile time to find many programming patterns that can lead to NVM programming bugs.

These aforementioned observations provide a basis for building a static analysis tool to find NVM programming bugs. We identify three programming patterns and two anti-patterns that are suitable for static analysis. We explain the patterns in section 5.2.

We can use inexpensive static analysis to identify violations of the programming patterns and occurrence of anti-patterns. We propose using data flow analysis to do the analysis and find bugs. We build *FlowNvm*, which can identify NVM programming bugs using inter-procedural data flow analysis.

The main contributions of this work are:

- Identification and description of several programming patterns and anti-patterns used in NVM programs that can lead to crash-consistency problems or cause program performance issues.
- Description of a static program analysis using dataflow analysis for identifying many instances of these patterns and when to report bugs.
- Implementation of an open-source extension to LLVM for C/C++ applications that identifies and reports these bugs using dataflow analysis.
- Evaluation of *FlowNvm* on existing NVM programs and demonstrating *FlowNvm*'s capability to find and report previously known bugs and a new bug.

5.2 Programming Pattern Examples

In this section, we present the three programming patterns (*pair*, *durability*, *log*) and the two anti-patterns (*double flush*, *double log*). We discuss each pattern in detail and show how the misuse of the patterns can lead to errors in recovery. We then discuss each anti-pattern that can lead to program runtime performance degradation. We use the persistency model described in section 2.1.2. We use the examples from section 5.1 and expand on them to motivate the programming patterns.

5.2.1 Pair Pattern

The pair pattern ensures that writes do not persist out of order and enforces persistence ordering of writes between two memory regions in NVM. We denote the location of the first write as `data` and the location of the second write as `sentinel`. The `sentinel` acts as a consistency variable. The program code can read the `sentinel` to check if the `data` is up-to-date in NVM. Writes to `data` and `sentinel` are separated by a fence instruction and the writes are written back before the execution of the fence instruction.

Figure 5.1 depicts program code for a *log entry* example. For the purposes of the example, assume that the log entry is an undo log entry and is useful for rolling the state of the program data back to a consistent state. The program first updates the contents of the log entry (Line 6). The program then flushes the contents (Line 7) and persists the modifications (Line 8) instruction to explicitly write back the cache line. Afterward, the program modifies the `flag` field and explicitly writes the field back. We assume that `data` and `flag` are on separate cache lines.

The pair pattern demonstrates the necessity of ordering persistent writes. Otherwise, crash-consistency is not assured. Assume that the programmer makes a mistake and forgets to

```
1  //initially
2  //log_entry→ data=random data
3  //log_entry→ flag=false
4
5  //program code
6  fill(&log_entry→ data, contents);
7  clflushopt(&log_entry→ data);
8  sfence();
9  log_entry→ flag = true;
10 clflushopt(&entry→ flag);
11 sfence();
```

Figure 5.1 – Pair programming pattern with a log entry example.

flush the `data` field. The lack of the flush statement would not affect execution and can in fact improve performance. However, if the program crashes at the end of figure 5.1, then it is possible to get a failure after a restart.

`log_entry`	Cache	NVM
`data`	`contents`	`random data`
`flag`	`true`	`true`

Table 5.1 – A possible state at the time of power failure if `data` field of the log entry is not flushed explicitly.

Table 5.1 presents one possible state at the end of the snippet in figure 5.1 if the programmer omits the flush for the `data` field (Line 7 in figure 5.1). Forgetting to flush can lead to a hard-to-reproduce error during recovery after a restart. At the power failure, the caches contain updated fields. However, the NVM state does not reflect the cache state as the entries in the cache have not propagated back to NVM. The `data` field can contain random data that can lead to unexpected behavior.

The challenge of identifying these types of bugs arises from the fact that the crash is non-deterministic, as not every execution leads to the same bug. We use the axioms described in section 2.1.2 to ensure that the persistent writes are separated.

Pair Pattern Summary: The writes to `data` and `sentinel` must be separated by using write-back or non-temporal move instructions followed by a fence.

5.2.2 Durability Pattern

The durability pattern ensures that atomic updates to memory regions are properly persisted. The durable location is denoted as `object` that will be reachable through a location called `pointer`. The program first needs to make the `object` durable, then the program can persist the `pointer`. It is similar to pair pattern (Section 5.2.1) as durability pattern also requires ordering between two writes. However, the main difference is the intention of the programmer, which is to capture pointer updates. Moreover, the durability pattern is optimized for pointer assignments, as explained in section 5.4.

```
1  //class node; //data, next fields
2
3  //program code
4  node* newnode = nvmalloc(sizeof(node));
5  //newnode contains random data
6  fill(newnode, contents);
7  clflushopt(newnode);
8  sfence();
9  head = newnode;
10 clflushopt(&head);
11 sfence();
```

Figure 5.2 – Durability programming pattern with a linked list example.

Figure 5.2 depicts the program code for a singly-linked list example. The program first creates a new node and updates the node entry. The program then flushes the `node` using (`clflush-opt`) followed by the `sfence` instruction. The `newnode` is durable, then it can be safely used. Afterward, the program links the `newnode` to the linked list.

The durability pattern is necessary for conducting atomic updates. Otherwise, after a crash, the `pointer` can refer to a region that contains invalid data. Assume that the programmer makes a mistake and forgets to flush the `newnode` (Line 7). Assume that `newnode` and `head` are in different cache lines. If the program crashes at the end of the execution of figure 5.2, the linked list can contain a node with random data.

Element	Cache	NVM
`newnode`	`contents`	`random data`
`head`	`newnode`	`newnode`

Table 5.2 – A possible state at the time of power failure if `newnode` is not flushed explicitly.

Table 5.2 presents one possible state at the end of the execution of the snippet from figure 5.2, if the programmer forgets to flush `newnode` explicitly (Line 7). The cache reflects the correct state similar to the pair pattern, however, NVM state is not up-to-date.

Durability Pattern Summary: For atomic-updates, updates to the `object` must be durable before the updates to the `pointer`.

5.2.3 Log Pattern

Both pair pattern (section 5.2.1) and durability pattern (section 5.2.2) use primitives (e.g., cache line flush, fence instructions) for NVM programming. The log pattern is a higher-level pattern specifically tailored for libraries that rely on write-ahead, undo logging transactions, such as PMDK. The log pattern is useful for ensuring that the programmer logs memory regions in the program code within transactions. The program uses in-place updates to update the memory regions and a write-ahead log for crash-consistency.

We can show the log pattern with a linked list similar to the durability pattern example (section 5.2.3). The difference is that the program modifies an existing node using a transactional region and an interface similar to PMDK [141].

```
1  //tx_log(void*, size_t);
2  //class node; //data, next fields
3  //node spans multiple cache lines
4
5  //program code
6  tx_beg();
7  {
8      tx_log(oldnode, sizeof(node));
9      fill(&oldnode→ data, contents);
10 }
11 tx_end();
```

Figure 5.3 – Log programming pattern with a linked list example.

Figure 5.3 depicts the program code for transactional node update. The program initiates a transactional region by calling `tx_beg` (Line 6) and terminates the transactional region by calling `tx_end` (Line 11). The crash-consistency mechanism expects the programmer to track the memory regions by calling `tx_log` function on the modified memory regions.

The log pattern identifies writes to memory regions that are not logged. Assume that the programmer makes a mistake and forgets to call the `tx_log` function. The lack of logging can improve performance but can lead to failure similar to the previous patterns. If the program crashes at the end of the execution of figure 5.3, then it is possible to fail upon a restart.

Struct	Cache	NVM
undolog	-	-
node->data	contents	partial write

Table 5.3 – A possible state at the time of power failure if `node` is not tracked explicitly.

Table 5.1 presents one possible state at the end of the execution of the snippet from figure 5.3, if the programmer forgets to log the `node` explicitly (Line 8). There would be no undo log entry for the memory region. Therefore if a crash occurs at the end of the transaction, the node can be partially written. However, there is no way to completely roll back the node, as its old value is not stored elsewhere.

Log Pattern Summary: Within a transactional code region, the system must log memory regions before modifying them.

5.2.4 Double Flush Anti-Pattern

A program can flush the same memory region multiple times, which is unnecessary and can affect performance. We use an example similar to the one from the pair pattern in section 5.2.1.

```
//initially
//log_entry→ data=random data
//log_entry→ flag=false

//fill function flushes updated memory locations

//program code
fill(&log_entry→ data, contents); //flush
clflushopt(&log_entry→ data); //double flush
sfence();
```

Figure 5.4 – Double flush anti-pattern with log entry example.

Figure 5.4 depicts an example of double flush anti-pattern. In this particular example, the `fill` function, which mutates log entry data, flushes all modified cache lines.

Double Flush Anti-Pattern Summary: The same memory region should not be flushed multiple times without any modification in-between.

5.2.5 Double Log Anti-Pattern

A program can log the same memory region multiple times in a transactional system, which is unnecessary and can degrade runtime performance. We use an example similar to the log pattern in section 5.2.3.

```
//tx_log(void*, size_t);
//class node; //data, next fields
//node spans multiple cache lines

//program code
tx_beg();
{
    tx_log(oldnode, sizeof(node)); //first log of data
    if(...){
        ...
        tx_log(&oldnode→ data, ...); //double log of data
        fill(&oldnode→ data, contents);
    }
}
tx_end();
```

Figure 5.5 – Double log anti-pattern.

Figure 5.5 shows a double log anti-pattern example. In this particular case, the program first logs the entire `oldnode` (Line 8), then logs `data` field again unnecessarily (Line 11).

Double Log Anti-Pattern Summary: The same memory region should not be logged multiple times within a transaction.

5.3 FlowNvm Overview and Design

In this section, we formalize the patterns and the anti-patterns (Section 5.2) using a toy programming language. We formalize the data flow frameworks for each pattern: (1) pair, (2) durability, (3) log, (4) double flush, and (5) double log. Finally, we formalize the rules that are useful for reporting violations of the programming patterns (pair, durability, log). We use the ordering axioms as specified in section 2.1.2 for a Linux machine with *x86-64* ISA.

v, v_1, v_2	$\in$ Binding to memory locations in NVM
i, i_1, i_2	$\in$ Immediate values
u, u_1, u_2	$\in$ Whole numbers
op_{cmp}	$\in$ Comparison operators
s, s_1, s_2	$\in$ Statements
g, g_1, g_2	$\in$ Identifiers
f, f_1, f_2	$\in$ Functions
P	$\in$ NVM program

$v :=$	$v \mid \&v \mid v[u_1 : u_2]$
$z_1 :=$	$w(v,i) \mid w(v_1,v_2) \mid r(v) \mid w(v)$
$z_2 :=$	$w_n(v,i) \mid w_n(v_1,v_2) \mid r_n(v) \mid w_n(v)$
$z_3 :=$	$w_p(v_1,v_2) \mid w_{pn}(v_1,v_2) \mid w$
$z_4 :=$	$v = a(i) \mid d(v)$
$z_5 :=$	$f(v) \mid f_o(v) \mid p_f() \mid c(v)$
$z_6 :=$	$t_b() \mid t_e() \mid t_l(\mathrm{v})$
$z :=$	$z_1 \mid ... \mid z_6$
$l :=$	$op_{cmp}(v,i) \mid op_{cmp}(v_1,v_2) \mid op_{cmp}(i_1,i_2)$
$m :=$	`if(`l`)` then s_1 {else s_2} | `while(`l`) do` s
$s :=$	$s_1\{s_2,...,s_k\} \mid z \mid m$
$f :=$	$g(\{g_1,...,g_k\})\{s\}$
$P :=$	$f_1\{f_2,...,f_k\}$

Figure 5.6 – Context-free grammar rules for the imperative language.

To simplify the formalization, we use a simple language as described formally in figure 5.6. We keep the notation from section 2.1.2. The toy language contains NVM programming primitives and high-level transactions. We assume that all memory is allocated on NVM. Figure 5.7 depicts an example for each pattern and anti-pattern.

The language contains names that are bound to memory regions. Memory region can be of any size and the read or write operations affect the entire range. The internal parts of the

```
$w(v_1)$
$f_o(v_1)$
$p_f()$
$w(v_2)$
$f_o(v_2)$
$p_f()$
```

(a) Pair.

```
$w(v_2)$
$f_o(v_2)$
$p_f()$
$w_p(v_1, v_2)$
$f_o(\&v_1)$
$p_f()$
```

(b) Durability.

```
$t_b()$
$t_l(v)$
$w(v)$
$t_e()$
```

(c) Log.

```
$w(v[0:u])$
$f_o(v[0:u])$
$if(...)$
$then$
  $f_o(v)$
  $p_f()$
```

(d) Double flush.

```
$t_b()$
$if(...)$
$then$
  $t_l(v[0:u])$
$t_l(v)$
$w(v)$
$t_e()$
```

(e) Double log.

Figure 5.7 – Example snippets for each pattern in the toy language.

memory region can be accessed by the array notation. A binding v can be considered as a C/C++ pointer or an object variable in Java.

Read and write operations to regions pointed by v use $r(v)$ and $w(v)$ respectively. These operations use the cache hierarchy. Non-temporal move equivalents are r_n and w_n. Intuitively, $w(v_1, v_2)$ acts as a `memcpy` operation and copies v_2's region to v_1.

For convenience, we also define a pointer update w_p and its non-temporal move equivalent w_{pn} to update bindings. $w_p(v_1, v_2)$ updates the binding of v_1 to v_2 and both point to the same memory region initially pointed by v_2. We provide the (&) operator to get a reference to the binding and executing $w(\&v)$ changes v to point to a new location. In C/C++, it is the address operator. The following relation holds: $w_p(v_1) = w(\&v_1)$.

We provide a single parameter version of these functions w, w_n, w_p, w_{pn} to emphasize only the write operation for the first argument of the function.

The language provides memory allocation using $v = a(i)$, which takes a size argument and binds the memory region to v. We deallocate the memory region pointed by v using $d(v)$. A memory region can be within a single cache line or can span multiple cache lines.

We denote asynchronous flush (e.g., `clflushopt` or `clwb`) as f_o, synchronous flush (e.g., `clflush`) as f, non-temporal move instruction as w_n, fence instruction (e.g., `sfence`) as (p_f). p_f persists all the previously executed asynchronous flush and non-temporal move operations.

The toy language also provides high-level transactional instructions. We denote the beginning of the transaction as t_b , the end of the transaction as t_e, and the transaction logging mechanism to track a memory region as t_l. For nested transactions, we consider the code region spanned by the outermost transaction. We solely impose the rule that a memory region should be logged before a write to the memory region.

The cache line function $c(v)$ returns the set of cache lines for a given memory region pointed by v. For the memory region pointed by v, asynchronous flush f_o and synchronous flush f flush the set of all cache lines $c(v)$ and log t_l tracks the entire memory region.

5.3.1 Dataflow Framework Formalization

In this section, we define dataflow analysis rules for the toy language (figure 5.6). The dataflow framework is useful for finding the violation of programming patterns and the existence of anti-patterns in NVM programs. The dataflow construction is suitable for inter-procedural, context-sensitive, and flow-sensitive analysis [151]. We build the lattice and the framework as both monotone and distributive.

The dataflow framework is $D = (G, D, V, M, F, I)$, where G is the program graph, D is the direction of the analysis, V is the set of dataflow values, M is the meet operator, F is the set of transfer functions, and I is the initial value of the dataflow variables.

Meet Operator: The meet operator is merges state information across distinct paths for dataflow analysis. Dataflow analysis defines state variables, which keep track of information over a program. Dataflow analysis executes over all paths of a program and it is necessary to keep track of the state of a dataflow variable across all paths. When the analysis comes to a branch instruction, the dataflow analysis forks the state of the variable across different paths in the control-flow graph. The analysis updates variables along each path. When the paths join, the analysis uses the meet operator to merge forked versions of a variable into a single state.

We define a dataflow framework for each pattern and anti-pattern. We provide the intuition behind the construction and provide a sketch of the proof for each dataflow framework. We finally provide the rules for reporting NVM bugs. For each framework, we use D as the forward direction and I as $\top$ (Unknown) and meet operator as intersection $M_\wedge$. For keeping track of nested transactions, we use $M_{min} = min$ operator, which takes the minimum number. For simplicity and distributivity, we build the lattice as a linear chain. For this reason, in the general case, we define the values V and the transfer functions F for each framework and provide the value ordering explicitly.

We present all the dataflow frameworks for finding NVM bugs. These frameworks are (1) pair framework (D_{pair}), (2) durability framework (D_{dur}), (3) log framework (D_{log}), (4) double flush framework (D_{dflush}), and (5) double log framework (D_{dlog}). The complete dataflow framework D is the union of all the frameworks ($D = \{D_{pair}, D_{dur}, D_{log}, D_{dflush}, D_{dlog}\}$).

5.3.2 Pair Framework

We define the pair framework (D_{pair}) as given below:

Dataflow Values $State_P \in \{Write_P, Flush_P, Fence_P, \top_P\} = V_P$

Value Ordering $Write_P < Flush_P < Fence_P < \top_P$

Transfer Functions F_P (figure 5.8)

Pair Framework $D_{pair} = (V_P, F_P)$

$$\frac{v = State_P \qquad w(v)}{v = Write_P}\;(\mathbf{P-write}) \qquad \frac{v = State_P \qquad w_n(v)}{v = Flush_P}\;(\mathbf{P-ntm})$$

$$\frac{v = State_P \qquad f(v)}{var = Fence_P}\;(\mathbf{P-clflush}) \qquad \frac{v = State_P \qquad f_o(v)}{var = Flush_P}\;(\mathbf{P-clflushopt})$$

$$\frac{v = Flush_P \qquad p_f()}{v = Fence}\;(\mathbf{P-sfence_1})$$

$$\frac{v = State_P \qquad State_P \neq Flush_P \qquad p_f()}{v = State_P}\;(\mathbf{P-sfence_2})$$

Figure 5.8 – F_P: Dataflow transition rules for keeping track of persistence state.

Figure 5.8 defines the transition functions to track persistence state. Intuitively, write $w(v)$ updates the to $Write_P$ state, asynchronous flush $f_o(v)$ updates the location state to $Flush_P$ (in-flight, not in NVM yet), fence $p_f()$ ensures all previous flushes complete by updating the state to $Fence_P$, synchronous flush $f(v)$ flushes and fences, non-temporal move $w_n(v)$ updates the location and puts the location to waiting state $Flush_P$ (in-flight, not in NVM yet).

```
//initially
//log_entry→ data=random data
//log_entry→ flag=false

//program code
fill(&log_entry→ data, contents); //data=Write
if(...){
    clflushopt(&log_entry→ data); //data=Flush
    sfence(); //data=Fence
} //Meet: data=Write, data=Fence −→ data=Write
log_entry→ flag = true; // Pair bug: write to flag when data=Write
```

Figure 5.9 – Meet example for the pair framework and depiction of the *pair bug*.

We use the ∧ operator as a meet operation to take the lower element in the lattice. Intuitively, when the meet operation merges different paths, dataflow picks the path with the lower state value in the lattice. Figure 5.9 shows an example meet operation for pair framework. The state of `data` field is persisted in the `then` path but not in the `else` path of the `if` construct. Therefore, it is necessary to find the path where a write is not persisted. Since $Write_P < Fence_P$, merge across paths can find a write that is not persisted.

Sketch of distributivity proof: For **P-write, P-ntm, P-clflush, P-clflushopt**, the function f

always leads to a single state output, therefore these functions are distributive for the lattice. For **P-sfence**, a meet operation for any state other than $Flush_P$ is idempotent, therefore the equality holds. If one of the states is $Flush_P$, doing the P-fence rule advances the state to $Fence_P$. If the other state is lower than $Flush_P$, the end result is the lower state for both sides of the equality. If both states are $Flush_P$, the equality holds trivially. If the other state is greater than $Flush_P$, then the result is $Fence_P$ and the equality holds.

We define two relations for bug violations. The first one is the pair relation r_p, which forms a pair between two different memory regions pointed by v_1 and v_2. We show this relation as $r_p(v_1, v_2)$, where $v_1 \neq v_2$. The pair relation is symmetric and the rules that apply to v_1 also apply in reverse order to v_2. We also define a sentinel relation r_s for annotating the memory region as sentinel. We show this relation as $r_s(v_2)$, where v_2 is the sentinel location in a pair relation $r_p(v_1, v_2)$.

$$\frac{v_2 \in \{Write_P, Flush_P\} \qquad r_p(v_1, v_2) \qquad c(v_1) \neq c(v_2) \qquad (w(v_1) \vee w_n(v_1) \vee f(v_1) \vee f_o(v_1))}{\textbf{Pair bug}}$$

$$\frac{v_1 \neq Fence_P \qquad r_p(v_1, v_2) \qquad r_s(v_2) \qquad c(v_1) \neq c(v_2) \qquad (w(v_2) \vee w_n(v_2) \vee f(v_2) \vee f_o(v_2))}{\textbf{Sentinel bug}}$$

$$\frac{v \neq Fence_p}{\textbf{End of program bug}}$$

Figure 5.10 – Pair pattern violation bugs.

Figure 5.10 provides the violation rules for the pair programming pattern. **Pair bug** reports cases where two persistent writes are not separated by a fence. Figure 5.9 shows an example case where `data` field is in $Write_P$ state and there is a write to `flag` field.

We also report other pattern violations, such as the **sentinel bug** and **end of the program bug**. The end of the program ensures that all the NVM memory regions pointed by bindings (e.g., v) are flushed. We provide the sentinel bug to identify sentinel relation and ensure that no writes occur to the first memory region before the second memory region. Sentinel bug does not capture all the possible cases. Meet operation for `data=`$Fence_P$ and `data=`$\top_P$ leads to false negatives as meet operation misses unknown conditions ($\top_P$). The main goal of the pair pattern is to ensure that two persistent writes are separated by using write-back or non-temporal move instructions followed by a fence instruction.

5.3.3 Durability Framework

We define the durability framework (D_{dur}) as given below:

Dataflow Values $State_P \in \{Write_P, Flush_P, Fence_P, \top_P\} = V_P$

Value Ordering $Write_P < Flush_P < Fence_P < \top_P$

Transfer Functions F_P (figure 5.8)

Durability Framework $D_{dur} = (V_P, F_P)$

We use the same framework from section 5.3.2 to keep track of the persistence of variables. The only difference is the pattern violation rules. We define $r_d(v_1, v_2)$ as the durability relation. v_1 is the *pointer* variable that is updated and v_2 variable points to a durable object o. The same proofs for distributivity also hold for the durability framework.

$$\frac{v_2 \neq Fence_P \qquad r_d(v_1, v_2) \qquad (w_p(v_1) \vee w_{pn}(v_1))}{\textbf{Durability bug}}$$

Figure 5.11 – Durability pattern violation bug.

Figure 5.11 provides the violation rules for the durability programming pattern. The atomic-update operation should result in a new binding for v_1, where v_1 points to the memory region pointed by v_2, which is persisted properly ($Fence_P$ state).

```
//class node; //data, next fields

//program code
node* newnode = nvmalloc(sizeof(node)); //newnode=Unk
//newnode contains random data
fill(newnode, contents); //newnode=Write
if(...){
    clflushopt(newnode); //newnode=Flush
    sfence(); //newnode=Fence
} //Meet: newnode=Write, newnode=Fence −→ newnode=Write
head = newnode; //Durability bug, updating next when newnode=Write
```

Figure 5.12 – Meet example for the durability framework and depiction of the *durability bug*.

Figure 5.12 shows an example case for the **durability bug**. Durability bug reports cases where a pointer can point to a memory region that is not persisted properly. In the example, `newnode` is not persisted properly in the `else` path of the if construct.

5.3.4 Log Framework

We define the log framework (D_{log}) as given below:

Dataflow Values For Log $State_L \in \{Log_L, \top_L\} = V_L$

Value Ordering For Log $\top_L < Log_L$

Meet Operator For Log $M_\wedge = \wedge$

Dataflow Values For Transaction Counting $State_T \in \{0, \ldots, N\} = V_T, n$ is a finite integer

Value Ordering For Transaction Counting $0 < \ldots < i < \ldots < N$

Meet Operator For Transaction Counting $M_{min} = min$

Dataflow Values $V_M = V_T \cup V_L$

Meet Operator $M_M = M_\wedge \cup M_{min}$

Transfer Functions F_L (figure 5.13)

Log Framework $D_{log} = (V_M, F_L, M_M)$

We assume the ordering to be $\top_L < Log_L$ where the $\top_L$ element is the *greatest lower bound* and Log_L is the *least upper bound.*

$$\frac{v = State_L \qquad t_l(v)}{v = Log_L}\ (\mathbf{L-txlog}) \qquad \frac{v_T = State_T \qquad t_b()}{v_T = min(State_T + 1, N)}\ (\mathbf{T-txbegin})$$

$$\frac{v_T = State_T \qquad t_e()}{v_T = max(State_T - 1, 0)}\ (\mathbf{T-txend})$$

Figure 5.13 – F_L: Dataflow transition rules for keeping track of logging state.

Figure 5.13 defines the transition functions to track log state and transaction state. There is a global v_T variable that counts the nesting of the transactions, where the maximum number of allowed nesting is denoted by N. Intuitively, write $t_l(v)$ updates the memory region to Log_L state, transaction begin $t_b()$ increases the transaction counter v_T and transaction end decreases the counter.

We use the $\wedge$ operator as a meet operation to take the lower element in the reversed lattice. Figure 5.14 shows the meet operation for the log framework. The state of `data` field is logged in the `then` path but not in the `else` path of the `if` construct. Therefore, it is necessary to find the path where a write is not logged prior. Since $\top_P < Log_P$, merge across paths can find a write that is not tracked.

Sketch of distributivity proof: For **L-txlog**, the function f always leads to a single state Log_L, therefore the lattice for keeping track of the logging state is distributive. Transaction count tracking lattice is also a linear chain and is distributive.

```
//tx_log(void*, size_t);
//class node; //data, next fields
//node spans multiple cache lines

//program code
tx_beg(); //vT=1
{
    if(...){
        tx_log(oldnode, sizeof(node)); //data=Log
    } //Meet: data=Unk, data=Log −→ data=Unk
    fill(&oldnode→ data, contents); //Log bug: write to data is not tracked
}
tx_end(); //vT=0
```

Figure 5.14 – Meet example for the log framework and depiction of the *log bug*.

$$\frac{v \neq Log_L \qquad v_T > 0 \qquad (w(v) \vee w_n(v))}{\textbf{Log bug}} \qquad \frac{v_T = 0 \qquad (t_l(v) \vee w(v) \vee w_n(v))}{\textbf{Outside transaction bug}}$$

Figure 5.15 – Log pattern violation bugs.

Figure 5.15 provides the violation rules for the log programming pattern. **Log bug** reports cases where a write is not tracked by the crash-consistency mechanism. Figure 5.14 shows an example case where `data` field is not tracked before a write. We track writes to memory regions outside a transaction using **outside transaction bug**.

5.3.5 Double Flush Framework

We define the double flush framework (D_{dflush}) as given below:

Dataflow Values $State_F \in \{Flush_F, Write_F, \top_F\} = V_F$

Value Ordering $Flush_F < Write_F < \top_F$

Transfer Functions F_F (figure 5.16)

Double Flush Framework $D_{dflush} = (V_F, F_F)$

Figure 5.16 defines the transition functions to track double flush state. Intuitively, we only need to keep track of writes and flushes as we are only interested in double flushes, which have no modifications in between. The state transitions are similar to the pair pattern as described in section 5.3.2. The main difference is the lack of fence instructions and the ordering of the states (V_F).

$$\frac{v = State_F \qquad w(v)}{v = Write_F}\ (\mathbf{F-write}) \qquad \frac{v = State_F \qquad f(v)}{var = Flush_F}\ (\mathbf{F-clflush})$$

$$\frac{v = State_F \qquad f_o(v)}{v = Flush_F}\ (\mathbf{F-clflushopt})$$

Figure 5.16 – F_F: Dataflow transition rules for keeping track of double flush state.

```
//initially
//log_entry→ data=random data
//log_entry→ flag=false

//fill function flushes updated memory locations
//for this example

//program code
if(...){
    fill(&log_entry→ data, contents); //data=Flush
}//Meet: data=Flush, data=Unk −→ data=Flush
clflushopt(&log_entry→ data); //Double flush bug: flushing already flushed data
sfence();
```

Figure 5.17 – Meet example for the double flush framework and depiction of the *double flush bug*.

Figure 5.17 shows an example meet operation for double flush framework. Compared to the pair framework (in section 5.3.2) and the durability framework (in section 5.3.3), flush is in a lower state ($Flush_F < Write_F$) to capture flush across different paths. The state of `data` field is flushed in the `then` path but not in the `else` path of the `if` construct. Therefore, it is necessary to find the path where a flush occurred. Since $Flush_F < Write_F$, merge across paths can find the flush without any modification in-between.

Sketch of distributivity proof: For **F-write, F-clflush, F-clflushopt**, the function f always lead to a single state, therefore these functions are distributive for the lattice.

$$\frac{v = Flush_F \qquad (f(v) \vee f_o(v) \vee w_n(v))}{\textbf{Double flush bug}}$$

Figure 5.18 – Double flush anti-pattern bug.

Figure 5.18 provides the violation rules for the double flush anti-pattern programming pattern. **Double flush bug** reports cases where a memory region is flushed twice without any modifications in-between. Figure 5.17 shows an example case where `data` field is flushed in one path and then flushed again afterward.

5.3.6 Double Log Framework

We define the double log framework (D_{dlog}) as given below:

Dataflow Values For Log $State_L \in \{Log_L, \top_L\} = V_L$

Value Ordering For Log $Log_L < \top_L$

Meet Operator For Log $M_\wedge = \wedge$

Dataflow Values For Transaction Counting $State_T \in \{0, \ldots, N\} = V_T, n$ is a finite integer

Value Ordering For Transaction Counting $0 < ... < i < ... < N$

Meet Operator For Transaction Counting $M_{min} = min$

Dataflow Values $V_M = V_T \cup V_L$

Meet Operator $M_M = M_\wedge \cup M_{min}$

Transfer Functions F_L (figure 5.13)

Log Framework $D_{log} = (V_M, F_L, M_M)$

The framework is similar to the log framework as described in section 5.2.3. We use the normal ordering for V_L. We assume the ordering to be $Log_L < \top_L$ where the $\top_L$ element is the *least upper bound* and Log_L is the *greatest lower bound.* Figure 5.13 defines the transition functions to track log state and transaction state.

```
//tx_log(void*, size_t);
//class node; //data, next fields
//node spans multiple cache lines

//program code
tx_beg();
{
    if(...){ //vT=1
        ...
        tx_log(&oldnode→ data, ...); //data=Log
        fill(&oldnode→ data, contents);
    } //Meet: data=Log, data=Unk −→ data=Log
    tx_log(oldnode, sizeof(node)); //Double log bug: data is already logged
}
tx_end(); //vT=0
```

Figure 5.19 – Meet example for the double log framework and depiction of the *double log bug.*

Figure 5.19 shows an example meet operation for double log framework. The state of `data` field is logged in the `then` path. Therefore, it is possible to log `data` field twice within a transaction. The proofs for distributivity from the log framework also hold for the double log framework.

$$\frac{v = Log_L \qquad v_T > 0 \qquad t_l(v)}{\textbf{Double log bug}}$$

Figure 5.20 – Log pattern violation bugs.

Figure 5.20 provides exposes the double log bug. **Double log bug** reports cases where a memory region is logged twice within a transaction. Figure 5.19 shows an example case where `data` field is logged twice within a transaction.

5.4 FlowNvm Implementation

We implement *FlowNvm* as an extension to LLVM for C/C++ programs to identify bugs that can lead to crash-consistency errors and performance degradations using inter-procedural dataflow analysis. We designed *FlowNvm* with *x86-64* architecture and ordering rules (Section 2.1.2) in mind. We implement the dataflow analysis to be context-sensitive, flow-sensitive, and inter-procedural program analysis [151]. *FlowNvm* requires programmer annotations for analyzing NVM programs and checking specified properties.

In this section, we describe the design decisions such as the overall flow of the system, the annotation system that we use for the dataflow analysis, and the several assumptions that we use for building *FlowNvm*. We make several relaxations to the dataflow framework and forego soundness [116] due to challenges of modeling the heap.

5.4.1 FlowNvm Internals

FlowNvm consists of 5 dataflow frameworks, which are *pair, durability, log, double flush,* and *double log* frameworks. Each framework can be enabled or disabled individually. The dataflow analysis can report *pair, sentinel, end of program, durability, log,* and *outside transaction, double flush,* and *double log* bugs. Each bug can be enabled or disabled.

The programmer can use the *FlowNvm* tool as follows:

1. The programmer annotates the NVM program code with *FlowNvm* annotations.
2. The programmer compiles the program code (clang for C and clang++ for C++) to generate the *LLVM IR* code for each compilation unit.
3. The programmer combines each compilation unit with *llvm-link*.

4. The final whole program IR contains all the functions that can be analyzed.

5. The programmer runs the dataflow analysis, which uses the enabled frameworks for keeping track of programming patterns and reports the enabled bugs.

FlowNvm runs a separate analysis for each function specified by the programmer. Before running the dataflow analysis, *FlowNvm* does pre-processing. *FlowNvm* creates the necessary data structures to easily keep track of annotations during the dataflow analysis.

We minimize the overhead from memory region tracking depending on the framework enabled. Pair framework keeps track of type-annotated memory regions and alias analysis. Durability and double flush frameworks keep track of all possible memory regions solely based on alias analysis in NVM. Log and double log frameworks keep track of all possible memory regions based on the alias analysis and attribute information for NVM transactions.

5.4.2 FlowNvm Annotations

Annotation	Category	Explanation
`nvm_fnc`	Function	Allows a function to independently be unit-under analysis.
`nvm_cst(f)`	Function	Custom functions for improving bug reporting
`sentinel(d)`	Type	For pair framework, pairs two types of memory regions.
`dur_field`	Type	For durability framework, persistent pointer annotation.

Table 5.4 – *FlowNvm* annotations.

FlowNvm uses two categories of annotations, which are *function* annotations and *type* annotations. We use function annotations to keep track of functions that are essential for the dataflow analysis. All the frameworks use `nvm_fnc` to denote the function, which is the unit under analysis. We do not support virtual functions nor multi-threading for dataflow analysis. The dataflow analysis skips virtual function calls and thread creation calls. *FlowNvm* can start the analysis from the `main` function for whole-program analysis. It is possible to analyze a single function inter-procedurally. When starting the analysis, all the variables are assumed to be in an unknown state and there are no active transactions.

We also support custom annotations (`nvm_cst(f)`) for functions to extend the functionality of *FlowNvm*. For example, we can annotate flush functions, non-temporal move functions, and allocation functions for better bug reporting.

The second category of annotation is type annotations. We use the data structure type information for the annotation system. We observe that structural accesses such as field accesses are heavily used. When a programmer modifies memory regions, most of the time, the modifications are done through the indirection of objects instead of raw memory. For instance, `entry->data` represents the memory region for the `data` field inside a log entry object. Writes to this location update only the `data` memory region for the entry object. This insight greatly simplifies our design approach as we use layout as a reference to do the dataflow analysis.

We leverage the type and attribute information stored within structures (e.g., class or struct) to identify memory regions. There are several reasons for using the structure type and layout information as a basis for annotations. By annotating types, the programmer can avoid annotating each program code location where the program data in NVM is modified. The programmer needs to annotate a structure to convey ordering relations, which can reduce programmer effort. The annotation improves the programmer understanding of the functionality of the attribute and provides better documentation of the intention. *FlowNvm* does not keep track of the persistence state for variables that live on the stack.

We present the set of all annotations for each framework. We group 5 frameworks into 3 categories for memory tracking overhead. We run each category of frameworks individually in the experiments. We run durability and double flush frameworks together. We run log and double log frameworks together.

Pair Annotations

Figure 5.21 shows the annotations in practice for the pair framework. Pair framework uses the `sentinel` keyword to define a pair relation between two memory regions. The field annotated with `sentinel` is considered as the sentinel memory region. The annotation takes an argument, which is the name for the `data` memory region. `data` can be a field of an object or can be an object. The programmer specifies the namespace (if it exists), name of the type, and the field for the `data` region. When `data` is not specified as an argument, *FlowNvm* takes all the fields in the object as `data` region other than the `sentinel`.

```
typedef struct kp_vte_ {
    ...
    // sentinel([namespace::]type[::<field name>]) ds_state state;
    sentinel() ds_state state;
} kp_vte;

int nvm_fnc main(){
    ...
}
```

Figure 5.21 – Annotations for the pair framework.

We assume that `data` and `sentinel` are in different cache lines. *FlowNvm* warns of the possibility of being in the same cache line for fields within the same object using 64-byte alignment boundary and object layout. For a single object, dataflow analysis follows the rules that we specified in section 5.3.

The pair framework treats each field as a distinct memory region within an object. We explain memory region modeling in section 5.4.3 in detail.

As figure 5.9 demonstrates, field accesses directly update the dataflow variable corresponding to the memory region. Updating the `data` field for the log entry updates the corresponding dataflow variable `data` directly.

Durability/Double Flush Annotations

Figure 5.22 shows the annotations in practice for the durability framework. *FlowNvm* uses `dur_field` annotation to check pointer updates. We do not use the `sentinel` keyword to signify the difference in the intent of the programmer. If the pointer points to a location that is not persisted properly, the dataflow analysis produces a warning.

```
typedef struct kp_vte_ {
    ...
    dur_field const void *value;
    ...
} kp_vte;

int nvm_fnc main(){
    ...
}
```

Figure 5.22 – Annotations for the durability framework.

Both the durability and the double flush frameworks need to keep track of all memory regions, which might be in NVM. We rely on the alias analysis to model memory regions to deal with situations such as casting. During pointer assignments, we check the state of the memory region that is being assigned. If a programmer assigns a `nullptr` to a pointer variable, then *FlowNvm* assumes that the assignment is safe.

Log/Double Log Annotations

We tailor the design of the log and double frameworks for Intel's PMDK [141]. PMDK's libpmemobj library provides transactions. We hardwire the design of *FlowNvm* to support PMDK transactions and we support a subset of PMDK functionality for Linux. We support only begin transaction (`pmemobj_tx_begin`) and end transaction (`pmemobj_tx_end`) for transactions. We support logging (`pmemobj_tx_add_range`, `pmemobj_tx_add_range_direct`), allocations (`pmemobj_tx_alloc`, `pmemobj_tx_zalloc`, `pmemobj_tx_realloc`, `pmemobj_tx_zrealloc`), reads, writes and pointer translations (`pmemobj_direct`, `pmemobj_oid`). Transactional allocations log the newly created memory region, so *FlowNvm* assigns Log_L state to the dataflow variable for the log and double log frameworks during allocations.

Figure 5.23 depicts the log framework in practice. The transactional libpmemobj C library

```
1 int nvm_fnc btree_map_insert(...){
2     TX_BEGIN(pop) {
3         ...
4         TX_ADD(root); //logging
5         TOID(struct rectangle) rect = TX_NEW(struct rectangle); //allocation
6         ....
7     } TX_END
8     ...
```

Figure 5.23 – Annotations for the log and double framework.

heavily relies on macros. The compiler preprocesses the macros and translates the macros to PMDK function equivalents. Line 4 logs the `root` variable, while Line 5 allocates a `rect` variable. Log memory modeling is similar to pair memory modeling, which relies on both attribute and alias analysis information.

5.4.3 Memory Regions and Modeling

We make sweeping simplifications for the memory region modeling. *FlowNvm* analysis is not sound because of memory modeling. *FlowNvm* does not use size information for memory regions and relies on heuristics such as type information and memory addresses to make inferences. An object and its fields represent dataflow variables (e.g., for `obj->data`: `obj` and `obj->data`). We treat an array field as a single memory region.

Writing to an entire object (e.g., `memcpy`) updates all dataflow variables corresponding to the object, while writes to a field update the object variable and the corresponding field variable. Flushing or logging an entire object updates the dataflow variables corresponding to the object, while flushing or logging a field updates only the corresponding field variable. If all the fields are either written, flushed, or logged individually, then the object variable is set to the corresponding state. We expect accesses to sentinel variables to be explicit, so a sentinel is not included in object variable state computation. We optimize durability and double flush analysis by computing state only over an object dataflow variable and the annotated pointer fields.

We use out-of-the-box alias analysis for object memory regions from LLVM for *FlowNvm* memory region modeling. The goal is to identify whether two pointers point to the same memory region, which, in turn, requires a points-to analysis. For this reason, *FlowNvm* uses Steensgard alias analysis [166] provided by LLVM. If two pointers can point to the same memory region, then we treat the pointers as belonging to the same memory region. Likewise, the approach compresses the `phi` nodes to a single memory region. *FlowNvm* assumes that a pointer is not a self-referencing pointer (does not point to itself) and is not put the same alias set. The alias analysis is field-insensitive, context-insensitive, and flow-insensitive. The imprecision of the analysis leads to a high false-positive rate (Table 5.5). It is possible to use more precise semantics for memory modeling to provide a more precise alias analysis. For example, *FlowNvm* can rely on Agamatto's memory modeling from symbolic execution.

We extend the analysis to recognize persistent allocation function and transactional allocations in the benchmarks. For PMDK, we extend the analysis to recognize the libpmemobj library. We do it by taking advantage of `PMEMoid` object to represent it as a pointer. A `PMEMoid` object consists of a location ID of the pool object and the offset within the pool. We make a best-effort analysis to ensure that allocations done within a function do not alias with other allocations.

In our analysis, we do not differentiate between heap objects that are in DRAM or NVM. *FlowNvm* can rely on taint analysis to reduce memory tracking for cases where a memory region is in NVM. Values that originate from dynamic memory allocation in DRAM (e.g., `malloc`) would taint the pointers, and the analysis could then disable bug reports for tainted values. We do not support taint analysis in our approach, as the annotations imply that the memory regions are in NVM.

5.4.4 Bug Reporting and Custom Functions

In this section, we describe the optimizations that we use for bug reporting and the custom functions. Dataflow analysis reports only a single bug for a memory region or for a line of code to reduce the false-positive rate.

We make the observation that in NVM programming, flush and non-temporal operations are done with a specific API. For example, the Echo key-value store [54] uses `kp_flush_range` to flush and/or fence a given memory region. The main issue with such a function is that, inside the function, a `for` loop flushes multiple cache lines asynchronously at the same time which can lead to *double flush* bugs. In addition, the function takes an argument that usually specifies the memory region of interest. *FlowNvm* uses the information provided by the custom function annotations to improve precision and provide better bug reports.

The argument (`f`) of the `nvm_cst(f)` annotation can be `flush`, `flushfence`, `ntm`, `alloc`, and `skip`. We provide `flush` annotation to denote a function that only flushes a memory region, `flushfence` annotation to denote a function that flushes and fences a memory region, `ntm` annotation to denote a non-temporal move instruction. We provide `alloc` to denote that the allocation is persistent and the annotation provides information to the alias analysis. `skip` allows dataflow analysis to skip the function. We assume that the programmer can annotate these functions.

Functions that use custom annotations simplify pointer analysis. Custom function annotations can suppress double flush bugs and improve bug reporting. Figure 5.24 shows an example use case of `flushfence`. The example shows buffer flushing for PMFS and the annotation enables *FlowNvm* to not consider the internals of the function.

```
void nvm_cst(flushfence) pmfs_flush_buffer(void *buf, uint32_t len, ...){
    ...
    if (support_clwb) {
        for (i = 0; i < len; i += CACHELINE_SIZE)
            _mm_clwb(buf + i);
    ...

    sfence();
}
```

Figure 5.24 – Function to flush a range of memory.

5.5 Evaluation

We present our results on 3 NVM programs from the Whisper NVM benchmarking suite [126]. The NVM programs are (1) PMDK [84, 141]: a persistent memory library by Intel for persistent allocations and transactions, (2) Echo [19, 54]: a key-value store designed for NVM and (3) PMFS [53, 143]: a file system designed for NVM.

We implemented *FlowNvm* as an extension to LLVM for C/C++ programs. *FlowNvm* mainly checks for pair, durability and log patterns and double flush, double log anti-patterns as described in section 5.4. We found a new bug using the pair framework as shown in figure 5.26 and exposed 4 previously known bugs in NVM systems. We also introduced 12 additional synthetic bugs to show that the tool performs as intended.

We used the machine with the following configuration for evaluation. The machine used Intel i7-6700 CPU @ 3.40GHz with 2 Kingston 8GB DDR4 DRAM @ 2133 MHz running Linux Ubuntu 16.04. We used LLVM and Clang version 8.0.0 to compile the code. The machine did not have any NVM attached to it. We compiled each source file in each program to a bitcode file and then used llvm-link to get a whole program IR. We added `-fno-discard-value-names -ggdb -femit-all-decls` flags for better debug information. We added annotations to each program as described in section 5.4 to check properties using dataflow analysis and find NVM bugs.

5.5.1 Bug Finding Results

NVM Program	reported bugs/correct/false positives	num of annotations
Echo	13/1/12	20
PMFS	10/3/7	2
PMDK	33/5/28	8

Table 5.5 – NVM program breakdown of bug locations.

Table 5.5 breaks down the results for the 3 NVM programs. We showed the number of reported bugs by *FlowNvm*, the number of false positives, and the number of correctly found bugs. We also reported the number of annotations.

Echo assigns a state (dead, allocated, active) to each of its durable data structures. We also checked each data structure if there is a pointer to a persistent location. We did not support thread libraries. That is why, we started the dataflow analysis from the `worker_thread-_entrypoint` function. We checked fields that should be persistent such as `commit_record`. We also annotated the `kp_flush_range` funtion with `flushfence` annotation and defined a seperate version with `flush` annotation for program locations in the code that do not execute `sfence`. Our tool found one **new** bug for the Echo benchmark.

We started the analysis for PMFS from the `pmfs_xip_file_write` function that does a lot of NVM operations. We annotated the transaction part in PMFS. Our tool reported 3 previously found bug locations.

We did not support virtual methods and functions. That is why, we annotated each data structure function separately for PMDK instead of using the `main` function. For our case, we targeted insert and remove operations in PMDK data structures (*btree*, *rbtree*, and *hashmap_atomic*). Our tool reported 5 previously found bug locations.

Bug Type	reported bugs/correct/false positives	introduced/found
Pair	9/1/8	2/2
Durability	5/0/5	1/1
Double flush	9/3/6	3/3
Log	6/4/2	1/1
Double log	27/1/26	2/2
Outside transaction	0/0/0	3/3

Table 5.6 – Bug type breakdown of bug locations.

Table 5.6 breaks down the same results with respect to different bug types. The table also demonstrates that *FlowNvm* can find artificially introduced bugs separately from the reported bugs. Our tool reported, 9 pair bugs, 5 durability bugs, 9 double flush bugs, 6 log bugs, 27 double log bugs.

We manually introduced bugs to NVM programs by (1) removing flush, (2) adding flush, (3) adding write, (4) adding log, (5) removing log, and (6) removing transaction begin similar to the process described in PMTest [115]. If the newly introduced bug created new bug locations and the tool correctly finds the bug, we stated that the synthetic bug is found. Our tool found all the manually introduced bug cases.

Most of the reported bugs are false positives. The main reason for false positives is the lack of precision in the alias analysis, especially in the log framework. Figure 5.25 shows a false positive from log framework, where alias analysis does a poor job. The program, in order to maintain red-black tree properties, logs, and mutates the color fields of both the parent and the grandparent nodes. The alias analysis is not precise enough to differentiate parent from

```
TOID(struct tree_map_node) rbtree_map_recolor(...){
    ...
    TX_SET(NODE_P(n), color, COLOR_BLACK);
    TX_SET(NODE_GRANDP(n), color, COLOR_RED);
    ...
}
```

Figure 5.25 – Example for red-black tree alias analysis issue.

grandparent, therefore, the alias analysis assumes these two locations are the same. We believe that a more precise alias analysis or a better heap modeling can be beneficial for our approach. Classical symbolic execution [27] can be used for precise heap modeling, however, introducing symbolic execution can incur computation and resource overhead due to problems such as path explosion and keeping track of program state.

There are other reasons for false positives. For Echo, if the program has a condition that leads to failure, Echo destroys the data structures and sets the data structure state to *DEAD* without considering ordering constraints. This is correct as there are only two possible conditions. If the state is *DEAD*, the other parts of the data structure is irrelevant for the recovery code and if the data structure is not set to *DEAD*, the operation can continue as normal. For PMFS, the tool gets a superblock virtual address using *pmfs_get_block* function, but we do not assume anything about the persistence of the return value.

Real Bugs

Bug Type	Program	File	Reported lines	Description
Pair (New)	Echo [163]	kp_kvstore.c	1707	Missing dereference
Double flush	PMFS [87]	xip.c	218, 273	Flush same buffer twice
Double flush	PMFS [83]	journal.c	135	Flush log entry twice
Log	PMDK [86]	btree_map.c	192, 200, 203,205	Node not logged
Double log	PMDK [85]	btree_map.c	270	Node logged twice

Table 5.7 – Correctly reported bug locations.

In this section, we provide the breakdown of real bugs and explain the behavior of the analysis. Table 5.7 summarizes the reported bug locations that are correct. We found one new bug in Echo.

Figure 5.26 summarizes the newly found bug. Echo has a version table entry data structure to keep each version of data buffers. The function creates a virtual table entry, where the entry is a pointer to pointer. The function modifies the entry and then flushes it. However, the program does not dereference the pointer to pointer object. The actual pointer for the object requires a dereference. Our tool warns the programmer and provides the location telling that before writing to state at line 1707 (Line 3 in figure 5.26), the `vte` object is not flushed correctly.

```
int kp_vte_create(kp_vte **vte, ...){
    ...
    kp_flush_range((void *)vte, ...);
    PM_EQU(((*vte)→ state), (STATE_ACTIVE));
    kp_flush_range((void *)&((*vte)→ state), ...);
    ...
}
```

Figure 5.26 – New bug found in Echo key value store.

The bug fix is incorporated to Echo key-value store [163].

The remaining 8 bug locations were previously fixed in 4 bug-fix commits. We exposed these bugs using *FlowNvm* as well. *FlowNvm* provides information about the previous location of pair, double flush, and double log. One thing to note is that *FlowNvm* can also provide trace information. Using the previous location information, the programmer can assess, whether a bug location in the report is reasonable. The tool correctly reports all the previous locations for all cases except the bug report for double log in `btree_map.c`. Instead of showing line 359, the tool reports line 168 in `btree_map.c`. The imprecision in the alias analysis is responsible for this effect.

5.5.2 Performance Evaluation

We report the time it takes to run the dataflow analysis on NVM programs. The configuration is the same as in the previous section. We measure each framework separately as the programmer does not have to use all of the frameworks at the same time.

Framework	NVM Program	Time (s)
Pair	Echo	1.82
Pair	PMDK:hashmap_atomic	0.03
Durability/Double flush	Echo	0.59
Durability	PMFS	0.18
Log/Double log	PMDK:btree	0.09

Table 5.8 – Time measurement for bug reporting.

Table 5.8 summarizes the runtime result of each framework averaged over 10 runs. As described in the implementation section, we ran double flush with durability framework and log framework with double log framework. The analysis is very fast given that testing and symbolic execution takes around minutes to hours. The majority of the time is spent in dataflow analysis with the rest of the time spent in dataflow data structure initialization. The analysis completes under a second except for pair framework for Echo, which has overheads due to pair matching and size of the unit of analysis.

5.6 Related Work

FlowNvm uses inter-procedural dataflow analysis to find patterns that can lead to crash-consistency bugs and performance degradation. In this section, we compare *FlowNvm* to other bug finding work for NVM programs.

5.6.1 Program Analysis and Testing for NVM

Other work for finding bugs in NVM programs uses runtime profiling and testing [82, 104, 114, 115, 132, 142] and symbolic execution [129].

Yat [104] is a general testing framework designed for Intel PMFS. Yat tracks all writes, flushes, and fences and divides the program profile into segments delimited by fences. It tests all possible reorderings within a segment and tests the recovery code, which is costly as it uses exhaustive reordering.

Oukid et al. [132] implement a framework for testing NVM programs using an old persistency model with `clflush` and `mfence`. The testing framework keeps a copy of the durable heap by replicating the flushes and copying data at each fence. While the program is executing, the framework randomly simulates a power failure crash. Upon crash, the testing framework pauses the main program and forks a testing process. The testing program can also crash, which again leads to another fork from the testing program that is created from the original testing program. The cases where recovery fails are reported to the user.

Intel developed pmemcheck [142], pmreorder [144] and Inspector Persistent Memory Debugger [82] specifically to find NVM programming bugs in programs using PMDK. The pmemcheck [142] tool is built on top of Valgrind. The pmemcheck tool profiles the program and creates a trace of stores, cache line flushes, and fences. The trace allows pmemcheck to report writes that are not properly persisted to NVM, writes that are not tracked inside transactions, writes to the same memory region multiple times without persist instructions in-between, double flush without any modification in between, and logging the same memory region in different transactions running concurrently in different threads (PMDK transactions require explicit locking for isolation).

The pmreorder [144] tool uses the trace information from pmemcheck [142]. The pmreorder [144] tool parses the trace information and reorder writes to NVM within fence-delimited regions. The user of pmreorder provides a program that checks the data structure consistency depending on the reorderings.

Intel Inspector Persistent Memory Debugger [82] is another runtime tool that provides a GUI. Inspector can find unnecessary cache line flushes, undo logging issues inside a transaction, missing persist instructions, and reordering bugs. Inspector is specifically tailored for PMDK and provides a runtime API that allows the programmer to annotate the code with phases. The phases allow the programmer to provide info to Inspector to know when a crash can occur and which code gets executed after a restart. Compared to the Intel tools, we can find similar bugs in using static analysis.

PMTest [115] is a runtime profiling tool that requires the developer to put PMTest logic in the program code for PMTest to find bugs. PMTest uses an interval tree to keep the state of memory address ranges, whether an address range is written, flushed, or made durable similar to *FlowNvm* design. The PMTest API is similar to assertions and PMTest can check two properties. The first property is whether a memory region is persisted before another memory region, which is similar to the pair pattern. The second property is persistence that checks if a memory region is persisted at a given program location. Using the two basic low-level properties, PMTest can report missing flush, missing fence, double flush errors. PMTest is also extended for PMDK to report missing log, double log, and incomplete transactions. Most NVM programming updates are done locally and PMTest requires the programmer to annotate the entire program code. *FlowNvm* annotates types and does not require annotations across the entire program. Determining this information for an entire program is challenging. The main benefit of *FlowNvm* is that the analysis focuses on types, so the developer only focuses on the correct use of the objects instead of worrying about each statement.

RECIPE [108] simplifies the crash-simulation methodology proposed by earlier systems minimizing the number of crash test locations. The testing methodology starts by running a workload for a program, crashes the program with a low probability after an atomic store, runs the recovery code, and checks the correctness of key-value pairs after recovery.

XFDetector [114] considers the interleaving between program execution and the recovery code. The programmer uses the XFDetector API to mark regions for analysis and compiles the NVM program. XFDetector adds a crash location before each fence operation. XFDetector keeps a shadow memory for NVM program data in the form of a simple state machine similar to PMTest [115]. XFDetector checks the state machine to see whether recovery uses the correct version of the program data and reports the cases where it fails.

Witcher [63] uses static analysis to infer properties about the program, to avoid programmer annotations, but still relies on testing to find bugs. Witcher automatically infers test cases for a data structure that use its API and uses these test cases for validation. Witcher then traces the write, flush, and fence instructions to create NVM program data for inferring program invariants. Witcher uses program dependency analysis and the NVM program data image to infer ordering constraints between memory regions such as persisting the memory regions in conditional expressions before the writes to the memory regions in the body of the conditional statement. Witcher reorders instructions to create possible crash states and uses the test cases to check whether there are any outputs that violate the invariants.

Agamotto [129] uses symbolic execution to exhaustively detect missing flush/fence and extra fence/flush instructions. Agamotto uses static analysis to find paths that heavily access NVM and optimizes the search space to select the NVM-access-heavy-paths. Agamotto keeps track of memory regions using a state machine similar to PMTest [115] and checks for NVM programming bugs. *FlowNvm* does not check extra fence operations, however, it would be simple to extend *FlowNvm* to support this feature. The main benefit of Agamotto is the memory model, which has precise semantics for keeping track of program data state without executing the program. However, Agamotto still requires symbolic execution and bug detection can take minutes.

Overall methods that rely on runtime profiles and testing [136] suffer from a lack of full program coverage and miss some feasible program paths. Static analysis, (1) achieves coverage over all program paths, and (2) reports errors during compile time without executing the program. There are classic issues such as precision versus completeness as our approach suffers from a significant number of false positives. However, we should note that a conservative approximation can be helpful for the developer, even at the cost of reviewing the false positives as NVM program recovery code is infrequently executed.

5.7 Summary

The chapter presents *FlowNvm* tool, which is an LLVM extension for finding NVM program bugs using inter-procedural dataflow analysis. We identify three programming patterns (pair, durability, log) that can lead to incorrect recovery of the durable heap state after a restart and two anti-patterns (double flush, double log) that can result in degrading a program's runtime performance. We formalize the dataflow framework and explain the intuition behind the design decisions.

We show how dataflow analysis can be leveraged to identify the violation of the programming patterns and find instances of anti-patterns in the code. We show how *FlowNvm* uses inter-procedural dataflow analysis to cover all the paths in the program. We demonstrate that *FlowNvm* can find and report previously fixed bugs and a new bug. Tools such as *FlowNvm* help ensure the main reason for using NVM in practice, which is crash consistency with good performance.

6 Conclusion

In this thesis, we explored several software tools to support NVM programming. We describe the challenges for NVM programming (Chapter 1), primarily how volatile processor caches complicate NVM programming and require careful reasoning about object layout and cache-line evictions. We demonstrate that these challenges can lead to hard-to-find bugs, which require a crash-consistency mechanism to exploit NVM's durability. In implementing crash-consistency mechanisms, we explore the challenges posed by runtime and recovery overheads, volatile and durable memory overheads, ease of programming, and correctness of the system. We provide extensive background information on NVM programming including programming primitives, formal rules, machine configuration, crash-consistency mechanisms and checkpointing (Chapter 2).

Our three main contributions mitigate the performance and the recovery overheads and provide means to find NVM programming bugs. Firstly, with *InCLL* (Chapter 3), we focus on minimizing write-backs and fence instructions. Write-back and fence instructions on the critical path of a program can lead to runtime performance degradation due to inefficient cache use and blocking at fences. We partially solve this problem with an in-cache-line log.

InCLL places an entire data structure in NVM, breaks program execution into fine-grained intervals on the order of milliseconds, and checkpoints by evicting the cache hierarchy to NVM. *InCLL* uses an in-cache-line log, which resides in the same cache line as its data, as a roll-back cache to undo modifications within a failed epoch. We extend the in-cache-line log to persistent memory allocation and make Masstree completely durable. By using an in-cache-line log, *InCLL* avoids explicit write-back and fence instructions on Masstree's fast-path operations such as insertion, deletion, and update.

Secondly, with *CpNvm* (Chapter 4), we focus on reducing the cost of using Optane. While Optane is a fundamental improvement over traditional block-level addressable durable media (disk and SSD) due to its byte-addressability, lower latency, and higher bandwidth, *InCLL* does not perform well on it. We mitigate costs related to Optane by operating on a DRAM copy of a data structure. *CpNvm* propagates modifications with redo logging, which is later compressed to an NVM version of the data structure. We provide *CpNvm* as an easy-to-use API and show its practicality in Memcached as well. With *CpNvm*, we achieve low overheads and drastically improve performance, while minimizing program changes.

Finally, with *FlowNvm* (Chapter 5), we focus on quickly finding NVM programming bugs, which can lead to failures and data inconsistency after a restart or performance degradations during execution. We use inter-procedural dataflow analysis and develop a framework to find violations of common programming patterns and the existence of anti-patterns. We demonstrate that *FlowNvm* can find existing and new bugs in NVM programs.

6.1 Future Directions

There are interesting areas in our work that can be further explored. Both *CpNvm* and *InCLL* block while taking a checkpoint. The main reason for such an approach is that all the worker threads are operating on shared memory and it is necessary to stop to checkpoint at a globally consistent state of the data structure. A non-blocking approach could reduce the time spent blocked and further reduce overheads, avoid latency spikes and mitigate tail latencies.

Another important limitation is that *CpNvm* and *InCLL* are single-node systems. While a single-node system is useful for recoverable faults, when the node fails, it is not possible to recover the data structure. One interesting approach would be to explore fault tolerance using replication. There is a huge body of work on multi-machine replication. It would be possible to use an existing replication scheme to extend both *InCLL* and *CpNvm*. One possibility for *CpNvm* would be to replicate the *persistent image* to other machines. The background replay thread could use RDMA or state-machine replication to replay the redo log to NVM copies in other machines. The user of the system can select the number of replicas and the persistence guarantees. This approach would require keeping the redo log for an epoch until all the replicas are updated. Using replication would allow the program to survive single-node NVM crashes.

In addition, memory capacity of *CpNvm* in Mixed Mode is limited to roughly half of the capacity of NVM. We have not explored ways to extend the available capacity of DRAM and NVM together. Implementing a paging mechanism, either in the user-space or the kernel, might allow more efficient use of memory. However, implementing a paging mechanism is not simple. The *persistent logs* and *persistent image* contain the last committed checkpoint state. In other words, it is not possible to bring in a page from *persistent image* without first replaying the *persistent logs*. This would require keeping track of whether a page in the *persistent image* is waiting an update from the log. Otherwise, the page brought in from *persistent image* will be stale and lead to inconsistencies. One solution would be to always keep the page in memory and put it into NVM as a swap space. This is similar to what we do in Mixed Mode.

Finally, memory modeling in *FlowNvm* is not precise and it is the main bottleneck for the efficiency of our approach. We have not looked into methods such as symbolic execution and runtime profiling to derive a more precise memory region model. A pointer can point to multiple memory regions at a point in the program. An interesting challenge is how to use the information from these multiple memory regions for a program point. We should note that Agamotto's memory region modeling, which relies on symbolic execution and inferring NVM memory accesses, is a suitable approach.

6.2 Discussion

In this section, we briefly describe the key takeaways from our three major contributions.

6.2.1 Avoiding Write-Back and Fence Instructions

Using fine-grained checkpointing with an in-cache-line log allows a programmer to avoid explicit write-back and fence instructions on the critical path of a program. The main idea that enables this elimination is the intrinsic ordering of two different writes to the same cache line. This write ordering allows the implementation of a logging mechanism inside a cache line. However, for cases where the capacity of an in-cache-line log is not sufficient, we fall back to using a traditional external undo log to build a fully durable data structure.

Evicting the cache hierarchy to NVM periodically ensures the data structure in NVM is consistent at the beginning of the epoch. Our results show that for fast NVM devices (with DRAM characteristics), the approach is feasible with overheads less than 20% in the common case. However, with a real NVM device (Optane), the program operations on the data structure residing in NVM are very costly. The overheads can rise up to 64% for *InCLL-NVM*, which limits the feasibility of the approach.

6.2.2 Using Optane

Memory accesses to Optane are slow due to the device's high latency and low bandwidth. We use write-combining in DRAM and delay the propagation of writes to NVM. We keep the checkpointing design with intervals on the order of milliseconds. The program accesses its data in DRAM and only periodically propagate the modifications to NVM. This way, the program only pays for the persistence of modifications, but not its memory accesses.

We use existing well-known techniques such as shadow memory, bitmap, DRAM copy for checkpointing, redo logging and combine them in a novel way for NVM programming to provide an application-level checkpointing API that mitigates the cost of using Optane and provides better throughtput with low-overheads (< 15% for Masstree) compared to the fine-grained checkpointing approach for *InCLL*.

6.2.3 Bug Finding at Compile Time

The tools that rely on runtime profiling, testing, and symbolic execution are precise. However, these tools take more than minutes to execute and the work on testing can miss cold paths which are not executed. With *FlowNvm* we demonstrate that dataflow analysis is suitable for bug finding at compile time.

6.3 Conclusion

With the release of Intel Optane, we expect many changes in programming the storage tier. NVM's byte-addressable interface, high capacity, and performance approaching DRAM require new tools and new approaches to data structure design and implementation. Our goal in this thesis was to build software tools to support NVM programming and allow the programmers to have a wide array of choices for programming and checking their codebases. We built *InCLL* and *CpNvm* checkpointing strategies to implement low-overhead checkpointing mechanisms instead of transactions. Specifically, we built *InCLL* to minimize write-back and fence instructions and *CpNvm* to reduce the cost of using Optane. We built *FlowNvm* to capture NVM programming bugs at compile time instead of using costly testing and runtime strategies. We hope that our contributions allow both academia and industry to take full advantage of NVM.

Bibliography

[1] The Home of Checkpointing Packages, 2001. https://checkpointing.org.

[2] Memtier Benchmark, 2013. https://github.com/RedisLabs/memtier_benchmark.

[3] The libvmmalloc library, 2014. https://pmem.io/vmem/libvmmalloc.

[4] Deprecating the PCOMMIT Instruction, 2016. https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html.

[5] Intel® Optane™ Persistent Memory, 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[6] Persistent Memory Documentation, 2019. https://docs.pmem.io/persistent-memory/.

[7] RECIPE, 2019. https://github.com/utsaslab/RECIPE.

[8] The Challenge of Keeping Up with Data, 2019. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[9] Evaluation of In Cache Line Logging On Intel Optane Persistent Memory, 2020. https://pirl.nvsl.io/pirl2020-program/#paper-3001.

[10] mmap(2) — Linux manual page, 2020. https://www.man7.org/linux/man-pages/man2/mmap.2.html.

[11] Third Generation Intel® Xeon® Processor Scalable Family Technical Overview, 2020. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html.

[12] Dell poweredge r940 rack server : Servers: Dell usa, 2021. https://www.dell.com/en-us/work/shop/dell-poweredge-servers/poweredge-r940-rack-server.

[13] Intel® Optane™ Persistent Memory Products, 2021. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[14] H. Akinaga and H. Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010. https://ieeexplore.ieee.org/document/5607274.

[15] P. Alcorn. Intel Optane DIMM Pricing, 2019. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[16] R. Appuswamy, G. Graefe, R. Borovica-Gajic, and A. Ailamaki. The five-minute rule 30 years later and its impact on the storage hierarchy. *Commun. ACM*, 62(11):114–120, Oct. 2019. https://doi.org/10.1145/3318163.

[17] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, Jan. 2018. https://dl.acm.org/doi/abs/10.1145/3164135.3164147.

[18] H. Avni and T. Brown. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.*, 10(4):409–420, Nov. 2016. https://doi.org/10.14778/3025111.3025122.

[19] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2527792.2527799.

[20] L. A. Barroso, U. Holzle, P. Ranganathan, and M. Martonosi. *The Datacenter As a Computer: Designing Warehouse-Scale Machines*. 3rd edition, 2018. https://www.morganclaypool.com/doi/10.2200/S00874ED3V01Y201809CAC046.

[21] M. Beck, J. S. Plank, and G. Kingsley. *Compiler Assisted Checkpointing*. Citeseer, 1994.

[22] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009. http://pages.cs.wisc.edu/~johnbent/Pubs/bent_sc09.pdf.

[23] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 677–694, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2983990.2984019.

[24] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller. Twizzler: a data-centric OS for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 65–80. USENIX Association, July 2020. https://www.usenix.org/conference/atc20/presentation/bittman.

[25] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI ’08, page 68–78, New York, NY, USA, 2008. Association for Computing Machinery. https://doi.org/10.1145/1375581.1375591.

[26] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level checkpointing for shared memory programs. *ACM SIGPLAN Notices*, 39(11):235–247, 2004. https://dl.acm.org/doi/10.1145/1024393.1024421.

[27] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013. https://doi.org/10.1145/2408776.2408795.

[28] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, page 60–73, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3381898.3397212.

[29] G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, 1998. https://ieeexplore.ieee.org/document/737697.

[30] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’11, page 265–276, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1989323.1989352.

[31] D. Castro, P. Romano, and J. Barreto. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[32] 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*, OOPSLA ’14, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2660193.2660224.

[33] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proc. VLDB Endow.*, 8(5):497–508, jan 2015. http://dl.acm.org/citation.cfm?doid=2735479.2735483.

[34] S. Chen, P. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. *Cidr ’11*, (Section 6):21–31, 2011. http://www.cs.cmu.edu/{~}./chensm/papers/pcm-db-algo-cidr11.pdf.

[35] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. 8(7):786–797, Feb. 2015. https://doi.org/10.14778/2752939.2752947.

[36] Z. Chen, S. W. Son, W. Hendrix, A. Agrawal, W. Liao, and A. Choudhary. Numarck: Machine learning algorithm for resiliency and checkpointing. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 733–744, 2014. http://cucis.ece.northwestern.edu/publications/pdf/CSH14.pdf.

[37] 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. *SIGARCH Comput. Archit. News*, 39(1):105–118, Mar. 2011. https://doi.org/10.1145/1961295.1950380.

[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*, ASPLOS '19, page 441–454, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3297858.3304046.

[39] N. Cohen, D. T. Aksun, and J. R. Larus. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018. https://doi.org/10.1145/3276523.

[40] N. Cohen, M. Friedman, and J. R. Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017. https://doi.org/10.1145/3133891.

[41] P. Colp, C. Matthews, B. Aiello, and A. Warfield. Vm snapshots. *Xen Summit*, 2009. http://www-archive.xenproject.org/files/xensummit_oracle09/VMSnapshots.pdf.

[42] 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*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. https://doi.org/10.1145/1807128.1807152.

[43] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 271–282, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3210377.3210392.

[44] T. Coy, S. He, B. Ren, and X. Zhang. Compiler aided checkpointing using crash-consistent data structures in nvmm systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3392717.3392755.

[45] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, Boston, MA, July 2018. USENIX Association. https://www.usenix.org/conference/atc18/presentation/david.

[46] B. R. de Supinski, G. Bronevetsky, A. Moody, and K. Mohror. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Transactions on Parallel and Distributed Systems*, (01):1, may 5555. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6494566.

[47] M. F. L. De Volder, S. H. Tawfick, R. H. Baughman, and A. J. Hart. Carbon nanotubes: Present and future commercial applications. *Science*, 339(6119):535–539, 2013. https://science.sciencemag.org/content/339/6119/535.

[48] A. Demeri, W.-H. Kim, R. M. Krishnan, J. Kim, M. Ismail, and C. Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st International Middleware Conference*, pages 207–220, 2020. https://dl.acm.org/doi/10.1145/3423211.3425671.

[49] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2463676.2463710.

[50] W. R. Dieter and J. E. Lumpp. User-level checkpointing for linuxthreads programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 81–92, USA, 2001. USENIX Association. https://www.usenix.org/legacy/event/usenix01/freenix01/full_papers/dieter/dieter_html/paper.html.

[51] M. Dong and H. Chen. Soft updates made simple and fast on non-volatile memory. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 719–731, 2017. https://www.usenix.org/system/files/conference/atc17/atc17-dong.pdf.

[52] Z. Duan, H. Liu, X. Liao, and H. Jin. Hme: A lightweight emulator for hybrid memory. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1375–1380, 2018. https://doi.org/10.23919/DATE.2018.8342227.

[53] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2592798.2592814.

[54] Echo. snalli/echo at bc9f30fcdd, 2017. https://github.com/snalli/echo/tree/bc9f30fcdd.

[55] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013. https://link.springer.com/article/10.1007%2Fs11227-013-0884-0.

[56] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. M. Hazelwood, A. Cidon, and S. Katti. Bandana: Using non-volatile memory for storing deep learning models. *CoRR*, abs/1811.05922, 2018. http://arxiv.org/abs/1811.05922.

[57] R. Elkhouly, M. Alshboul, A. Hayashi, Y. Solihin, and K. Kimura. Compiler-support for critical data persistence in nvm. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–25, 2019. https://dl.acm.org/doi/abs/10.1145/3371236.

[58] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626. IEEE, 2012. https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/paper7.pdf.

[59] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. *SIGPLAN Not.*, 24(1):112–123, Nov. 1988. https://doi.org/10.1145/69215.69226.

[60] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. Phoenix: Memory speed hpc i/o with nvm. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 121–131. IEEE, 2016. https://ieeexplore.ieee.org/document/7839676.

[61] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, (124), 2004. https://www.linuxjournal.com/article/7451.

[62] M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 377–392, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3385412.3386031.

[63] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, C. Min, and D. Lee. Witcher: Detecting crash consistency bugs in non-volatile memory programs. *arXiv preprint arXiv:2012.06086*, 2020. https://arxiv.org/abs/2012.06086.

[64] S. Gao, B. He, and J. Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 263–272, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2751205.2751212.

[65] K. Genç, M. D. Bond, and G. H. Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 59–74, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3385412.3385991.

[66] Z. Ghodsi, S. Garg, and R. Karri. Optimal checkpointing for secure intermittently-powered iot devices. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 376–383. IEEE, 2017. https://arxiv.org/abs/1711.01454.

[67] E. Giles, K. Doshi, and P. Varman. Continuous checkpointing of htm transactions in nvm. *SIGPLAN Not.*, 52(9):70–81, June 2017. https://doi.org/10.1145/3156685.3092270.

[68] E. R. Giles, K. Doshi, and P. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015. https://ieeexplore.ieee.org/document/7208276.

[69] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(10):1304–1318, Apr. 2020. https://doi.org/10.14778/3389133.3389145.

[70] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 9–9. IEEE, 2005. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.448.3079&rep=rep1&type=pdf.

[71] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener. Fast database restarts at facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 541–549, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2588555.2595642.

[72] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 913–928, Renton, WA, July 2019. USENIX Association. https://dl.acm.org/doi/10.5555/3358807.3358885.

[73] S. Gupta, A. Daglis, and B. Falsafi. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 466–478, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3352460.3358321.

[74] D. Hakkarinen and Z. Chen. Multilevel diskless checkpointing. *IEEE Transactions on Computers*, 62(4):772–783, 2012. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6127862.

[75] S. Haria, M. D. Hill, and M. M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378472.

[76] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory: 2nd Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010. https://doi.org/10.2200/S00070ED1V01Y200611CAC002.

[77] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *Proceedings of the 2020 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '20, pages 877–892, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3318464.3389716.

[78] 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*, EuroSys '17, page 468–482, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3064176.3064204.

[79] Q. Hu, J. Ren, A. Badam, and T. Moscibroda. Log-structured non-volatile main memory. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX, July 2017. https://www.microsoft.com/en-us/research/publication/log-structured-non-volatile-main-memory/.

[80] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, Dec. 2014. https://doi.org/10.14778/2735496.2735502.

[81] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, July 2018. USENIX Association. https://www.usenix.org/conference/atc18/presentation/huang.

[82] Inspector. Intel® Inspector | Intel® Software, 2019. https://software.intel.com/en-us/inspector.

[83] Intel. PMFS-new/journal.c at 2c62f0a20f98afe128e59d5e7f0aff40489b27f7 · snalli/PMFS-new, 2016. https://github.com/snalli/PMFS-new/blob/2c62f0a20f98afe128e59d5e7f0aff40489b27f7/journal.c.

[84] Intel. snalli/nvml at 54871035ad2a8a59f26dd00f02287aecf19d5b01, 2017. https://github.com/snalli/nvml/tree/54871035ad.

[85] Intel. examples: btree: remove not needed snapshot · pmem/pmdk@b923240, 2018. https://github.com/pmem/pmdk/commit/b9232407a794040102e769ed98b967d797c173fd.

[86] Intel. examples: btree: snapshot node before modifying it · pmem/pmdk@25f5e4f, 2018. https://github.com/pmem/pmdk/commit/25f5e4f676e3d9cd7a4c9dc7aa8f2f36e83ff6c2.

[87] Intel. Remove duplicate flush buffer · snalli/PMFS-new@ded1b07, 2018. https://github.com/snalli/PMFS-new/commit/ded1b075eb911c469233433d83cb678ee800367c.

[88] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[89] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 427–442, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2872362.2872410.

[90] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In C. Gavoille and D. Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007%2F978-3-662-53426-7_23.

[91] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019. https://arxiv.org/abs/1903.05714.

[92] W. M. Jones, J. T. Daly, and N. DeBardeleben. Application monitoring and checkpointing in hpc: looking towards exascale systems. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 262–267, 2012. https://dl.acm.org/doi/10.1145/2184512.2184574.

[93] 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)*, pages 361–372, 2017. https://ieeexplore.ieee.org/document/7920839.

[94] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3341301.3359631.

[95] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using nvm as virtual memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 29–40, 2013. https://ieeexplore.ieee.org/document/6569798.

[96] P. Karhula, J. Janak, and H. Schulzrinne. Checkpointing and migration of iot edge functions. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '19, page 60–65, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3301418.3313947.

[97] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger. Viyojit: Decoupling battery and dram capacities for battery-backed dram. *SIGARCH Comput. Archit. News*, 45(2):613–626, June 2017. https://doi.org/10.1145/3140659.3080236.

[98] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. *SIGARCH Comput. Archit. News*, 44(2):399–411, Mar. 2016. https://doi.org/10.1145/2980024.2872381.

[99] I. Koren and C. Krishna. *Fault-Tolerant Systems*. Elsevier Science, 2020. https://books.google.ch/books?id=YrnjDwAAQBAJ.

[100] J. B. Kwon. On bypassing page cache for block devices on storage class memory. In J. J. J. H. Park, S.-C. Chen, and K.-K. Raymond Choo, editors, *Advanced Multimedia and Ubiquitous Engineering*, pages 361–366, Singapore, 2017. Springer Singapore. https://link.springer.com/chapter/10.1007/978-981-10-5041-1_59.

[101] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3132747.3132770.

[102] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013. http://sites.computer.org/debull/A13june/TimesTen1.pdf.

[103] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019. https://doi.org/10.1145/3335772.3335936.

[104] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz.

[105] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009. https://doi.org/10.1145/1555815.1555758.

[106] C. Lee, W. Shin, D. J. Kim, Y. Yu, S. Kim, T. Ko, D. Seo, J. Park, K. Lee, S. Choi, N. Kim, V. G, A. George, V. V, D. Lee, K. Choi, C. Song, D. Kim, I. Choi, I. Jung, Y. H. Song, and J. Han. Nvdimm-c: A byte-addressable non-volatile memory module for compatibility with standard ddr memory interfaces. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 502–514, 2020. https://ieeexplore.ieee.org/document/9065586.

[107] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, Feb. 2017. USENIX Association. https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon.

[108] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3341301.3359635.

[109] R. Levick. We need a safer systems programming language, 2019. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/.

[110] A. Limited. ArmR Architecture Reference Manual Armv8, for Armv8-A architecture profile, 2019. https://developer.arm.com/docs/ddi0487/latest/armarchitecture-reference-manual-armv8-for-armv8-aarchitecture-profile.

[111] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1997. https://minds.wisconsin.edu/handle/1793/60116.

[112] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *SIGARCH Comput. Archit. News*, 45(1):329–343, Apr. 2017. https://doi.org/10.1145/3093337.3037714.

[113] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018. https://doi.org/10.1145/2872362.2872410.

[114] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378452.

[115] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 411–425, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3297858.3304015.

[116] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, Jan. 2015. https://doi.org/10.1145/2644805.

[117] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2168836.2168855.

[118] 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. https://arxiv.org/abs/1804.00701.

[119] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association. https://www.usenix.org/conference/hotstorage17/program/presentation/marathe.

[120] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska. Unexpected performance of intel® optane™ dc persistent memory. *IEEE Computer Architecture Letters*, 19(1):55–58, 2020. https://ieeexplore.ieee.org/document/9072482.

[121] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 499–512, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3064176.3064215.

[122] A. Memaripour, J. Izraelevitz, and S. Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 789–806, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378456.

[123] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016. https://ieeexplore.ieee.org/document/7120149.

[124] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992. https://doi.org/10.1145/128765.128770.

[125] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010. https://www.cct.lsu.edu/~korobkin/tmp/SC10/papers/pdfs/pap236s4.pdf.

[126] 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*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3037697.3037730.

[127] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 31–44, USA, 2019. USENIX Association. https://dl.acm.org/doi/10.5555/3323298.3323302.

[128] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott. Dalí: A Periodically Persistent Hash Map. In A. W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://drops.dagstuhl.de/opus/volltexte/2017/8014.

[129] I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, Nov. 2020. https://www.usenix.org/conference/osdi20/presentation/neal.

[130] N. Neves and W. K. Fuchs. Coordinated checkpointing without direct coordination. In *Proceedings. IEEE International Computer Performance and Dependability Symposium. IPDS'98 (Cat. No.98TB100248)*, pages 23–31, 1998. https://www.di.fc.ul.pt/~nuno/PAPERS/IPDS98.pdf.

[131] X. Ni, T. Islam, K. Mohror, A. Moody, and L. V. Kale. Lossy compression for checkpointing: Fallible or feasible? In *Proceedings of the International Conference For High Performance Computing, Networking, Storage and Analysis (SC)*, 2014. http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech_poster/poster_files/post271s2-file3.pdf.

[132] I. Oukid, D. Booss, A. Lespinasse, and W. Lehner. On testing persistent-memory-based software. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2933349.2933354.

[133] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017. https://dl.acm.org/doi/abs/10.14778/3137628.3137629.

[134] 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*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/2882903.2915251.

[135] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European*

Conference on Computer Systems, EuroSys '13, page 225–238, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2465351.2465374.

[136] R. Patton. *Software testing*. Sams Pub, 2006. https://www.amazon.com/Software-Testing-Ron-Patton/dp/0672327988.

[137] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 265–276. IEEE Press, 2014. https://ieeexplore.ieee.org/document/6853222.

[138] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, page 18, USA, 1995. USENIX Association. https://dl.acm.org/doi/book/10.5555/898770.

[139] J. S. Plank, Kai Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=730527.

[140] J. S. Plank, J. Xu, and R. H. Netzer. Compressed differences: An algorithm for fast incremental checkpointing, 1995. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.1812.

[141] PMDK. pmem.io: PMDK, 2019. https://pmem.io/pmdk/.

[142] Pmemcheck. Discover Persistent Memory Programming Errors with Pmemcheck | Intel® Software, 2018. https://software.intel.com/en-us/articles/discover-persistent-memory-programming-errors-with-pmemcheck.

[143] PMFS. snalli/PMFS-new at 94323d525a10980fc6b9371234c26195acf0fc9f. https://github.com/snalli/PMFS-new/tree/94323d525a10980fc6b9371234c26195acf0fc9f.

[144] Pmreoder. pmem.io: PMDK man page. https://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html.

[145] D. Pritchett. Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue*, 6(3):48–55, May 2008. https://doi.org/10.1145/1394127.1394128.

[146] A. Raad and V. Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018. https://doi.org/10.1145/3276507.

[147] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019. https://doi.org/10.1145/3371079.

[148] 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.*, 3(OOPSLA), Oct. 2019. https://doi.org/10.1145/3360561.

[149] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, 2019. https://doi.org/10.1109/DSN.2019.00028.

[150] 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*, MICRO-48, page 672–685, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2830772.2830802.

[151] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery. https://doi.org/10.1145/199448.199462.

[152] G. V. Research. Non-volatile Dual In-line Memory Module Market Size, Share, Trends Analysis Report By Product (NVDIMM-N, NVDIMM-F, NVDIMM-P), By Capacity, By End Use, By Region, And Segment Forecasts, 2020 - 2027, 2020. https://www.grandviewresearch.com/industry-analysis/non-volatile-dual-in-line-memory-module-market.

[153] E. Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002. https://core.ac.uk/display/20753633.

[154] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello. Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013. https://dl.acm.org/doi/10.1145/2503210.2503271.

[155] A. Rudoff. Persistent memory programming. 2017. https://www.usenix.org/system/files/login/articles/login{_}summer17{_}07{_}rudoff.pdf.

[156] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *ICDE*, pages 452–462, 1989. https://cs.uwaterloo.ca/~kmsalem/pubs/saga87.pdf.

[157] S. Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers.* Springer Nature, 2020. https://www.amazon.com/Programming-Persistent-Memory-Comprehensive-Developers/dp/1484249313.

[158] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for nvram. In R. Bordawekar, T. Lahiri, B. Gedik, and C. A. Lang, editors, *VLDB*, pages 61–72, 2015. http://dblp.uni-trier.de/db/conf/vldb/adms2015.html#SchwalbBFDP15.

[159] S. M. Shahri, S. A. V. Ghahani, and A. Kolli. (almost) fence-less persist ordering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 539–554. IEEE, 2020. https://www.microarch.org/micro53/papers/738300a539.pdf.

[160] 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*, PLDI 2019, page 316–332, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3314221.3314608.

[161] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., USA, 5 edition, 2005. https://dl.acm.org/doi/book/10.5555/993519.

[162] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61. IEEE, feb 2011. http://ieeexplore.ieee.org/document/5749716/.

[163] snalli. Merge pull request for Echo-KV, 2019. https://github.com/snalli/echo/commit/ded09b5aaed63655cdcac60baa75de3663ed4556.

[164] SNIA. NVM Programming Model, 2017. https://www.snia.org/tech_activities/standards/curr_standards/npm.

[165] J. Speer and M. Kirchberg. C-aries: A multi-threaded version of the aries recovery algorithm. In *International Conference on Database and Expert Systems Applications*, pages 319–328. Springer, 2007. https://link.springer.com/chapter/10.1007%2F978-3-540-74469-6_32.

[166] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery. https://doi.org/10.1145/237721.237727.

[167] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 109–120, 2011. https://dl.acm.org/doi/10.1145/1993744.1993757.

[168] M. H. Sun and D. M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Georgia Institute of Technology, 2010. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.182.3520.

[169] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2517349.2522713.

[170] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7, 2019.

[171] A. V. Vathsala and H. Mohanty. A survey on checkpointing web services. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, PESOS 2014, page 11–17, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2593793.2593795.

[172] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida. Speculative memory checkpointing. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 197–209, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2814576.2814802.

[173] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 37–49, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2814576.2814806.

[174] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2592798.2592810.

[175] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 46(3):91–104, Mar. 2011. https://doi.org/10.1145/1961296.1950379.

[176] J. P. Walters and V. Chaudhary. Application-level checkpointing techniques for parallel programs. In S. K. Madria, K. T. Claypool, R. Kannan, P. Uppuluri, and M. M. Gore, editors, *Distributed Computing and Internet Technology*, pages 221–234, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007%2F11951957_21.

[177] H. Wan, Y. Lu, Y. Xu, and J. Shu. Empirical study of redo and undo logging in persistent memory. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2016. https://ieeexplore.ieee.org/abstract/document/7547178.

[178] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 524–533. IEEE, 2010. https://ieeexplore.ieee.org/document/5695644.

[179] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar. Vm-μcheckpoint: Design, modeling, and assessment of lightweight in-memory vm checkpointing. *IEEE Transactions on Dependable and Secure Computing*, 12(2):243–255, 2014. https://ieeexplore.ieee.org/document/6824750.

[180] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3295500.3356159.

[181] H. Wen, W. Cai, M. Du, L. Jenkins, B. Valpey, and M. L. Scott. Montage: A general system for buffered durably linearizable data structures, 2020. https://arxiv.org/pdf/2009.13701.pdf.

[182] S. Wu, F. Zhou, X. Gao, H. Jin, and J. Ren. Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications. 15(4), Jan. 2019. https://doi.org/10.1145/3291057.

[183] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 1–7, 2016. http://ranger.uta.edu/~sjiang/pubs/papers/wu16-nvmcached.pdf.

[184] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3399666.3399898.

[185] Z. Wu, K. Lu, A. Nisbet, W. Zhang, and M. Luján. Pmthreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 623–637, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3385412.3386000.

[186] 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*, USENIX ATC '17, pages 349–362, USA, 2017. USENIX Association. https://dl.acm.org/doi/10.5555/3154690.3154724.

[187] J. Xu, J. Kim, A. Memaripour, and S. Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–439, 2019. https://dl.acm.org/doi/10.1145/3297858.3304077.

[188] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 169–182, 2020. https://www.usenix.org/system/files/fast20-yang.pdf.

[189] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, Feb. 2015. USENIX Association. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang.

[190] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai. Algorithm-directed crash consistence in non-volatile memory for hpc. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 475–486. IEEE, 2017. http://www.pasalabs.org/papers/2017/cluster17_algorithm_directed_memory.pdf.

[191] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite. Optimizing {VM} checkpointing for restore performance in vmware esxi. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 1–12, 2013. https://homes.cs.washington.edu/~iyzhang/papers/vmrestore-atc13.pdf.

[192] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 14–23, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1555754.1555759.

[193] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019. https://doi.org/10.1145/3360554.

David Teksen Aksun

https://github.com/daksunt | https://www.linkedin.com/in/david-teksen-aksun/
davidteksenaksun@gmail.com

EDUCATION

PH.D., COMPUTER SCIENCE
EPFL
2015-2021 | Lausanne, Switzerland
Supervisor: Prof. James Larus
Thesis: Software Support for Non-Volatile Memory (NVM) Programming

BS, COMPUTER ENGINEERING
ISTANBUL TECHNICAL UNIVERSITY
2011-2015 | Istanbul, Turkey
Thesis: Active Learning for Person Identification
Summa cum laude 4.00/4.00 Video

HIGH SCHOOL
AMERICAN ROBERT COLLEGE
2011 | Istanbul, Turkey

SKILLS

PROGRAMMING
Proficient: C/C++ (LLVM, DynamoRIO), Java, Python (Matplotlib, Jupyter notebook, Numpy, Pandas, Tensorflow, PyTorch)
Exposure: SQL, Scala, Javascript, HTML, CSS, Cython, Shell scripting

TOOLS
Database: MySQL, Oracle, MongoDB, Redis, Memcached
ML: Neural Networks, SVM, etc.
Tools: Vtune, Perf

AWARDS

- HiPEAC 2019 Paper Award for publication (2019)
- Doctoral degree fellowship by IC EPFL (2015)
- Ranked 1st in Istanbul Technical University 4.00/4.00 GPA (2015)
- Best thesis award in Bachelor's degree (2015)

EXTRA-CURRICULAR

- Member of Next Gen Leader Club (2013 - present)
- Voluntary work in elderly people home (2010 - 2011)

EXPERIENCE

EPFL | PH.D. STUDENT IN COMPUTER SYSTEMS IN VLSC
September 2015 – July 2021 | Lausanne, Switzerland

- My main research focuses on building software tools for programming **non-volatile memory**. My research interests are the intersection of databases, operating systems, parallel programming and program analysis.
- Designed and built checkpointing tools for NVM with less than 15% overhead for Masstree and 6% overhead for Memcached on real Optane device for write-intensive workloads.
- Designed and built **data flow analysis tool** to finds bugs in real NVM programs at compile time.
- Worked on the implementation and evaluation of **durable data structure design** for Masstree for NVDIMM-N technology with overhead less than 15.5%. [ASPLOS, 2019]
- Worked on object-oriented recovery issues such as object reconstruction and pointer-position fixing after restart for C/C++ programs for NVM. [OOPSLA, 2018]
- Researched into finding bugs in C/Java programs using deep neural networks
- Implemented a simplified DynamoRIO version of Daikon to research bug finding using invariant analysis and machine learning

MICROSOFT RESEARCH | RESEARCH INTERN IN RISE GROUP
Summer 2017 | Redmond, WA

- Worked in a team to research into the time spent in recurrent-neural network training and inference on CPUs using a novel concurrency technique previously used to optimize dynamic programming algorithms.

AGITO | WEB DEVELOPER
August - September 2013 | Istanbul, Turkey

- Worked in a team to develop website pages for an (undisclosed) insurance company using Java Web technologies such as JSF 2 and PrimeFaces. Used PL/SQL for the Oracle database

BOSPHORUS UNIVERSITY | LAB INTERN
June - August 2013 | Istanbul, Turkey

- Worked on designing operational amplifiers

DATASERV | SECURITY ANALYST
January - February 2013 | Istanbul, Turkey

- Configured NAC devices for small-size companies

TEACHING (DOCTORAL ASSISTANT)

- Information, Computation, Communication at EPFL CS-119(g, d), Autumn 2019, 2020
- Global Issues: Communication B at EPFL HUM-122(b), Spring 2017, 2019, 2020
- Programming II at EPFL CS-112(g), Spring 2018
- Software Engineering at EPFL CS-305, Autumn 2018
- Applied data analysis at EPFL CS-401, Autumn 2017
- Programming I at EPFL CS-111(g), Autumn 2016