

Snapshotting Mechanisms for Persistent Memory-Mapped Files

by

Mohammad Moridi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

© Mohammad Moridi 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this research, we explore ways to enhance the reliability of persistent memory systems. Using Montage (ICPP’21) as our reference model, we identify areas of potential improvement, especially concerning the risk of data loss in certain failure scenarios. Our investigation led us to focus on the concept of snapshotting and its role in system resilience. We delve into various consistency models for snapshotting mechanisms and introduce a new definition for snapshotting consistency known as Buffered-Durable Consistency. Montage offers impressive resilience against system-wide crash failures. However, we perceive opportunities for further fortification, specifically when persistent memory failures occur, which can lead to substantial data loss. Addressing this challenge, we propose two new snapshotting mechanisms - stop-the-world and online snapshotting - for memory-mapped files. These mechanisms selectively replicate only those data portions that have been modified since the last snapshot, thus significantly reducing the volume of data copying during snapshot operations. In order to ensure data consistency and optimize snapshotting, we introduce modifications to both Montage and its allocator. This includes a parallel chunk-copying strategy for the stop-the-world snapshotting implementation. Additionally, our online snapshotting mechanism allows updates to chunks not being replicated by the snapshotter, which increases system responsiveness. We have also developed an algorithm that, when snapshotting is not in progress, disables the reader locks, reducing the overhead associated with such locks and further enhancing performance. To demonstrate the effectiveness of our approach, we present an experimental analysis showing throughput and latency for various scenarios. Overall, our work not only heightens the fault tolerance capabilities of Montage, but also offers critical insights and potential directions for future research and optimization in the field of persistent memory, especially for memory-mapped file replication.

Acknowledgements

My deepest gratitude to Professor Wojciech Golab, whose guidance in topic selection, proof establishment, and experimental execution was indispensable to this work.

Dedication

Heartfelt thanks to my parents, Abbas Moridi and Azardokht Alimohammadi, and to my sisters, Maryam and Mona. Your constant encouragement and unwavering faith have been invaluable in my journey. Thank you!

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	x
1 Introduction	1
1.1 Persistent Memory	1
1.2 Persistent Memory Programming Model	2
1.2.1 Memory-Mapped Files	2
1.3 Montage	3
1.4 Snapshotting	4
1.4.1 The Drawbacks of Forking	4
2 Background	7
2.1 Persistent Memory in Systems with Volatile Components	7
2.1.1 Address Space Layout Randomization (ASLR)	8
2.1.2 Persistent Pointers	8

2.2	Safety Properties	9
2.2.1	Model	10
2.2.2	Definitions	12
2.3	Montage	15
2.3.1	Payload	15
2.4	Ralloc	16
2.5	Recovery Mechanisms in Montage and Ralloc	18
2.5.1	Montage Recovery	18
2.5.2	Ralloc Recovery	18
2.5.3	Montage and Ralloc Recovery Integration	19
3	Consistency Models for Snapshots	20
3.1	Model	20
3.2	Existing Consistency Definitions	22
3.3	Buffered Durable Consistency	25
3.3.1	Example	26
3.3.2	Terminology and Notation	27
3.3.3	Consistency Definition	29
3.4	Consistency Model for Montage	30
3.4.1	Consistency Definition	31
4	Stop-the-World Snapshotting	33
4.1	Ralloc Regions	34
4.2	Modifications to Montage	36
4.3	Modifications to Ralloc	40
4.4	Correctness	41

5	Online Snapshotting	44
5.1	Synchronization	44
5.2	Maintaining Superblock Region Consistency	45
5.3	Snapshotting Descriptor and Metadata Regions	48
5.4	Correctness	50
5.5	Limitations	51
6	Enhancing Performance with Double-Checked Locking	52
6.1	Algorithm	52
6.2	Maintaining Consistency	56
6.2.1	Online Snapshotting	56
6.2.2	Stop-the-World Snapshotting	58
6.3	Correctness	59
6.3.1	Online Snapshotting	59
6.3.2	Stop-the-World Snapshotting	60
7	Experimental Evaluation	62
7.1	Finding the Optimal Chunk Size	63
7.2	Throughput Analysis	68
7.2.1	Responsiveness	70
7.3	Snapshotting Time	71
7.4	Impact of Write-Backs	73
7.5	Take-Aways	77
8	Related Work	78
8.1	Snapshotting	79
8.1.1	Consistency Models	79
8.1.2	Checkpointing	79
8.1.3	In-Memory Databases	80

9 Conclusion and Future Work	82
References	84
APPENDICES	92
A Alternative Implementation Techniques	93

List of Figures

2.1	A file on PMEM can be mapped to different addresses in different processes or during several program executions.	9
2.2	Ralloc regions	17
3.1	Process p performs a read/write operation in memory region r at the time t	22
3.2	This figure demonstrates four potential cuts among two causally dependent events, identifying one inconsistent cut (c_1) and three consistent cuts (c_0 , c_2 , and c_3).	23
3.3	Instantaneous snapshot	24
3.4	Consistent quasi-instantaneous snapshot	24
3.5	Inconsistent quasi-instantaneous snapshot	25
3.6	Snapshotting in a buffered-durable platform	26
3.7	Recovery in a buffered-durable platform	28
4.1	Chunks in the memory-mapped files	34
4.2	First two snapshots	35
4.3	A reader/writer lock ensures consistency during snapshotting by allowing application threads to modify memory-mapped files in shared mode while the snapshotter thread acquires the lock in exclusive mode.	36
5.1	Online snapshotting	47
6.1	DCL in Online Snapshotting: Potential inconsistencies during snapshotting are resolved by updating the altered part at the backup.	56

7.1	Data structure: Hash map (g0i50r50). Algorithm: Multi-threaded stop-the-world snapshotting without double-checked locking.	65
7.2	Data structure: Hash map (g90i5r5). Algorithm: Multi-threaded stop-the-world snapshotting without double-checked locking.	66
7.3	Data structure: Queue (enq50deq50). Algorithm: stop-the-world snapshotting without double-checked locking.	67
7.4	Average throughput for hash map	69
7.5	Average throughput for queue (enq50deq50)	70
7.6	Average number of conflicts of a hashmap during online snapshotting . . .	71
7.7	Average snapshotting time for hash map	72
7.8	Average snapshotting for queue (50enq50deq)	73
7.9	Impacts of cache-line write-backs on hashmaps	75
7.10	Impacts of cache-line write-backs on queue(enq50deq50)	76
A.1	Throughput of various implementations of a hash map (log-scale y-axis) . .	94
A.2	Average throughput of alternative implementations of a queue	96

Chapter 1

Introduction

Persistent memory has catalyzed a revolutionary shift in data storage and retrieval. This chapter discusses the role of memory-mapped files and the Montage [72] system in capitalizing on this technology. To tackle potential memory failures, the chapter proposes a snapshotting mechanism, enhancing fault tolerance and data recovery. It further clarifies why the Linux `fork()` system call is not suitable for snapshotting persistent memory, advocating for an explicit file-copying method instead.

1.1 Persistent Memory

Research on byte-addressable non-volatile memory (NVM) has witnessed significant advancements in recent years. Known as persistent memory (PMEM), this technology is emerging as a potential game changer, aiming to redefine our data storage and access paradigms. PMEM offers performance levels that are comparable to volatile DRAM [32], coupled with the advantage of retaining its content during transient failures like power outages. Challenging the traditional memory hierarchy of DRAM paired with secondary storage, PMEM can operate as an independent storage solution, eliminating the necessity for secondary storage in various scenarios.

However, this transformation is not without its challenges. Programming for persistent memory presents difficulties that necessitate novel solutions. Academic research has risen to this challenge, pioneering more effective programming methods that allow developers to leverage the efficiency advantages of persistent memory. The volatile nature of CPU caches poses a significant challenge as there is no assurance regarding the timing or order of data

being transferred from the caches to persistent memory. This uncertainty creates potential inconsistencies as a thread could potentially access a value that has not been durably written to persistent memory and might get lost if a system failure occurs. Persistent memory has changed the way programmers manage data. Consistency in data is crucial for modern storage systems. Yet, without the right system tools, it is challenging to keep track of and update data efficiently. To help with this, there are specialized processor instructions like cache line flush instructions and memory fences. These instructions make sure data is transferred back to persistent memory in a certain order. There are various strategies in place, from improved data structures to transactional memory, which ensure that data remains consistent even when multiple processes access it.

1.2 Persistent Memory Programming Model

The Storage Networking Industry Association (SNIA) introduced the SNIA NVM Programming Model to create a consistent approach to programming with persistent memory [64]. This model lets applications ask the operating system about available persistent memory and decide where data should be stored: in persistent memory or regular DRAM [24]. One key method facilitated by this model is the use of memory-mapped files. These files allow applications to interact with persistent memory directly, using familiar programming techniques, thus bypassing traditional storage pathways and gaining direct access to the speed and persistence of NVM [64].

1.2.1 Memory-Mapped Files

Memory-mapped files play a crucial role in the utilization of persistent memory in software applications. This technology permits the mapping of a file or a portion of a file stored on a file system into the address space of a process. In the context of persistent memory, this enables applications to directly access non-volatile, byte-addressable memory regions as if they were conventional memory, thereby capitalizing on the distinctive benefits of persistent memory technology. Typically, an application would initiate this procedure by opening a file on persistent memory. Creating a memory mapping for the file using a system call such as `mmap()` (in Unix-like systems) is the next step. This operation returns a memory pointer that the application can use to directly read from and write to the persistent memory region that has been mapped [41]. Upon successful mapping, the application can perform read and write operations to the persistent memory using standard load and store instructions, as if it were interacting with regular volatile memory. This approach bypasses

the OS’s traditional I/O operations, such as `msync`, which is designed to ensure that all write operations complete in accordance with synchronized I/O data integrity [42]. This results in faster data access. Crucially, data written to this persistent memory remains durable (if it is flushed to NVM), ensuring that it is retained even after the application stops or the system restarts.

It is essential to mention the significance of cache coherence in multiprocessor systems that utilize volatile caches. When data is written by a processor, cache line flushing ensures the update is visible to all processors, preserving system-wide memory consistency. Memory-mapped files offer many advantages for working with persistent memory, but they also come with challenges. Consistency is vital, especially when multiple threads are in play or if the system crashes unexpectedly. Steps like cache line flushing and using memory barriers may be needed to ensure data is saved promptly and correctly. Cache line flushing ensures data from the CPU’s cache is written to the memory, and memory barriers control the order of memory operations to prevent unwanted instruction reordering by the CPU.

1.3 Montage

Montage [72] is a specialized system for managing persistent “*payloads*” within concurrent data structures. In this context, a *payload* is a unit that captures the key data and structural information of the data structure [72]. For a clear perspective, let us delve into the example of a queue. A queue inherently holds items in a distinct order. In Montage, the payload for each item in the queue would encapsulate not just the item’s data, but also its relative position within the queue [72]. This preservation ensures that, in the wake of unexpected disruptions, the queue can be seamlessly reconstructed with all its items in their correct sequence.

At its core, Montage operates by segmenting time into “*epochs*”, serving as reliable milestones in the order of operations [72]; all payloads are persisted at the end of each epoch. While Montage manages this alignment, synchronization between data structure operations remains the responsibility of the application. Optimized for C++, Montage provides seamless tools for payload creation, modification, and deletion, while also proficiently managing epoch transitions.

1.4 Snapshotting

Montage ensures system consistency following a system-wide crash failure by employing its recovery routine [72], which will be thoroughly discussed in Section 2.5. Yet, it does not address failures specific to persistent memory. It is noteworthy that snapshotting serves as a resilience strategy in numerous relational databases. Given that even reliable modules, such as Optane PMEM, are vulnerable to unanticipated external events — whether environmental (e.g., earthquakes, floods, fires) or human-induced — or can fail due to manufacturing defects, a dependable backup solution is indispensable. Hence, we advocate for the integration of **snapshotting into Montage**. Such an approach would duplicate the data structure’s state to an alternative persistent memory drive (PMEM) or a hard disk drive (HDD), bolstering fault tolerance and ensuring a safeguard against persistent memory failures.

1.4.1 The Drawbacks of Forking

In Linux, the `fork()` system call creates a new process by duplicating the current one. The new process, termed the child, is identical to the parent process with certain exceptions such as differing Process IDs. When considering memory-mapped files and their interactions with `fork()`, it is pivotal to differentiate between the two primary mapping modes: `MAP_PRIVATE` and `MAP_SHARED`.

`MAP_PRIVATE`

- When a file is memory-mapped with `MAP_PRIVATE`, the Copy-on-Write (CoW) mechanism is employed. In this scenario, the parent and child processes share the same memory pages immediately after the `fork()` call.
- If either the parent or child alters a page, a new version of that page is created in DRAM, separating it from the original page. These modifications are transient: they are not visible to other processes mapping the same file, nor are they written back to the underlying persistent memory-mapped file. Therefore, while the original content remains intact, these modifications are temporary, lasting only for the duration of the process’s lifespan.

MAP_SHARED

- When using `MAP_SHARED`, the memory-mapped file is shared among processes. Any changes to the mapped region are directly written back to the underlying file, ensuring data persistence and making them visible to other processes that map the same file.
- This behaviour means that after a `fork()`, modifications made by the child process are seen by the parent and vice versa because both processes interact with the same shared memory segment.

Given this explanation, the `fork()` system call introduces challenges for memory-mapped files.

1. **Lack of Isolated Snapshots:** `MAP_SHARED`, when used in conjunction with `fork()`, poses challenges for recovery or snapshot creation. Any change made by either the parent or child post-fork is reflected in the underlying file. This behaviour is at odds with the requirement for a distinct, stable point-in-time snapshot.
2. **Transient Modifications with `MAP_PRIVATE`:** Although `MAP_PRIVATE` protects the original file's content from changes thanks to the CoW mechanism, these changes are ephemeral. This is counter-intuitive when considering the primary advantage of persistent memory: data durability beyond process termination.
3. **Explicit Copying for Snapshots:** Creating a true and independent snapshot of a memory-mapped file necessitates an explicit copying procedure. This means generating a separate file and transferring the content from the memory-mapped region to this new file—a potentially resource-intensive operation in terms of I/O and time, especially for large files.

In summary, while the `fork()` system call offers many advantages, it does not directly provide a mechanism to create a durable, standalone snapshot of a memory-mapped file. For such requirements, systems should implement specialized strategies aligned with snapshotting needs. On the other hand, our solutions replicate memory-mapped files that Montage's allocator uses for memory management. In case of failure, the backup files can replace the primary files if a consistent copy is available. This approach improves the dependability and resilience of Montage without requiring significant changes to the existing implementation.

In the following chapter, Chapter 2, we will explain key concepts that are fundamental to understanding the breadth of our research. Chapter 3 explores existing consistency models for snapshots and introduces a new definition most appropriate for Montage. Subsequently, we propose two snapshotting algorithms, detailed in Chapters 4 and 5, and an algorithm to improve the performance of the explained mechanisms in Chapter 6. In Chapter 7, we conduct experiments to assess the efficacy and performance of these algorithms. A comprehensive review of related research and findings is presented in Chapter 8. Finally, Chapter 9 encapsulates the conclusions drawn from our work and outlines potential avenues for future research.

Chapter 2

Background

In this chapter, we will delve into various safety properties that are pivotal in the context of persistent memory management. We will explore key concepts such as Strict Linearizability, Persistent Atomicity, Recoverable Linearizability, and Durable Linearizability. We will then introduce and elaborate on Buffered Durable Linearizability, a safety property that assumes full-system crashes and ensures the system state remains consistent post-crash. We will also explore its practical implementation through the Montage system, a buffered-durably linearizable platform designed to manage persistent data structures effectively. In the later sections, the chapter will touch upon Ralloc, a persistent memory allocator that works in conjunction with Montage, and the recovery strategies adopted by both systems to restore consistent states following crashes.

2.1 Persistent Memory in Systems with Volatile Components

- **Memory Architecture:**
 - *Persistent Memory*: Directly attached to the memory bus, byte-addressable, retains data even after system-wide crash failures.
 - *Volatile Components*: CPU caches and registers lose data after power outages.
- **Cache Line Write-back Mechanisms:**

- *Implicit Write-Back*: Automated by the processor’s cache controller, triggered by cache policies or a need for fresh data.
- *Explicit Write-Back*: The programmer or specific software components issue cache line write-backs using designated processor instructions. This mechanism offers fine-grained control over cache behaviour.

- **Intel Persistence Instructions:**

- *Common Purpose*: Both CLWB and CLFLUSH are instructions used to execute explicit write-backs to memory.
- *Distinct Behaviours*:
 - * CLWB: The cache line is written back, and whether it retains its presence in the cache depends on the generation of the Intel processor.
 - * CLFLUSH: In addition to writing back, it invalidates the cache line. Consequently, future accesses require the line to be reloaded from memory, which can be slower than CLWB.

- **Ordering:**

- SFENCE: Guarantees that all store instructions before it are globally apparent, ensuring proper data persistence order.

2.1.1 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a security technique used by modern operating systems. Its primary function is to randomize the memory addresses where different sections of a process’s address space are loaded. This approach mitigates attacks like buffer overflow, where attackers take advantage of predictable memory layouts to control or corrupt an application’s behaviour. Consequently, due to ASLR, the exact address at which any segment is loaded into a process’s address space can vary between different executions. With respect to memory-mapped files on persistent memory using this technique, the files on persistent memory will be memory-mapped to different addresses for each process, resulting in a different base address for each mapping (as shown in Figure 2.1).

2.1.2 Persistent Pointers

Consistent addressing across restarts is often desired for applications utilizing persistent memory. However, the variability introduced by ASLR complicates the use of persistent

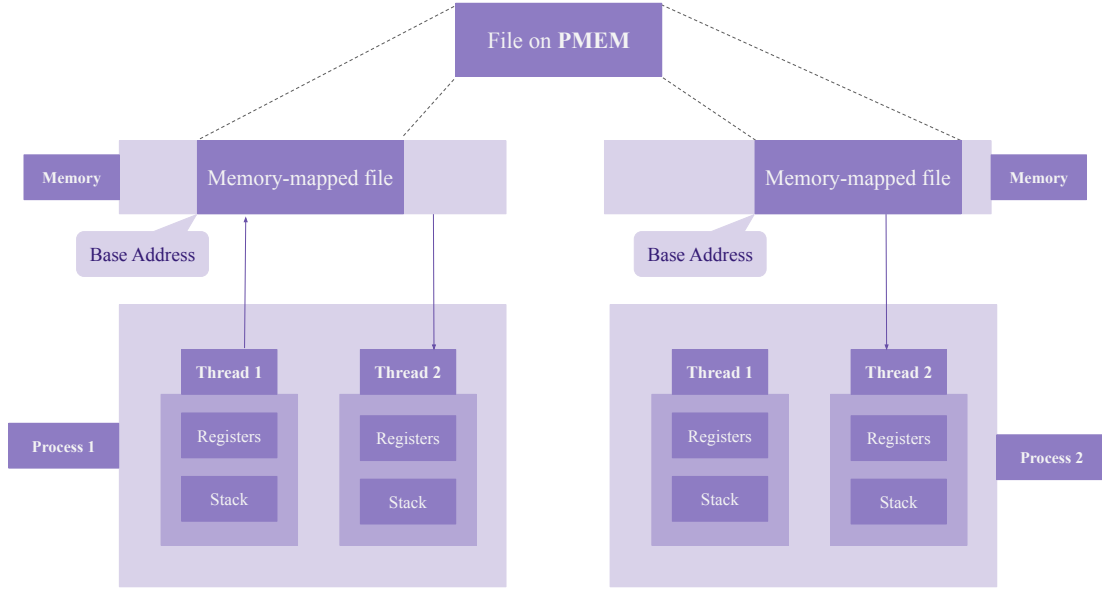


Figure 2.1: A file on PMEM can be mapped to different addresses in different processes or during several program executions.

pointers. Since a file might be loaded at a different virtual address during each memory mapping due to ASLR, persistent pointers cannot rely solely on absolute addresses. To address this challenge, a common solution is to use relative offsets. Rather than storing an absolute memory address, persistent pointers capture the offset from the base address of the memory-mapped file. When the file is mapped again, and the base address might differ due to ASLR, the application can compute the correct memory address by adding the saved offset to the new base address of the mapped file.

2.2 Safety Properties

To fully comprehend the concept of *Buffered Durable Linearizability*, it is important to first understand some other foundational safety conditions related to persistent memory management first.

2.2.1 Model

This subsection elaborates on the concurrent system model, providing detailed terminologies and notations to facilitate comprehension. The model follows the framework established by [26] and [31]:

- *Asynchronous Processes/Threads (p_{id}):* The system is composed of asynchronous processes/threads. Each process/thread has a specific identifier and communicates with the others using memory access primitives.
- *System Crashes:* A crash event signifies an unexpected system failure. While persistent memory remains intact, data in CPU caches and registers are lost, assuming we are working with the 2nd generation of the Intel Xeon Scalable processor series.
- *Shared Objects:* Each shared object carries a unique global identifier and is associated with a specific data type that dictates its behaviour through a set of operations.
- *Invocation and Response Events:* These events delimit the lifecycle of an operation. The invocation event marks the initiation, and the response event signifies the conclusion of an operation on a shared object by a process.
- *Cache Line Write-Back:* Represents the action of writing back data from a cache to the persistent memory. This is pivotal for ensuring that changes cached are consistently reflected in the PMEM. For objects spanning multiple cache lines, each line must be individually written back.
- *Recovery Routine:* Following a system crash, this routine restores a crashed process's operations by harnessing the intact content in persistent memory. Additionally, there is a central recovery routine that recovers all processes.
- *Execution:* Captures the sequence of events during a system's run, each marked with a unique timestamp, embodying the entire behaviour, including operations and state transitions.
- *Histories (H):* Represented as H , a history models the system's execution as a finite sequence of invocation, response, and crash events, as outlined in [21, 26].
- *Projections:* These are segments derived from the history H , specifically associated with either a process $H|p$ or an object $H|O$.

- Considering our assumption of system-wide crash failures, it is implied that every process in the system experiences these crash events. As a result, both $H|p$ and $H|O$ projections retain these crash events.
- *Equivalence of Histories:* Two histories, H and H' , are considered equivalent if $H|p = H'|p$ for each process p .
- *Sequential Histories:* These are histories where operations are linearly ordered. Every operation completes before the next one starts, ensuring there are no overlapping or concurrent operations. Additionally, they exclude crash events.
- *Sequential Specification:* Each object type provides this specification, detailing the object's abstract state and describing the impacts of various operations on this state.
- Example: Sequential Specification of a FIFO Queue X
 - State: A list Q representing the queue's elements. Initially, Q is empty.
 - Operations:
 - * `enqueue(x)`:
 - **Precondition:** None.
 - **Effect:** Appends element x to the end of the list Q .
 - **Postcondition:** Element x is now the last element in the list Q .
 - **Return Value:** None.
 - * `dequeue()`:
 - **Precondition:** Q must not be empty.
 - **Effect:** Removes and returns the element from the front of the list Q .
 - **Postcondition:** The front element of Q has been removed. If the precondition was not met (i.e., Q was empty), a special error code indicating that an error occurred during the execution is returned.
 - **Return Value:** The front element of Q or an error if Q was empty.
 - Behaviour:
 - * If you `enqueue` an element x to an empty queue, Q will contain just x .
 - * If you `enqueue` elements in the sequence x, y, z , the queue will have them in the order x, y, z .
 - * A `dequeue` operation will remove and return x first, then y on a subsequent `dequeue`, and finally z .

- *Legal Sequential Histories*: A history earns the “legal” tag if it abides by the rules set by the sequential specification of every participating object type.

2.2.2 Definitions

Definition 2.2.1 (Well-Formedness [30]). A history H is said to be well-formed if and only if $H|_p$ is sequential for every process p .

Definition 2.2.2 (Happens-Before in History H ($\prec_H$) [30]). In any well-formed history H containing events E_1 and E_2 , we say that E_1 happens before E_2 in H (denoted $E_1 \prec_H E_2$) if E_1 comes before E_2 in the sequential order of events in H (i.e., E_1 occurs earlier in the timeline of H than E_2) and any of the following holds:

1. E_1 is a crash,
2. E_2 is a crash,
3. E_1 is a response and E_2 is an invocation,
4. there exists an event $\hat{E}$ such that $E_1 \prec_H \hat{E}$ and $\hat{E} \prec_H E_2$.

Linearizability [26] is a fundamental concept in concurrent computation, representing a strict correctness criterion for concurrent object operations. It is a safety property that ensures the orderly completion of operations, thereby preventing unanticipated or unforeseen outcomes. As one of the most well-known single-object consistency models, linearizability ensures that every operation appears to occur instantaneously and atomically in a sequence that corresponds to their real-time sequence.

Definition 2.2.3 (Linearizability[26]). A well-formed history H is linearizable if:

- H does not contain any crash events.
- There exists an extended history H' such that H' is derived from H either by appending responses to its pending invocations or by removing them. Not all operations in H might have matching invocation-response pairs, but those in H' do.
- H' is equivalent to some legal sequential history S , meaning for every process p , $H'|_p = S|_p$.

- If event e_1 happens before event e_2 in H (denoted $e_1 <_H e_2$), and both events are in S , then e_1 must appear before e_2 in S (i.e., $e_1 <_S e_2$).

Strict linearizability [1] is a refined version of the linearizability model in concurrent computing that allows operations to terminate their execution prematurely while imposing restrictions on their effects [3]. In the context of a crash-recovery model, linearizability is intensified to ensure that operations either take full effect before a crash or do not influence the system at all, a property referred to as strict-linearizability [1]. Consequently, if a system operation is disrupted by a crash, strict-linearizability mandates that the in-progress operation either follows through in a linearized manner or completely aborts as of the crash.

Definition 2.2.4 (Strict Linearizability [1]). A well-formed history H is strictly linearizable if there exists an extended history H' , derived from H , such that:

- For every crash event C in H and for each operation op invoked before C without a corresponding response before C (i.e., $inv(op) \prec_H C$ and $res(op) \not\prec_H C$), one of the following conditions holds:
 - There is a response r for op in H' such that $inv(op) \prec_H r \prec_H C$,
 - Following crash C within H' , $inv(op)$ is absent, indicating that the operation was discarded.
- Furthermore, for any two operations op_1 and op_2 present in both H and H' , if $op_1 <_H op_2$, then $op_1 <_{H'} op_2$.

Persistent Atomicity, proposed by Guerraoui and Levy [23], ensures that object states remain consistent even after system crashes by emphasizing atomic operations. However, it lacks *locality*, potentially leading to incorrect merged histories of different objects. In contrast, *Recoverable Linearizability*, introduced by Berryhill, Golab, and Tripunitara [5], retains post-crash consistency while incorporating *locality*, allowing for the correct merging of object histories. These approaches contribute to the broader concept of *Durable Linearizability* in systems vulnerable to crashes. *Durable Linearizability* [30] combines the guarantees of persistent atomicity and recoverable linearizability, ensuring system state consistency after a “full-system” crash failure where all processes fail simultaneously.

In the previous definitions, we operated under the assumption of a bounded set of processes. Additionally, after a system-wide crash failure, processes would “resurrect,” meaning they would recover and continue operation using the same IDs they held before

the failure. However, the next two definitions will deviate from these assumptions. Here, processes or threads can reuse the same ID once a previous one is completed. Moreover, after a crash, the entire set could be replaced with new processes or threads.

Definition 2.2.5 (*Durable Linearizability [30]*). *A history H is durably linearizable if and only if:*

- H is well-formed.
- The projection of H obtained by excluding all crash events is linearizable.

While classic linearizability ensures operations on shared objects appear instantaneous between their start and end, the “durable” version extends this assurance to persistent memory operations. *Durable Linearizability* provides a correctness property for crash recovery, yet it may face latency issues when operations must persist before returning to their callers. Addressing this, *Buffered Durable Linearizability* [30] was conceptualized. This variant refines the traditional linearizability concept [26], focusing on persistent memory operations and their resilience post-crash. The term “buffered” stems from the employment of a write buffer in contemporary systems. Data is first written to a buffer to improve performance, and then later flushed to the persistent memory for durability. This layer introduces complexity, given the flushing order may deviate from the initial write sequence.

Definition 2.2.6 (*Consistent Cut [30]*). *Given a history H and any transitive order $\prec$ on events of H , a $\prec$ -consistent cut of H is a projection P of H such that, if $E \in P$ and $E' \prec E$ in H , then $E' \in P$ and $E' \prec E$ in P .*

Definition 2.2.7 (*Buffered Durable Linearizability [30]*). *A history H is partitioned by c crash events, each indicated by C_i where i ranges from 1 to c , as follows:*

$$H = \mathcal{E}_0 C_1 \mathcal{E}_1 C_2 \dots \mathcal{E}_{c-1} C_c \mathcal{E}_c$$

Here, $\mathcal{E}_i$ denotes the i -th era of the history H , representing the sequence of events between the $(i - 1)$ -th and the i -th crash events.

The history H is buffered durably linearizable if and only if:

- H is well-formed.
- There exist projections $P_0, P_1, \dots, P_{c-1}$ of H such that for all $0 \leq i \leq c$:

- P_i is a $\prec$ -consistent cut of $\mathcal{E}_i$.
- Concatenating the projections as $P = P_0P_1 \dots P_{i-1}\mathcal{E}_i$ results in a linearizable history.

In buffered durable linearizability, an operation is only recognized as complete when its effects have moved from the write buffer to the persistent memory, rather than just at the end of its execution. This ensures that the results of completed operations remain intact, even after a system crash. A system that adheres to “buffered durably linearizable” standards appears linearizable, taking into account both program operations and the transitions from buffer to persistent storage. This model offers a robust consistency safeguard against system failures [72].

2.3 Montage

Montage [72] is designed to use buffered durable linearizability to maintain the integrity and order of data operations. It divides the execution of operations into epochs, with each epoch boundary representing a consistent cut of the happens-before relationship among operations. When a crash occurs, Montage arranges to recover all managed data structures to their state as of some common epoch boundary. Montage also ensures that all payloads (the subset of data needed to restore the data structure state) created or modified by a given operation are labelled with the same epoch number, which is crucial for maintaining the order of operations. Furthermore, Montage ensures that all payloads created or modified in a given epoch persist together, instantaneously, when the epoch clock ticks over.

2.3.1 Payload

In the context of the Montage system, a “payload” refers to a block of persistent data that Montage manages. The payloads contain the necessary information to capture the state of a data structure, such as items in a set, key-value pairs in a mapping, or vertices and edges in a graph. Payloads can be of different types, including:

- **ALLOC:** This indicates a payload that has been newly allocated.
- **UPDATE:** This represents the need to modify an existing payload that was allocated during a previous epoch.

- **DELETE:** These specific payloads exist for precisely two additional epochs, after which the payload they are nullifying can be securely reclaimed.

Payloads are labelled with the epoch during which they were either created or last modified. This labelling enables Montage to maintain the consistency of its managed data structures, even in the event of system crashes. Following a crash, Montage’s recovery mechanism retrieves all payloads from up to two epochs prior, guaranteeing the consistency of the restored data [72]. This is achieved by extracting the epoch numbers from all payloads and discarding those associated with the two most recent epochs to ensure data integrity. Montage is constructed on top of the Ralloc persistent memory allocator [8]. The persistent memory allocation and freeing capabilities required by Montage are provided by Ralloc. It allows the program to allocate and deallocate persistent memory in a manner familiar to programmers from traditional DRAM allocations.

2.4 Ralloc

Ralloc [8] is a persistent memory allocator designed for applications using large amounts of persistent memory. It aims to provide an effective method for managing persistent memory, reducing external fragmentation, and enabling fast crash recovery. Ralloc persists all metadata and uses a tracing garbage collector to find blocks reachable from persistent roots [8]. Given that the memory preserves its contents even after system interruptions or failures, there is a need for a predetermined location to access the data upon system or application reboot. This function is fulfilled by the persistent root. The metadata is rebuilt during recovery, and the tracked blocks are maintained, with other blocks being reclaimed for future use. Ralloc memory-maps three contiguous regions into the process’s address space:

- **Superblock Region:** This is where the actual allocated memory resides. The size of this region is user-specified during memory-mapped file initialization. It must be at least 1 GB [8]. Superblocks are the basic units of memory allocation in the system, sized in multiples of 64 KB, and can be allocated or deallocated non-blockingly. The superblock region dynamically expands as more memory is required, up to a defined limit.
- **Descriptor Region:** This area contains metadata about the superblocks. Its size is determined by the number of superblocks in the superblock region, with one descriptor required per superblock. Here, we maintain details like the state of the

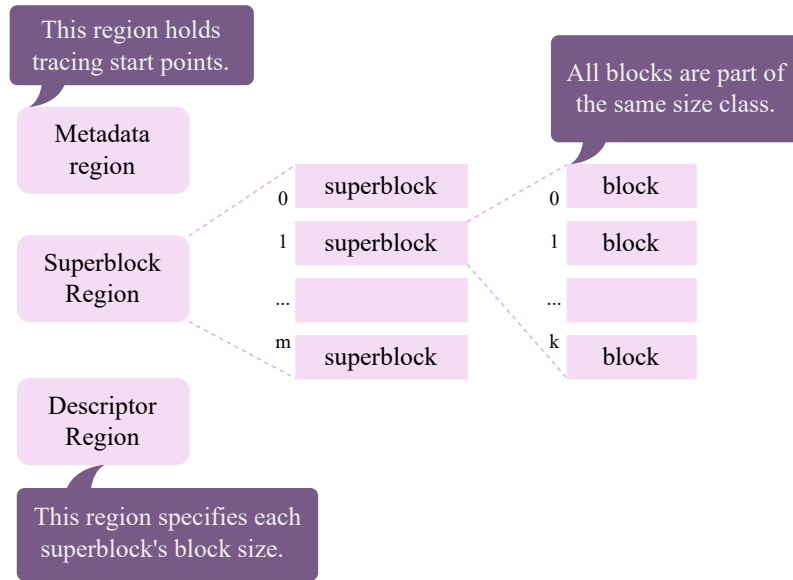


Figure 2.2: Ralloc regions

superblock, the size of its contained objects, and a pointer to the block's free list, among other relevant data.

- **Metadata Region:** With a size fixed at compile time, this region stores the allocator's critical data for crash recovery. This includes persistent roots and list heads, along with any additional metadata required by the garbage collector. It also holds a dirty bit that signals whether a crash occurred during a previous run. The region must be large enough to accommodate all persistent root metadata and any additional garbage collector's metadata.

Each of these regions serves a distinct purpose and is crucial to the efficient and safe operation of the Ralloc system. The management of these regions is designed to ensure consistency and recoverability in the event of system crashes, which is a key requirement for persistent memory systems. Ralloc supports both small and large allocations, dividing small allocations into several size classes for better efficiency. It also maintains temporary thread-local caches of blocks belonging to each size class to improve allocation performance [8]. Montage and Ralloc together form a powerful framework for creating robust and fault-tolerant persistent applications.

2.5 Recovery Mechanisms in Montage and Ralloc

Montage [72] and Ralloc [8] recovery routines are designed to ensure consistent data restoration. This section elaborates on their recovery mechanisms and how they integrate with each other.

2.5.1 Montage Recovery

1. **Two-Epoch Protocol:** As referenced from Nawab et al. [55], this protocol permits operations to overlap across two epochs, leading to a delay in memory reclamation and the generation of “*anti-payloads*”.
2. **Epoch Advancer:** This thread increments the epoch and, upon reaching epoch e , persists the payloads from epoch $e - 1$. As a result, the payloads from the two most recent epochs may not be explicitly persisted and are excluded during recovery to ensure data integrity.
3. **Memory Reclamation:**
 - (a) **Anti-Payload Creation:** Montage’s approach to memory reclamation revolves around the creation of “*anti-payloads*”. An anti-payload indicates a specific data payload designated for potential deletion in a subsequent epoch.
 - (b) **Progress without Disruptions:** Assuming an anti-payload is created at epoch e , the associated original payload is reclaimed during the transition from $e + 2$ to $e + 3$. The anti-payload itself is then discarded as the epoch progresses from $e + 3$ to $e + 4$.
 - (c) **System-Wide Crash Failure Handling:**
 - For an anti-payload generated at epoch e , if a crash occurs before persistence, the original payload is retained, and the anti-payload is discarded post-recovery.
 - If a system-wide crash failure arises at epoch $e + 2$ after persistence, both the original payload and the anti-payload are deallocated.

2.5.2 Ralloc Recovery

Ralloc’s recovery mechanism is grounded in its three primary memory regions, leveraging tracing from persistent roots [8].

1. **Reinitialization:** As a first step, Ralloc remaps its memory regions into the system, setting the stage for the recovery process.
2. **Tracing Reachable Blocks:** Through tracing, Ralloc identifies valid data blocks that are accessible from the persistent roots. This ensures that only relevant memory blocks are reconstructed.
3. **Descriptor Region Update:** Once the necessary blocks are identified, Ralloc reconstructs blocks and updates associated descriptors.

2.5.3 Montage and Ralloc Recovery Integration

Montage refines Ralloc’s recovery process. It discards all recovered blocks associated with the payloads from the last two epochs because their consistent persistence is not guaranteed. After this filtration, Montage receives the payloads, aiding in the reconstruction of the data structure state and the subsequent reestablishment of transient constructs.

Understanding the interplay between safety properties, memory management, and recovery mechanisms is vital for efficient and robust system design. Buffered Durable Linearizability, in particular, offers a compelling model that ensures system consistency. Its practical implementation in systems like Montage and its integration with persistent memory allocators like Ralloc highlights the effectiveness of these principles in real-world applications. The recovery strategies discussed offer insight into how such systems manage to maintain consistency and continue operations after disruptive events.

Chapter 3

Consistency Models for Snapshots

This chapter will focus on the study of two snapshot correctness criteria, which are essential for understanding the correctness requirements needed for our platform: instantaneous snapshotting and quasi-instantaneous snapshotting [56]. We will discuss how these criteria, while useful in many contexts, can fall short when dealing with snapshots on a platform that is buffered durable linearizable. In the last section of this chapter, we propose a new and more appropriate definition for snapshot correctness, tailored to this particular setting.

Snapshots play a vital role in ensuring failure recovery by duplicating memory regions as a precautionary action against system crashes. These precise duplicates could then be used to recover the entire memory system following a crash. However, copying everything at once is impractical, and stopping all operations for too long can make the system unresponsive. Therefore, we need a snapshotting process that can achieve its goal without causing these issues. To design such a process, we first need to understand what “**correctness**” means in this context. In the following sections, we will develop the concepts and terminology needed to give a precise definition of snapshot correctness for a buffered-durable platform.

3.1 Model

Here are the symbols used in this chapter, largely based on the notation from [16, 76, 56]:

- $R = r_1, \dots, r_m$: This denotes the memory regions in the system.
- $e = (p_i, r_j)$: This is an event which signifies that process p_i performed a read/write operation on memory region r_j .

- E : This is the set of all events.
- *Timestamp*: A timestamp is a unique identifier representing a specific point in time, expressed as a natural number.
- T is the set of timestamps corresponding to the execution. Each event that occurs is assigned a value from T , represented by t . This assigns a unique timestamp to each event, which establishes a chronological sequence among all events.
- *Universal Snapshot* $s_U : R \rightarrow V_{all} \times T_{all}$: This **function** maps a memory region to the copied value and the time it was copied, adopted from the definition proposed by Vömel and Freiling [68]. We denote by $s(r).v$ the value stored for region r in snapshot s and by $s(r).t$ the corresponding time.
 - V_{all} represents the collection of all possible values.
 - T_{all} represents the set of all natural numbers, equivalent to all possible timestamps.
 - $\mathcal{U}$ is defined as the Cartesian product of sets V_{all} and T_{all} , i.e., $\mathcal{U} = V_{all} \times T_{all}$, representing the universal set of all possible snapshots.
- *Snapshots Between a and b (i.e., $[a, b]$)*: These snapshots are a specialized version of the universal snapshot, limited to values and times within the interval $[a, b]$ in a specific execution. Represented by $s_{[a,b]} : R \rightarrow V_{[a,b]} \times T_{[a,b]}$, where:
 - $V_{[a,b]} \subset V_{all}$ represents the values that appeared between times a and b .
 - $T_{[a,b]} \subset T_{all}$ represents the set of natural numbers between a and b , equivalent to the timestamps during this interval.
 - $\mathcal{U}_{[a,b]} \subset \mathcal{U}$ is defined as the Cartesian product of sets $V_{[a,b]}$ and $T_{[a,b]}$, i.e., $\mathcal{U}_{[a,b]} = V_{[a,b]} \times T_{[a,b]}$, representing the possible snapshots between times a and b .
- In order to retrieve the value of a memory region r at a specific point in time t captured by snapshot s , we use the notation $s(r).t$.
- This chapter uses space/time diagrams based on the work of Mattern [47], where time is represented from left to right. Each horizontal line corresponds to a specific memory region, and a dot on the line signifies an event occurring at a particular time by a process (see Figure 3.1 for an example).
- The “*happened-before*” relation, denoted as $e \rightarrow e'$, captures the causal order between two events. Specifically:

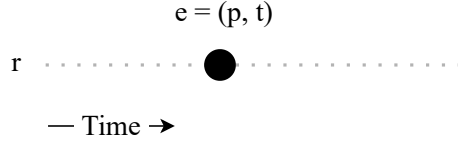


Figure 3.1: Process p performs a read/write operation in memory region r at the time t

- If event e occurs before event e' within the same *process*, then e chronologically precedes e' , resulting in $e \rightarrow e'$.
- When e and e' are executed by different processes and interact with the same *memory region*, the direction of the arrow indicates the causal order. To elaborate, if the changes or results from event e can be observed or have impacted the outcome of event e' , then we denote this causal relationship as $e \rightarrow e'$.
- If neither $e \rightarrow e'$ nor $e' \rightarrow e$ is established, it signifies that events e and e' are “concurrent”.

This relation is foundational and derives from Lamport’s work on the logical clock, helping to establish a partial order of events in a distributed system [35].

- *Consistent Cut Example:* The definition of a consistent cut can be found in Definition 2.2.2. An example of a consistent cut is illustrated in Figure 3.2; it is evident that if there are two events with a causal dependency, there are four possible ways to draw a cut. Among these four options, only c_1 represents an *inconsistent cut*, as it includes an event (e_2) without including its cause (e_1). On the other hand, the remaining three cuts (c_0 , c_2 , and c_3) adhere to the condition of a *consistent cut*, where all events in the past (left side of the cut) have their corresponding causes also in the past. Additionally, c_0 represents an *initial cut*, which does not contain any events.

3.2 Existing Consistency Definitions

Causally consistent snapshots ensure that the snapshots reflect the cause-and-effect relationships between events [68]. This type of consistency is tailored for systems where real-world time is absent, focusing instead on sequences of causally related events. In contrast, Ottmann et al. [56] propose an approach that emphasizes the exact time each memory region is duplicated, drawing inspiration from Scott D. Stoller’s work [65]. Within

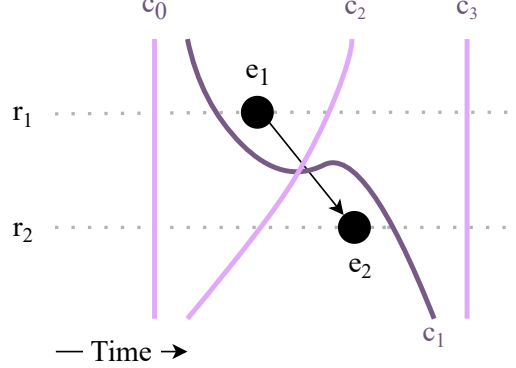


Figure 3.2: This figure demonstrates four potential cuts among two causally dependent events, identifying one inconsistent cut (c_1) and three consistent cuts (c_0 , c_2 , and c_3).

this context, the “*actually happened before*” relation in distributed systems emerges, seeking to capture the objective reality of event sequences. Rather than merely providing a logical sequence, this relation aims to record the true chronological order of events in a distributed environment. For systems grounded in real-time, this approach leverages real-time clocks to introduce a stricter ordering. This ordering essentially facilitates a deterministic assessment of event sequences. Based on this work, the simplest definition for ensuring the correctness of a snapshot is based on the assumption that all memory regions can be copied simultaneously, without any modifications occurring during the copying process.

Definition 3.2.1 (*Instantaneous Consistency* [56]). *A snapshot s satisfies instantaneous consistency if all memory regions in s were copied at the same point in time. Formally:*

$$\forall r, r' \in R, s(r).t = s(r').t$$

The given definition implies that copying all memory regions simultaneously is necessary. This requirement relies on hardware support to enable such a copy while freezing memory changes (as shown in Figure 3.3). However, meeting this criterion can be challenging, if not impossible, particularly when dealing with a large number of memory regions. As a result, Ottmann et al. [56] introduced an alternative definition known as “*Quasi-instantaneous Consistency*”, which offers a weaker requirement compared to instantaneous consistency.

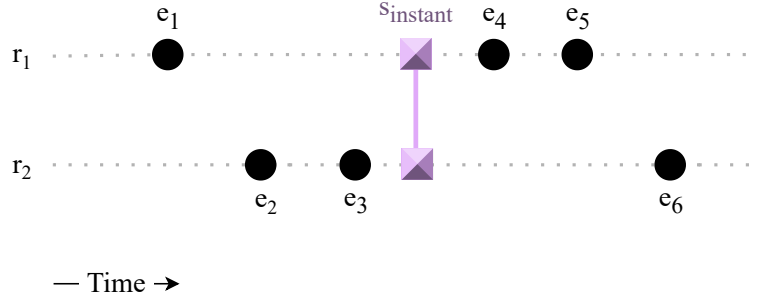


Figure 3.3: Instantaneous snapshot

Definition 3.2.2 (Quasi-instantaneous Consistency [56]). A snapshot s satisfies quasi-instantaneous consistency if and only if the values in the snapshot could have also been acquired with an instantaneous snapshot s' . Formally:

$$\exists s' \in \mathcal{U} : (\forall r, r' \in R : s'(r).t = s'(r').t) \wedge (\forall r \in R : s'(r).v = s(r).v)$$

The concept of quasi-instantaneous consistency allows for a flexible duration during the snapshotting process, eliminating the need to copy all memory regions simultaneously. However, this definition still requires the existence of an equivalent instantaneous snapshot. In other words, there should be a specific point in time where the values stored in all memory regions, according to the instantaneous snapshot, are equal to the values in the quasi-instantaneous snapshot.

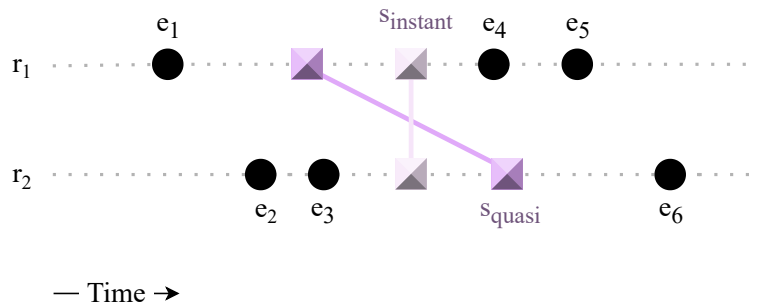


Figure 3.4: Consistent quasi-instantaneous snapshot

Figure 3.4 demonstrates that the quasi-instantaneous snapshot (s_{quasi}) has an equivalent instantaneous snapshot (s_{inst}) that contains exactly the same set of events as the

quasi-instantaneous snapshot. This indicates a *consistent* quasi-instantaneous snapshot. However, in Figure 3.5, we observe an *inconsistent* quasi-instantaneous snapshot (s), where there is no possible instantaneous snapshot that can match the contents of the taken snapshot. This is because there is no way to draw a vertical line (which would represent an instantaneous snapshot) that includes e_4 but excludes e_3 as e_3 appears before e_4 in the figure.

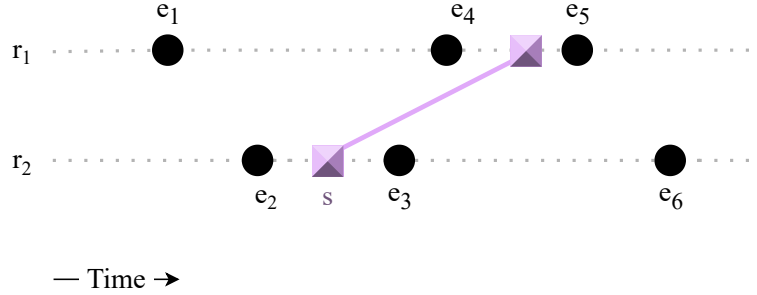


Figure 3.5: Inconsistent quasi-instantaneous snapshot

3.3 Buffered Durable Consistency

Existing snapshotting criteria, namely instantaneous (Definition 3.2.1) and quasi-instantaneous (Definition 3.2.2) consistency, are insufficient for platforms operating on a *buffered durable linearizable* model like Montage, which will be aptly explored in the following paragraphs. In these platforms, the recovery routine plays a crucial role by discarding operations that have not been fully persisted before a system crash. Moreover, none of the mentioned criteria exclude an empty snapshot that is taken at the beginning of the execution; i.e., they do not impose any constraint on the time when the snapshot has been taken.

The concept of instantaneous consistency demands a simultaneous copy of all memory regions, affording a consistent view of the memory state¹ at a specific point in time. However, this approach faces two significant challenges: (1) it is practically infeasible to achieve a truly simultaneous operation, and (2) it can conflict with the functioning of a recovery routine. Specifically, the recovery process might discard or modify partially completed operations that were captured by a snapshot. This action can lead to inconsistencies

¹By “memory state,” we refer to the actual values written to the main memory, excluding those in the caches.

between the snapshot and the actual memory state after recovery, as the recovery routine manipulates the memory state to ensure overall consistency.

On the other hand, quasi-instantaneous consistency assumes the existence of an instantaneous snapshot during the snapshotting process. However, this model fails to account for potential modifications made by the recovery routine to maintain consistency. These modifications can alter the snapshot to the extent that it no longer accurately reflects the memory state at any time during the recovery, thereby violating the quasi-instantaneous definition (Definition 3.2.2). Therefore, it is necessary to devise a new definition for snapshot correctness on a buffered-durable platform. The objective is to produce a snapshot that is as consistent as an instantaneous snapshot. This instantaneous snapshot could have been taken prior to the initiation of the snapshotting process and should represent the most recent consistent cut in the execution history. Furthermore, it must be consistent with the memory state at a point in time where all the not-fully-persisted operations have been discarded, aligning with the principles of buffered durable linearizability.

3.3.1 Example

In a buffered-durable linearizable platform, events can be classified as either fully-persisted or not-fully-persisted. Fully persisted events indicate that they have been explicitly written back to the memory, ensuring their durability. On the other hand, not-fully-persisted events refer to those that may or may not have been written back yet. Figure 3.6 illustrates this distinction, where events e_1 and e_2 represent fully persisted events, while the remaining events are still in the process of being fully persisted.

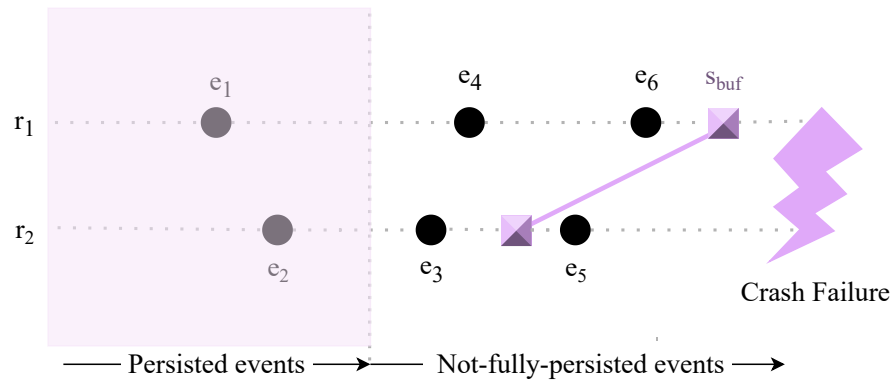


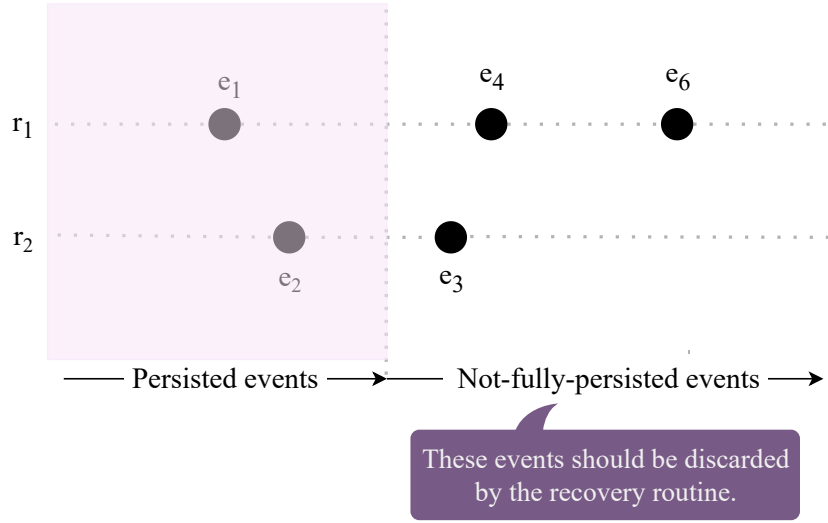
Figure 3.6: Snapshotting in a buffered-durable platform

In Figure 3.6, s_{buf} represents a snapshot that has been taken during the execution of a program that uses a buffered durable linearizable data structure. It started at some point after the most recent consistent cut in the execution history (i.e., the fully-persisted region). Therefore, it contains all the events in the fully-persisted region and includes some events (but not all of them) from the not-fully-persisted region. As illustrated in Figure 3.6, there is no way to take an instantaneous snapshot (represented by drawing a vertical line) that respects the ordering between events. Specifically, such a snapshot might include e_6 without also including e_5 . This is in contradiction with the concept of “*actually happened before*” [65], where the chronological order of events determines their ordering. Hence, if a snapshotting process begins by capturing all the events, it does not conform to the quasi-instantaneous consistency model since finding an equivalent instantaneous snapshot with the same events as s_{buf} is not possible. As a result, a buffered durable linearizable platform demands a different form of correctness. It requires a memory state that matches a consistent **prefix** of the execution history, not an instantaneous snapshot taken **during** the snapshotting process, which is what the quasi-instantaneous consistency model dictates.

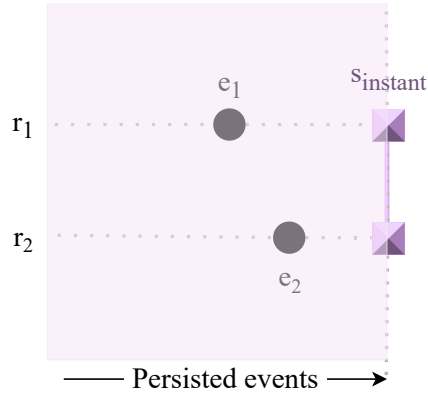
Consider Figure 3.7a, which depicts the memory state after a crash. Here, the not-fully-persisted operation e_5 is lost. The recovery routine discards all changes from not-fully-persisted events, as represented in Figure 3.7b. The resulting memory state, therefore, aligns with an instantaneous snapshot taken just after the most recent consistent cut. When we say a snapshot is taken right after the consistent cut, we mean that there exists a point in time until which no operations have occurred on any of the memory regions since the consistent cut. In other words, this snapshot includes all the events in the fully-persisted region and excludes all the events in the not-fully-persisted region. This snapshot fits better with the concept of buffered durable linearizability. In summary, buffered-durable data structures introduce a challenge where some events are visible but not yet persisted, risking snapshot inconsistencies post-recovery. There is a pressing need for a new snapshot correctness property that ensures consistent snapshot of a system state after potential crashes.

3.3.2 Terminology and Notation

- **Abstract State:** It refers to the conceptual representation of the state of a data type following a sequence of operations. This state embodies the effects of a series of operations and is often used to represent high-level behaviours.
- **Quiescent State:** This is a state where no operations are currently active on the



(a) Memory region states after restarting the program



(b) Recovery routine result: Memory region states that align with an instantaneous snapshot (s_{inst}) from a past point in time.

Figure 3.7: Recovery in a buffered-durable platform

data structure DS . It is a moment of inactivity, often used as a checkpoint in understanding the system's behaviour.

- **Concrete State:** The physical or actual state of a data structure at a specific point in time. It represents the real-time status of memory and includes the current values of all memory regions associated with the data structure.
- **abstract Function:** A function that relates a quiescent concrete state of a data structure DS to the abstract state of data type T . This function provides a bridge from the physical state of a data structure to the conceptual level of the abstract data type.
- **recovery Routine:** In the context of buffered durable linearizability, the **recovery** routine takes a concrete state as input and generates a new concrete state. This new state adheres to the principles of buffered durable linearizability, reflecting a prefix of the system's execution history and effectively removing any events that have not been fully persisted [30]. The primary objective of this routine is to create a state that represents the consistent and persisted portion of the system's events.

3.3.3 Consistency Definition

Definition 3.3.1 (*Buffered-Durable Consistency*). Let t_{cons} denote the latest time at which all preceding operations were fully persisted, marking a consistent cut. For a snapshot s_{buf} that begins at t_{start} (with $t_{cons} \leq t_{start}$) and ends at t_{end} , s_{buf} satisfies buffered-durable consistency if, post-recovery, its state matches an instantaneous snapshot captured within the interval $[t_{cons}, t_{end}]$.

Formally:

$$\begin{aligned} & \exists s_{inst} \in \mathcal{U}_{[t_{cons}, t_{end}]} : \\ & (\forall r, r' \in R : s_{inst}(r).t = s_{inst}(r').t) \\ & \wedge (\mathbf{abstract}(s_{inst}) = \mathbf{abstract}(\mathbf{recovery}(s_{buf}))) \end{aligned}$$

The definition holds true when the abstract state of a data structure, obtained by running a recovery routine on s_{buf} , can also be derived from an instantaneous snapshot,

denoted as s_{inst} . Thus, consistency is confirmed by matching the abstract states between two snapshots: one resulting from a recovery process, and the other being an instantaneous snapshot taken at a certain time point. The timestamp t_{cons} is pivotal to our definition. It indicates the most recent point in the system’s execution timeline where a set of operations prior to that point were fully persisted. This timestamp also corresponds to a consistent cut of the history. Consequently, an empty snapshot will not be considered correct according to this definition unless the snapshot was taken before the first consistent persistence timestamp.

3.4 Consistency Model for Montage

To demonstrate the correctness of the snapshotting mechanisms in Montage, we need to understand several system components. A key element is Montage’s recovery routine. This routine discards payloads from the two most recent epochs using the Ralloc recovery mechanism, ensuring the system’s buffered-durable linearizability. It returns the data structure state to a consistent prefix of the execution history. When referring to a data structure’s state, we are referring to crucial data—specifically, Montage payloads—that are required for program recovery and are stored on persistent memory. It is also worth noting that if an anti-payload was persisted before the crash and then recovered during the recovery process, both the original payload and the anti-payload would be deallocated.

We can conceptualize the Montage’s recovery routine as a function, which we will call **MontageRecovery**. It takes a quiescent state post-crash (s_{buf}) and returns a concrete state of the data structure, excluding the payloads from the two most recent epochs:

$$\begin{aligned} s_{rec} &= \text{MontageRecovery}(s_{buf}) \\ abstract_{rec} &= \text{abstract}(s_{rec}) \end{aligned}$$

In Montage, epochs serve as units for time segmentation, with each payload being assigned with an epoch number to denote its most recent update. Suppose a snapshot is captured at epoch e . If we obtain a snapshot (s_{inst}) immediately following the transition of the epoch from $e - 2$ to $e - 1$, it would encapsulate the quiescent concrete state of the data structure. This state corresponds to the abstract state of the data structure just after the advancement in the epoch from $e - 2$ to $e - 1$, before any operations executed in epoch $e - 1$, as denoted by $\text{abstract}(s_{inst})$. Hence, the abstract state of the data structure following the epoch transition from $e - 2$ to $e - 1$, $\text{abstract}(s_{inst})$, matches the state obtained from the **MontageRecovery**, termed as $abstract_{rec}$. If we interpret epoch numbers as time associated with each region, we can formally describe our snapshotting consistency as follows:

3.4.1 Consistency Definition

Definition 3.4.1 (*Buffered-Durable Consistency in Montage*). *Let:*

- e be the epoch in which snapshotting ends,
- t_{e-1} be the time when the epoch counter transitions from $e - 2$ to $e - 1$,
- s_{buf} be a snapshot started copying at t_{start} and ending at $t_{end} > t_{e-1}$,
- **MontageRecovery** be Montage-specific function for recovery.

Then, the snapshot s_{buf} satisfies buffered-durable consistency in Montage if, following the epoch advancement t_{e-1} , there exists an instantaneous snapshot s_{inst} such that the abstract state derived from the state recovered from s_{buf} using Montage's recovery routine is equivalent to that derived from s_{inst} .

Formally:

$$\begin{aligned}
& \exists s_{rec}, s_{inst} \in \mathcal{U}_{[t_{e-1}, t_{end}]} : \\
& (\forall r, r' \in R : s_{inst}(r).t = s_{inst}(r').t) \\
& \wedge (\forall r \in R : s_{inst}(r).t \geq t_{e-1}) \\
& \wedge (\mathbf{abstract}(\mathbf{MontageRecovery}(s_{buf})) = \mathbf{abstract}(s_{inst}))
\end{aligned}$$

The provided formal expression faithfully adheres to the buffered-durable consistency definition, Definition 3.3.1. For each region r in the set R , the time associated with r in the instantaneous snapshot s_{inst} remains identical. This statement corresponds with the conditions in the initial expression that indicate the time for all regions in s_{inst} is equal to the time right after epoch advancement, t_{e-1} . In the definition, the recovery of the taken snapshot s_{buf} yields the reconstructed snapshot s_{rec} . This process is mirrored in the above expression, where the **MontageRecovery** function conducts the recovery, and the **MontageConcrete** function employs the recovered payloads to assemble s_{rec} . Lastly, the abstract state derived from s_{rec} aligns with the one derived from s_{inst} , which is explicitly confirmed in the above-mentioned expression. Therefore, by satisfying these conditions, the formal expression aligns with the criteria of buffered-durable consistency. Hence, for the proposed algorithms in the upcoming chapters, our focus is primarily on ensuring that a snapshot completed in epoch e captures all payloads from $e - 2$ and previous epochs. Once

this condition is met, we can confidently assert that the snapshot also satisfies buffered-durable consistency.

In conclusion, we have explored the limitations and issues associated with the instantaneous and quasi-instantaneous snapshotting approaches and have put forth a novel definition for snapshot correctness pertinent to buffered-durable linearizable platforms - the *Buffered-Durable Consistency*. This model aims to establish a correlation between the concrete and abstract states of data, generating a snapshot that precisely mirrors the memory state at a certain moment, after all the not-fully-persisted operations have been omitted. In the upcoming chapters, we aim to prove that our proposed algorithms adhere to this specific definition for Montage, Definition [3.4.1](#).

Chapter 4

Stop-the-World Snapshotting

In this chapter, our emphasis will be on a particular type of snapshotting called “*stop-the-world*” snapshotting. This approach necessitates that all modifications to the data structure’s state be temporarily halted during the snapshotting process. In the literature, the term “*synchronous snapshotting*” is also used to describe this type of snapshotting [51].

In the context of stop-the-world snapshotting, ensuring the durability of data requires pausing the program from making any modifications to the memory-mapped files utilized by Ralloc [8]. Then, these files are copied onto an alternative PMEM module or HDD. This measure serves to mitigate the risk of persistent memory failure. Our objective is to identify the specific functions within Montage that control changes to the data structure’s state, and prevent them from modifying the program state while the snapshotter is copying the memory-mapped files. The purpose of halting the program is to ensure a consistent copy of the memory-mapped files, a term that denotes a snapshot that accurately reflects the data structure’s state at a specific moment in time. In other words, with this approach, we ensure that the snapshotting algorithm fully complies with the Buffered Durable Consistency definition proposed in Chapter 3.

In subsequent sections, we will delve into the exploration of Ralloc regions, and detail the alterations we have implemented in Montage and Ralloc to facilitate stop-the-world snapshotting. In the final section, we will present formal proof of the correctness of the snapshot, which is based on the Definition 3.4.1.

4.1 Ralloc Regions

Extending the discussion from Chapter 2, we understand that Ralloc is constructed around three distinct regions: the superblock, descriptor, and metadata.

The *Superblock Region* is the largest and is where the actual allocated memory resides. However, its size poses challenges to snapshotting; replicating the entire superblock region every time a snapshot is taken compromises efficiency. As a workaround, only the sections of the superblock region that have undergone changes since the previous snapshot are duplicated during the snapshotting process. Following in size, the *Descriptor Region* contains metadata about these superblocks, linking the allocated memory with its administrative data. Lastly, the *Metadata Region* is the smallest region and has a size that is fixed at compile time.

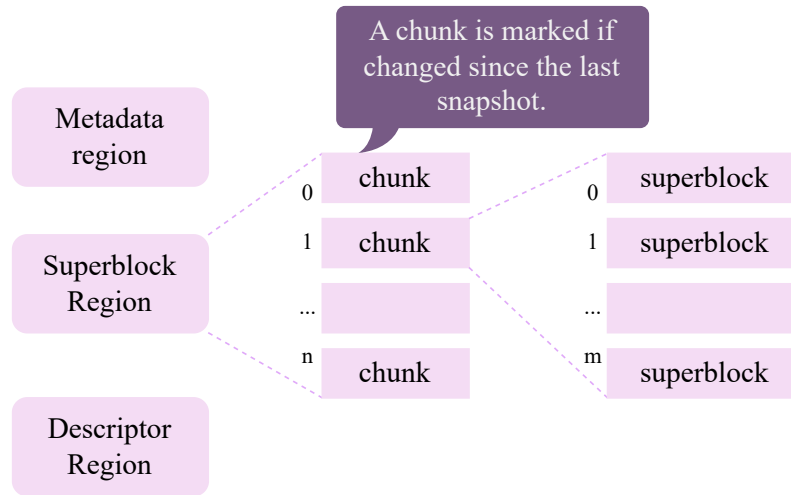


Figure 4.1: Chunks in the memory-mapped files

The first step in implementing this functionality is to identify the Montage functions that modify the memory-mapped files. The code is instrumented to track changes to the memory-mapped files, allowing the program to detect the **chunks** (each chunk is a multiple of superblocks, as shown in Figure 4.1) that have been changed since the last snapshot. The process of creating database snapshots in SQL Server involves a method reminiscent of “file chunking”. In this approach, whenever a database page is modified, the original unaltered page is transferred to a separate sparse file [49]. In our approach, only the chunks marked as modified are copied during the snapshotting process, eliminating the need to replicate the entire memory-mapped file each time. The marked chunks are also unmarked

after the snapshotting process. This results in greater efficiency and reduces the amount of data stored by ensuring that only the actual changes to the superblock region are copied during the snapshot operation (as shown in Figure 4.2).

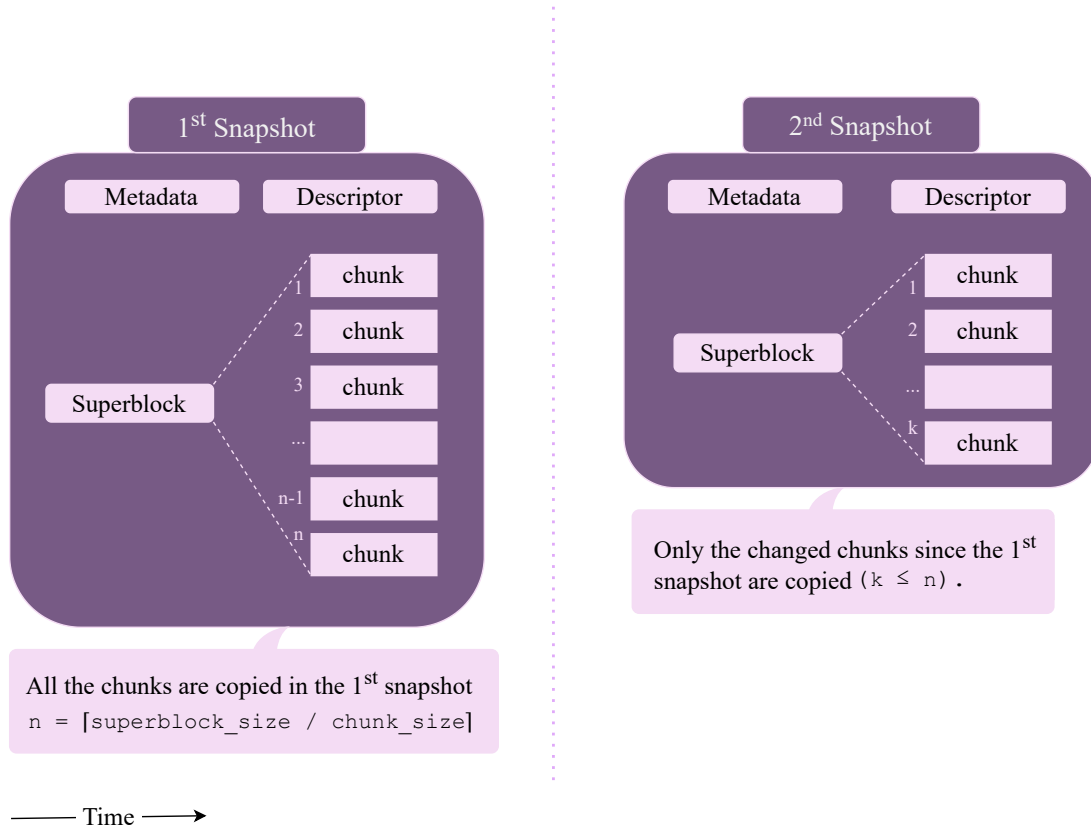


Figure 4.2: First two snapshots

A dedicated thread is assigned the task of periodically capturing the system's snapshot, with the timing dictated by a command-line argument. A **global** reader/writer lock is used to maintain synchronization between the snapshotter thread and the application threads (threads that are engaged in modifying the data structure state in Montage). In this approach, the application threads attempt to modify the memory-mapped files while obtaining the lock in shared mode. The snapshotter thread obtains the same lock in exclusive mode when the snapshotting process is started. This ensures that while the memory-mapped files are being copied, no other threads may make changes to them. As a result, after the snapshotting operation is finished, a consistent copy of the memory-mapped files is obtained (as shown in Figure 4.3).

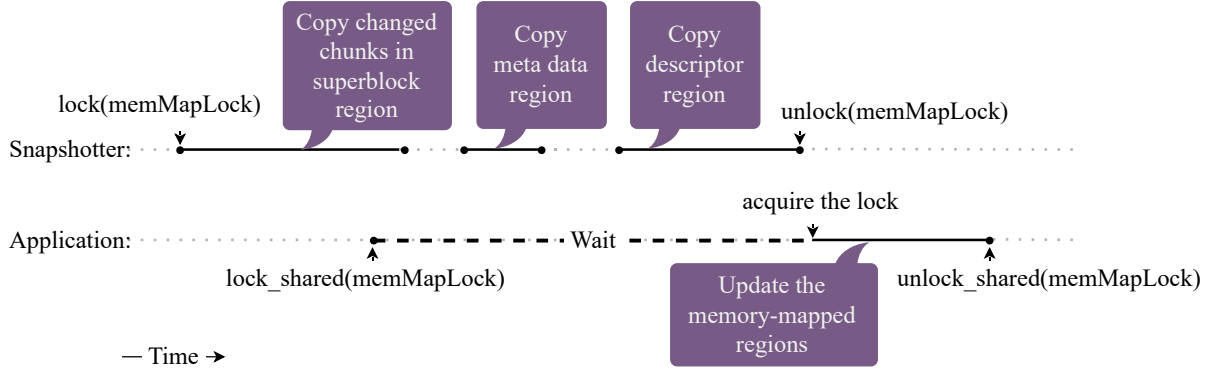


Figure 4.3: A reader/writer lock ensures consistency during snapshotting by allowing application threads to modify memory-mapped files in shared mode while the snapshotter thread acquires the lock in exclusive mode.

4.2 Modifications to Montage

We implemented our work using a variation of the original Montage [73]. Each thread in this variant buffers up to 64 payload write-backs per epoch. If the buffer becomes full before adding a new entry, it writes back the oldest one. Moreover, it includes a background thread that manages the advancement of the epoch and handles any remaining write-backs. In this variation, the reclamation of freed payloads and anti-payloads is also taken care of by worker threads.

When the snapshotter begins copying, the program disallows any explicit write-backs on persistent memory. If a snapshot is initiated at global epoch e , no further changes will be persisted in the current epoch. In the event of a persistent memory module failure, we can replace the corrupted memory-mapped files with the backup ones and rely on Montage’s recovery routine to discard all payloads in epochs e and $e - 1$. It is crucial to underscore that this locking strategy does not inhibit the processor from autonomously writing back payloads to the primary persistent memory. In other words, it is a function managed automatically by the processor’s memory management unit (MMU) to optimize the speed of memory accesses. As a result, changes in the persistent memory may occur during the snapshotting process. However, since all these changes take place in epochs greater than or equal to e , we rely on Montage’s recovery routine and buffered durable linearizability definition [30] to discard the not-fully-persisted suffix of the history. This ensures the overall system’s consistency and correctness. The proof of the snapshot’s correctness, based on the definition proposed in Section 3.3.1, is provided in Section 4.4.

Algorithm 1: This function is responsible for persisting the modified cache line to persistent memory using (`clwb`) instruction.

```
1 Function doPersistProtected(addr):  
    Input: addr: The address of the memory page  
2    memMapLock.lockShared() // Acquire the lock in shared mode  
3    clwb(addr) // Persist the modified memory page  
4    markChunk(addr) // Mark the changed chunk  
5    memMapLock.unlockShared() // Release the lock from shared mode
```

The `doPersistProtected` function, Algorithm 1, utilizes the cache line write-back instruction `clwb` to store the updated cache line in persistent memory while concurrently acquiring and releasing a shared lock from the snapshotter object during the operation. In scenarios where multiple consecutive cache lines require writing back to the persistent memory and they correspond to the same payload, a specific function comes into play. This function employs a for loop, iterating over multiple cache lines for each payload update. Notably, these multiple cache line write-backs necessitate only one lock acquisition (global lock in shared mode). The design of this approach is aimed at improving system performance by lessening the frequency of lock acquisition and release operations. For simplicity and to maintain the overall clarity of our explanation without sacrificing the generality, our discussion has primarily revolved around functions that involve a single `clwb` instruction.

Algorithm 2: The `takeSnapshot` function is responsible for capturing a snapshot of the memory-mapped files.

```
6 Function takeSnapshot():  
    // Acquire the lock in exclusive mode  
    // It would halt the epoch advancer  
7    memMapLock.lockExclusive()  
8    parallel for i from 0 to numChunk - 1 do  
9        if chunkChanged(i) then  
10           copyChunk(i) // Copy the changed chunk  
  
    // Copy all data in the metadata region (indicated by META_IDX)  
11    copyAll(META_IDX)  
    // Copy all data in the descriptor region (indicated by DESC_IDX)  
12    copyAll(DESC_IDX)  
    // Release the lock from exclusive mode  
13    memMapLock.unlockExclusive()
```

The `takeSnapshot` function, Algorithm 2, is responsible for creating a snapshot of the memory-mapped files. It makes use of a lock to make sure that data consistency is upheld when the file is being accessed and modified by many threads at once. To prevent other threads from making changes to the memory-mapped file while the snapshot is being taken, the function takes an exclusive lock on the file. The exclusive lock is released when the snapshot has been taken, allowing other threads to read and amend the file. The `takeSnapshot` function parallelizes the copying of chunks with the number of threads determined as a command line argument. This results in a faster and more scalable snapshotting process, with multiple threads working on different file chunks simultaneously. However, this approach is still limited by the speed of the memory controller and the PMEM module.

Algorithm 3: The `copyAll` function: Copies memory from a specified memory-mapped file (either Metadata or Descriptor region) to its backup file.

```

14 Function copyAll(heapIdx):
    Input : heapIdx: Index representing a specific region
    Global Variable: heapBaseAddr[]: Starting addresses of memory regions
    Global Variable: heapFileSize[]: Sizes of memory regions
    Global Variable: heapBackupPath[]: Backup file addresses in the file system
    // Get the primary chunk address
15 srcMemMap  $\leftarrow$  heapBaseAddr[heapIdx]
    // Get the size of the heap file
16 srcSize  $\leftarrow$  heapFileSize[heapIdx]
    // Get the backup chunk address
17 destMemMap  $\leftarrow$  getMemMap(heapBackupPath[heapIdx], srcSize)
    // As this function is only called for Metadata or Descriptor
    regions, copy entire content to backup
18 memcpy(destMemMap.addr, srcMemMap, srcSize)

```

Algorithm 4: The `copyChunk` function is responsible for copying only the portions of the memory-mapped file that have been changed.

```
19 Function copyChunk(chunkId):  
    Input: chunkId: The identifier of the chunk  
    // Get the primary chunk address  
20    srcAddr  $\leftarrow$  chunkAddresses[chunkId].srcAddr  
    // Get the size of the chunk to be copied  
21    size  $\leftarrow$  chunkAddresses[chunkId].size  
    // Get the backup chunk address  
22    destAddr  $\leftarrow$  chunkAddresses[chunkId].destAddr  
    // Copy the chunk  
23    memcpy(destAddr, srcAddr, size)  
    // Indicate that the chunk is no longer changed  
24    chunkChanged[chunkId] = false;
```

Algorithm 5: The `chunkChanged` function returns true if the chunk has been changed since the previous snapshot.

```
25 Function chunkChanged(chunkId):  
    Input: chunkId: The identifier of the chunk  
26    return chunkChanged[chunkId]
```

Algorithm 6: The `getMemMap` function returns a memory-mapped file for a specified address and size

```
27 Function getMemMap(filePath, size):  
    Input : filePath: Backup file path in the file system to be memory mapped  
    Input : size: The size of the memory chunk  
    Output: An instance of MemMapInfo with the memory-mapped information  
    // Open the file at the specified address  
28    destFd  $\leftarrow$  open(filePath);  
    // Change the size of the file to match the given size  
29    ftruncate(destFd, size)  
    // Create a memory map of the file  
30    destMemMap  $\leftarrow$  mmap(size, destFd)  
    // Return the memory map information  
31    return MemMapInfo(destMemMap, destFd)
```

Algorithm 7: Structure `MemMapInfo` used for holding memory mapping information.

```

32 Structure MemMapInfo:
33   srcAddr // Memory address of the primary chunk
34   destAddr // Memory address of the backup chunk
35   size // Size of the memory chunk

```

4.3 Modifications to Ralloc

In Ralloc, every pointer is associated with a 64-bit smart pointer, known as an **off-holder**, which retains an offset value rather than an absolute virtual address [8]. When a pointer is required, the offset is added to the region’s base address to determine the absolute address. During setup, Ralloc keeps track of the base addresses for the three persistent regions (metadata, descriptor, and superblock), enabling it to look them up when converting an **off-holder** to an absolute address. This guarantees that the same heap can be mapped into various virtual addresses by several processes and that the persistent data in the heap can still be accessed via the **off-holders** [8]. We changed the Ralloc class in our implementation such that it now returns the base addresses for the three regions of its memory-mapped files. The index of each chunk in the marking chunks was then determined using these base addresses. We subtracted the base address of the region from the address of the chunk and divided the resulting value by the size of the chunk to obtain the index (Algorithm 8).

Algorithm 8: The `markChunk` function sets the associated indicator of the appropriate chunk to `true`.

```

36 Function markChunk(addr):
37   Input: addr: The pointer to the start of the chunk
37   index  $\leftarrow$  getChunkIdx(addr)
37   // Mark the chunk as changed
38   if chunkChanged[index] == false then
39     chunkChanged[index]  $\leftarrow$  true

```

Algorithm 9: The `getChunkIdx` function calculates the index of the chunk by dividing the offset variable by `chunkSize`.

```

40 Function getChunkIdx(addr):
    Input : addr: The pointer to the start of the chunk
    Output: The chunk ID corresponding to this address is returned.
    // baseAddr[SB.IDX]: base address of the superblock region
41   offset ← addr − baseAddr[SB.IDX]
    // chunkSize is set by a CLI argument. The following line
    // calculates the index of the chunk
42   return floor(offset/chunkSize)

```

4.4 Correctness

The primary goal of this section is to determine how stop-the-world ensures that all changes from epoch $e-2$ and earlier are captured in the snapshot, based on the provided algorithms.

Theorem 4.4.1 (*Buffered-Durable Consistency for the Stop-the-World Snapshotting Mechanism on top of Montage*). *The stop-the-world snapshotting mechanism on top of Montage produces a snapshot that satisfies buffered-durable consistency per Definition 3.4.1.*

Proof. Let e be the epoch number when the snapshotter has started its process by acquiring the lock in exclusive mode. To prove the above theorem, we will examine the mechanisms and their effects in sequence:

Chunk Modification Tracking

1. When a memory page undergoes modifications, the `doPersistProtected` function is executed, and the global lock is acquired in shared mode (Line 2).
2. This function subsequently invokes the `markChunk` function (Line 4), marking the modified chunk.
3. Using the `chunkChanged` array, the corresponding index is set to `true` (Line 39), signaling any alterations in the memory page for copying.

Exclusive Snapshot Locking

1. The `takeSnapshot` function uses the `memMapLock.lockExclusive` function, ensuring the entire memory-mapped region is locked exclusively (Line 7).
2. This lock guarantees a consistent state of memory throughout the snapshot process, as no other threads or processes can modify the memory during this period.

Snapshot Procedure

1. After obtaining the lock, the snapshot process checks each chunk (Line 8).
2. It identifies if a chunk has been modified since the last snapshot using the `chunkChanged` function (Line 9).
3. Modified chunks are copied to their backup locations with the `copyChunk` function (Line 10).
4. Once copied, a chunk's status is reset, marking it as unchanged until the next modification (Line 24).

Epoch Guarantees

The epoch mechanism ensures that:

1. Any chunk that has been altered in epoch $e - 2$ or earlier is flagged as changed (as indicated in Algorithm 8). This is because, when the snapshotter begins copying the chunks at epoch e , Montage ensures that all the payloads from epoch $e - 2$ and earlier have been explicitly persisted.
2. During the snapshot in epoch e , all flagged chunks are copied (Line 8).
3. The snapshot s_{buf} captures all changes up to epoch $e - 2$ by copying the modified chunks since the last snapshot with `copyChunk`. Additionally, the entire descriptor and metadata regions are copied using `copyAll` (lines 11 and 12). This alignment ensures that when we recover s_{buf} , we receive $s_{\text{recovered}}$ that holds the state at the end of epoch $e - 2$.

Please note that when the snapshotter begins copying the memory-mapped files, it prevents any explicit cache-line write-backs, thereby preventing the epoch advancer from advancing the epoch counter. Consequently, when forming $s_{\text{recovered}}$, payloads from epochs e and $e - 1$ are discarded, preserving the end state of epoch $e - 2$. Applying **abstract** on $s_{\text{recovered}}$ provides the system’s abstract state at the end of epoch $e - 2$. Given that **abstract** is deterministic, identical states will produce identical abstract results. Hence, the Montage snapshotting mechanism ensures buffered-durable consistency as defined in Definition 3.4.1. $\square$

In conclusion, we have successfully improved the fault tolerance of Montage by integrating the stop-the-world snapshotting mechanism that concentrates on duplicating only the altered parts of the superblock region. The proposed alterations to Montage and Ralloc ensure data consistency and efficient parallelization of the snapshotting process. We also established the correctness of the stop-the-world snapshotting mechanism, demonstrating that it satisfies buffered-durable consistency as per a specified definition. As a result of these enhancements, our modified version of Montage is now capable of more effective recovery from persistent memory failures, thereby strengthening its reliability and resilience in managing persistent memory for various applications.

Chapter 5

Online Snapshotting

This chapter delves into the concept of “*Online Snapshotting*,” which offers a solution to mitigate the **responsiveness** issue encountered with the `takeSnapshot` method discussed in the previous chapter. Responsiveness refers to the system’s ability to react promptly to user inputs or requests, minimizing perceived delays. Stop-the-world strategy halts the program during snapshotting, adversely affecting the system’s responsiveness, particularly with larger data structures. To improve this, we present an alternative approach—online snapshotting. The essence of this technique lies in allowing modifications to the chunks that are not being replicated by the snapshotter, enhancing the system’s overall responsiveness. The following sections elucidate the synchronization techniques used to implement online snapshotting and maintaining the superblock region’s consistency. We consider various scenarios that could occur when an application thread attempts to modify a chunk and propose algorithms to maintain consistency within the system. Furthermore, the chapter demonstrates how the descriptor and metadata regions are snapshotted during the process, ensuring the comprehensive consistency of memory-mapped files. This chapter also presents a proof of the buffered-durable consistency of the snapshotting mechanism, according to Definition [3.4.1](#).

5.1 Synchronization

To implement online snapshotting, we propose several modifications to the `takeSnapshot` (Algorithm [13](#)) and `doPersistProtected` (Algorithm [10](#)) methods. Unlike the stop-the-world snapshotting approach which employs a single global reader/writer lock—obtained

exclusively by the snapshotter and shared by the application threads for updating memory-mapped files—online snapshotting introduces a unique reader/writer lock for **each chunk** of the superblock region. This lock is exclusively acquired when copying a specific chunk and subsequently released before moving on to the next chunk. Moreover, the corresponding lock is also acquired in shared mode by application threads when they attempt to modify a particular chunk.

During the snapshotting process and while a specific chunk is being copied, it is possible for application threads in Montage to change chunks and access the lock for other chunks in shared mode. It is important to note that these locks are only ensuring the consistency of each chunk by itself. As such, additional measures are needed to maintain consistency between different chunks. To fully grasp this, we will go through different scenarios more carefully in the next section.

5.2 Maintaining Superblock Region Consistency

Let us consider four potential situations that could occur when an application thread is trying to modify a chunk:

1. If there is no snapshotting occurring at the moment of modification, the chunk is marked as modified and the snapshotter will copy it in snapshotting.
2. Second, if a snapshotting process is ongoing, the behaviour depends on whether or not the chunk has been copied.
 - (a) If the chunk has not been copied, it is marked as modified and the snapshotter will deal with it later.
 - (b) If the chunk has already been copied, the primary memory-mapped file and the corresponding portion of the backup memory-mapped file are modified. A consistent copy of the chunk is maintained in this manner (see Figure 5.1a).
 - (c) If the chunk is being copied, it cannot acquire the lock in shared mode and must await the release of the exclusive lock before proceeding (as shown in Figure 5.1b).

Scenarios 1 and 2a correspond to the stop-the-world snapshotting approach. In these two instances, we simply need to mark the chunk as changed and allow the snapshotter

to copy the entire chunk on its turn (in other words, any modifications will be included in the snapshot simply by copying the whole chunk). Regarding scenario 2c, the locking mechanism prevents any modifications to the primary chunk at that particular moment. Once the lock is released, it also transitions into scenario 2b, which requires more detailed consideration.

Given that we use memory-mapped files to access persistent memory, we can directly access different sections of memory. This enables us to jump into a certain memory location and copy a segment of memory starting from there. Therefore, to address potential inconsistencies that could occur in scenario 2b, we update both the primary and backup chunks for the specific altered portion (i.e., not the entire chunk). This means if a chunk is modified after being copied, the corresponding segment in the backup chunk is also updated to reflect the new value in the primary chunk.

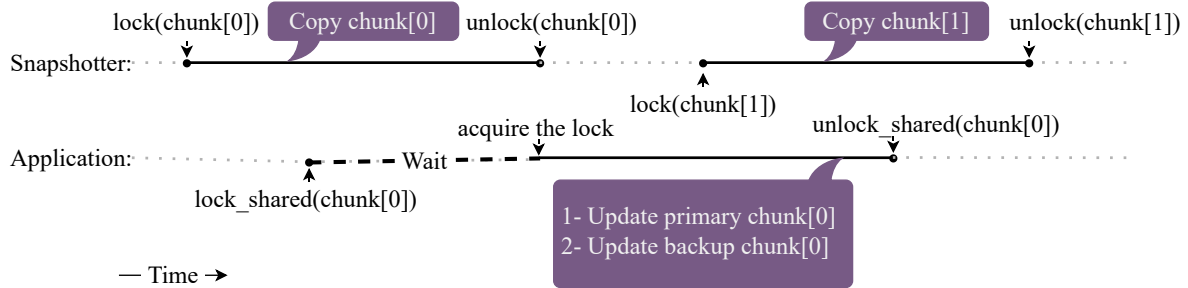
The function `doPersistProtected` needs to be modified on line 47 to accommodate for updates happening on chunks that have already been copied. This is checked on line 53. If the check indicates that the chunk has been copied, it is a match with scenario 2b. Here, we need to determine the appropriate starting position in the backup memory-mapped file and then perform a copy operation from the primary memory-mapped file to the backup file. To facilitate this, Algorithm 12 is employed to calculate the starting position in the backup file using the offset of the address in the source file.

Algorithm 10: The `doPersistProtected` function handles modifications to both the primary and backup memory-mapped files by calling `markOrUpdateChunk` inside `doPersist`.

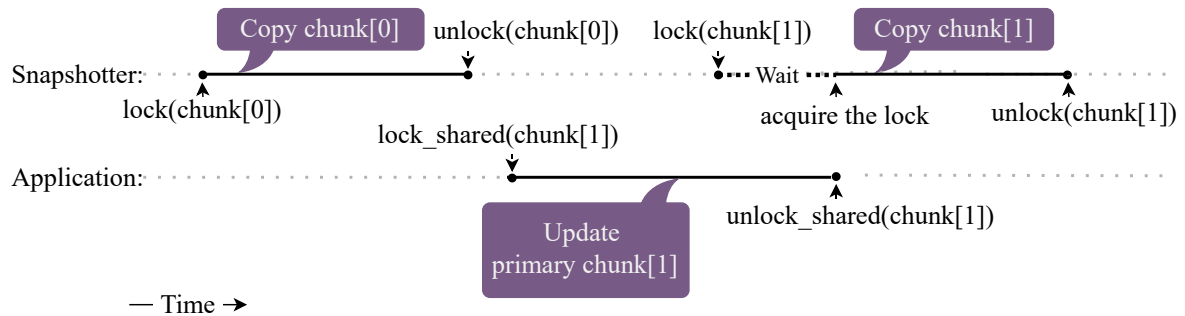
```

43 Function doPersistProtected(addr):
   |   Input: addr: The address to persist
44   |   chunkId  $\leftarrow$  getChunkIdx(addr)
45   |   memMapLock[chunkId].lockShared()
46   |   clwb(addr)
47   |   markOrUpdateChunk(addr)
48   |   memMapLock[chunkId].unlockShared()

```



(a) Updating both primary and backup chunks after the chunk is being copied



(b) Marking chunk as changed if it has not yet been copied

Figure 5.1: Online snapshotting

Algorithm 11: `markOrUpdateChunk` function maintains the consistency of the snapshot by ensuring that any changes to a primary chunk are reflected in the backup memory-mapped files.

```

49 Function markOrUpdateChunk(srcAddr):
    Input: srcAddr: The memory address of the chunk to mark or update
    // Get the size of the allocated memory by Ralloc
50 size  $\leftarrow$  mallocSize(srcAddr)
51 chunkId  $\leftarrow$  getChunkIdx(srcAddr)
52 if chunkChanged[chunkId]  $==$  false then
53     if chunkCopied[chunkId] then
54         // Get the backup address of the chunk
55         backupAdr  $\leftarrow$  getBackupAddress(srcAddr, chunkId)
56         // Copy the content of the chunk to the backup address
57         memcpy(backupAdr, srcAddr, size)
    else
        // Mark the chunk as changed
        chunkChanged[chunkId]  $\leftarrow$  true

```

Algorithm 12: The `getBackupAddress` function calculates and returns the backup address for a given source address and chunk ID.

```

58 Function getBackupAddress(addr, chunkId):
    Input : addr: The pointer to the location in memory
    Input : chunkId: The identifier of the chunk
    Output: It returns the backup address that corresponds to the provided
               address.
59 destAddr  $\leftarrow$  chunkAddresses[chunkId].destAddr
60 offset  $\leftarrow$  addr - chunkAddresses[chunkId].srcAddr
61 return destAddr + offset

```

5.3 Snapshotting Descriptor and Metadata Regions

Importantly, during the snapshotting process, the superblock region is not the sole area that needs duplication. To maintain consistency, the descriptor and metadata regions also require replication. Given that these files are notably smaller compared to the superblock region, after copying the superblock chunks and permitting application threads to alter

both primary and backup files, lines 63-68, all locks are acquired in an exclusive mode to prevent further changes to the superblock region. Then, replication of the descriptor and metadata regions, which are in sync with the superblock region, is performed. It is worth noting that when all locks are taken, no additional alterations can be made to the superblock region and their corresponding parts in the descriptor and metadata regions. Hence, once we enter the critical section of duplicating descriptor and metadata regions, these files align with the superblock region, which was duplicated consistently. Consequently, descriptor and metadata regions are copied maintaining coherence with the superblock region. Eventually, all locks are released and all opened backup memory-mapped files are closed, allowing the program to continue with its normal execution. This process is described in Algorithm 13.

Algorithm 13: The `takeSnapshot` function creates a consistent snapshot of the data structure’s memory state by distinguishing between duplicating changed memory chunks and copying the descriptor and metadata regions separately.

```

62 Function takeSnapshot():
63   for  $i$  from 0 to numChunk - 1 do
64     memMapLock[i].lockExclusive()
65     if chunkChanged[i] then
66       | copyChunk( $i$ )
67     chunkCopied[i]  $\leftarrow$  true
68     memMapLock[i].unlockExclusive()
    // Acquire all the locks in exclusive mode
69   for  $i$  from 0 to numChunk - 1 do
70     | memMapLock[i].lockExclusive()
    // Copy all data in the metadata region (indicated by META_IDX)
71   copyAll(META_IDX)
    // Copy all data in the descriptor region (indicated by DESC_IDX)
72   copyAll(DESC_IDX)
    // Release all the locks from exclusive mode
73   for  $i$  from 0 to numChunk - 1 do
74     | memMapLock[i].unlockExclusive()

```

5.4 Correctness

Online snapshotting in Montage is carefully orchestrated to capture a consistent state of the data structure, even when concurrent operations are in progress. This is accomplished through memory chunk management, reflected in Figures 5.1a and 5.1b, and further elaborated in the associated algorithms. The primary challenge is ensuring that the snapshot mechanism correctly duplicates any changed memory chunk and copies the descriptor and metadata regions consistently.

Theorem 5.4.1 (*Buffered-Durable Consistency for the Online Snapshotting Mechanism on top of Montage*). *The online snapshotting mechanism on top of Montage yields a snapshot adhering to buffered-durable consistency as per Definition 3.4.1.*

Proof. Let e be the epoch number when the snapshotter starts copying the descriptor and metadata regions by acquiring all the locks in exclusive mode. We aim to prove that the online snapshotting mechanism on top of Montage produces a snapshot that satisfies buffered-durable consistency per Definition 3.4.1.

1. Whenever a change occurs, the `doPersistProtected` function is invoked. This function ensures that both primary and backup memory-mapped files are updated.
2. Within the `doPersistProtected` function, the `markOrUpdateChunk` procedure determines if the relevant chunk has been copied (Line 53). If it has not been copied, it is marked as changed (Line 56). But if it has been copied, changes are propagated directly to the backup memory (Line 55). This action maintains the integrity of the snapshot, as illustrated in Figure 5.1a.
3. According to Figure 5.1b, if a chunk that has not been copied is altered, it is labelled as changed. This procedure ensures the `takeSnapshot` function identifies which chunks need duplication and which do not when iterating over all chunks (lines 63 to 68).
4. After all modified chunks are addressed, the snapshotting mechanism acquires all memory chunk locks to guarantee atomicity. Subsequently, the copying of the descriptor and metadata regions begins, as indicated by lines 70 to 72 in Algorithm 13. This step is crucial because it confirms the state of the copied descriptor and metadata regions corresponds with the state of the duplicated memory chunks in the superblock region. Therefore, if the snapshotting process initiates at epoch e , it encompasses all payloads from epoch $e - 2$ and before.

Please note that when the snapshotter begins copying the memory-mapped files, it prevents any explicit cache-line write-backs, thereby preventing the epoch advancer from advancing the epoch counter. Given the procedures detailed above, s_{buf} captures all changes up to epoch $e - 2$. Applying **abstract** to **MontageRecovery**(s_{buf}) yields the data structure’s abstract state at the end of epoch $e - 2$. Hence, Montage’s online snapshotting mechanism adheres to the buffered-durable consistency as per Definition 3.4.1.

□

5.5 Limitations

In the current implementation, each chunk is a multiple of superblocks. Ralloc utilizes superblocks, with each superblock holding blocks of exactly one class and having a size of 64 KB. The size classes support blocks ranging from 8 to 14 K bytes [8]. As a result, for these predefined size classes, one object cannot be shared between two consecutive chunks. When it comes to large objects, Ralloc rounds the size of a large allocation up to a multiple of the superblock size (64 KB) and allocates it by expanding the used space in the superblock region. Addressing our current implementation’s limitation of not supporting large allocations is included in our future work plan.

Through the course of this chapter, we thoroughly explored online snapshotting as an efficient methodology to enhance the responsiveness of the system during snapshotting. By providing each chunk in the superblock region with a unique reader/writer lock, we allowed application threads to modify the chunks not being replicated by the snapshotter. By considering several possible scenarios when an application thread modifies a chunk, we have discussed the required measures to ensure the system’s consistency. Moreover, we have presented an efficient approach for snapshotting descriptor and metadata regions, which further contributes to maintaining data consistency. The algorithms given in the chapter detail the step-by-step processes to achieve online snapshotting efficiently. Moreover, we have provided proof showing that our online snapshotting mechanism produces a snapshot that meets the requirements of buffered-durable consistency, as defined in Definition 3.4.1.

Chapter 6

Enhancing Performance with Double-Checked Locking

“Double-Checked Locking” (**DCL**) is a software design pattern that reduces the cost of acquiring locks in situations where lock acquisition is only necessary under certain conditions [60]. In the context of our work, we leverage this pattern to improve the efficiency of snapshotting mechanisms by preventing unnecessary acquisition of reader locks when snapshotting is not in progress. As snapshotting occurs infrequently compared to the duration of an epoch (~ 100 ms according to [71]), optimizing lock acquisition can lead to significant performance improvement, which is discussed in Section 7.2.

This chapter presents a deep dive into online and stop-the-world snapshotting mechanisms. We explore how the application of the double-checked locking algorithm can ensure the consistency of backup memory-mapped files, while also mitigating potential inconsistencies between the primary and backup files. We further delve into the potential inconsistencies that can arise under different scenarios and the countermeasures adopted to resolve them using DCL.

6.1 Algorithm

The algorithm utilizes double-checked locking in the following way:

1. A new `boolean` variable, `isSnapshotting`, is introduced to indicate whether snapshotting is currently in progress. This variable is set to `true` when the snapshotting process begins and `false` when it ends (as shown in Algorithm 14).

2. At the start of the `doPersistProtected` function, the value of `isSnapshotting` is checked. If it is:
 - `false`, the algorithm skips acquiring the reader lock and writes back the cache line to the primary memory-mapped file.
 - `true`, the algorithm acquires the reader lock and executes the necessary snapshotting operations (such as updating the modified chunk in online snapshotting).
3. After writing the cache line back, the algorithm rechecks the value of `isSnapshotting`. If `true`, this indicates that snapshotting has begun during this time, and the algorithm must perform the necessary snapshotting operations to ensure that the backup chunk is consistent with the primary chunk. To accomplish this, it attempts to acquire the reader lock, updates the chunk, and then releases the lock.

Despite the performance benefits, the implementation of this double-checked locking approach requires careful consideration to ensure thread safety. Additionally, it is important to note that this algorithm is applicable to both stop-the-world and online snapshotting. Consequently, we have two functions, each specific to a snapshotting algorithm, which can be utilized to ensure consistency once the snapshotting process has commenced.

Algorithm 14: DCL-based Snapshotting Algorithm

```

75 Function takeSnapshotDCL():
76   isSnapshotting  $\leftarrow$  true
77   takeSnapshot()
78   isSnapshotting  $\leftarrow$  false

```

Line 77 suggests that the implementation of the `takeSnapshot` function can be utilized for both online and stop-the-world snapshotting mechanisms. In either case, the first step is to indicate that the snapshotting process has started. This is achieved by setting the `isSnapshotting` variable to `true`. After the snapshot is taken, the variable is unset by setting it back to `false`. This ensures that the system maintains a clear indication of whether the snapshotting process is currently in progress or not.

Algorithm 15: `doPersistProtected` function with double-checked locking improves snapshotting performance by reducing the cost of obtaining reader locks.

```
79 Function doPersistProtected(addr):  
    Input: addr: The memory address of the chunk to persist  
    // Initialize the variable for copying to backup  
80    snapshottingStarted  $\leftarrow$  false  
81    if isSnapshotting == false then  
82        clwb(addr)  
83        markChunk(addr)  
84        if isSnapshotting then  
85            snapshottingStarted  $\leftarrow$  true  
86    else  
87        snapshottingStarted  $\leftarrow$  true  
88    if snapshottingStarted then  
89        doPersistLocked(addr)
```

The `doPersistProtected` function in Algorithm 15 utilizes double-checked locking to enhance snapshotting performance by limiting the cost of obtaining reader locks.

- At the beginning of the function, a boolean variable `snapshottingStarted` is initialized as `false` (line 80). This variable will be used to track whether snapshotting has begun during the execution of this function.
- The function then inspects the status of the `isSnapshotting` boolean (line 81). If `isSnapshotting` is `false`, implying that snapshotting is not currently underway, the function invokes `clwb(addr)` (line 82) to write back the cache line at the address `addr` to the primary memory-mapped file. Subsequently, it marks the chunk as persistently modified by invoking `markChunk(addr)` (line 83).
- After executing these steps, it inspects `isSnapshotting` again (line 84). If it has turned to `true`, this signifies that snapshotting has started in the interim. Hence, `snapshottingStarted` is set to `true` (line 85).
- If the initial check for `isSnapshotting` at line 81 found it to be `true`, the function skips the aforementioned operations and directly sets `snapshottingStarted` to `true` (line 85).

- Finally, if `snapshottingStarted` is `true`, indicating that snapshotting has either begun during the execution of this function or was already in progress at the start, the function invokes `doPersistLocked(addr)` (line 89) to perform the required snapshotting operations under a reader lock. This ensures the backup chunk remains consistent with the primary chunk.

Algorithm 16: `doPersistLocked` function performs the persistence operation using reader locks. Moreover, it updates the backup chunk to maintain the consistency of the memory-mapped files.

```

90 Function doPersistLocked(addr):
    Input: addr: The memory address of the chunk to persist
91   chunkId  $\leftarrow$  getChunkIdx(addr)
92   memMapLock[chunkId].lockShared()
93   clwb(addr)
94   markOrUpdateChunk(addr)
95   memMapLock[chunkId].unlockShared()

```

The `doPersistLocked` function, as shown in Algorithm 16, performs the persistence operation under reader locks, ensuring thread safety during the snapshotting process. It also updates the backup chunk to maintain consistency between the primary and backup memory-mapped files.

- First, the function acquires a shared lock on a chunk to persist the corresponding address (line 92). This lock prevents other threads from writing to the chunk while the snapshotting process is in progress.
- Next, the function call `clwb(addr)` (line 93), writing back the cache line at address `addr` to the primary memory-mapped file.
- Then, the function invokes `markOrUpdateChunk(addr)` (line 94). This step marks the chunk as modified or updates the chunk based on its status at the time of the function call.
- Finally, the function releases the shared lock on the chunk (line 95), allowing other threads to access the chunk.

Please note that in the stop-the-world snapshotting approach there is no requirement to update the modified chunk. We will delve into this scenario in the subsequent section.

6.2 Maintaining Consistency

In both online and stop-the-world snapshotting mechanisms, consistency of the backup memory-mapped files is crucial. By employing the double-checked locking algorithm, we can ensure this consistency while minimizing the frequency of lock acquisitions. This section delves into the methods and considerations of preserving consistency in both snapshotting mechanisms through double-checked locking.

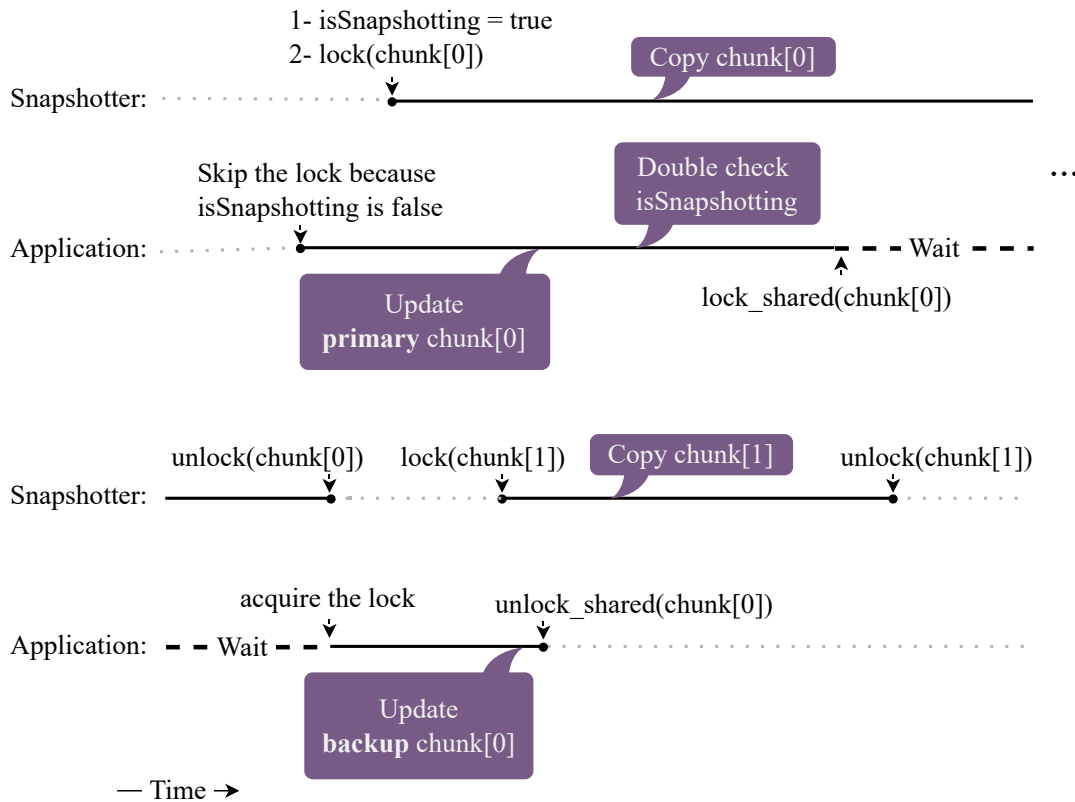


Figure 6.1: DCL in Online Snapshotting: Potential inconsistencies during snapshotting are resolved by updating the altered part at the backup.

6.2.1 Online Snapshotting

In online snapshotting, potential inconsistencies between the primary and backup memory-mapped files can arise under certain scenarios.

Potential Inconsistencies

1. In one scenario, consider the execution of the `doPersistProtected` function, where the `isSnapshotting` variable is read as `false`. This triggers a write operation to the primary memory-mapped file. However, suppose that a write operation occurs immediately after snapshotting starts. The changes made by this write operation might not be captured in the ongoing snapshotting process. As a result, the primary memory-mapped file is updated, but these changes are not reflected in the backup file.
2. In another potential inconsistency, the snapshotting process starts after the application thread checks the `isSnapshotting` variable and sees it as `false`. Subsequently, the snapshotter sets the variable to `true`. During this period, the application thread modifies a specific portion of the primary memory-mapped file. However, before the application thread checks the `isSnapshotting` variable again, the snapshotter sets it back to `false`. As a result, this change made by the application thread will never be reflected in the backup memory-mapped file.

Resolving Inconsistencies with DCL

1. To manage the first potential inconsistency, we check the state of the `isSnapshotting` variable twice — once before executing `clwb` (Line 81) and once after that (Line 84). If `isSnapshotting` has turned `true`, this indicates that snapshotting commenced during the write operation, and it is possible that the ongoing snapshot may not include the changes made by the write operation.

To handle this situation, the DCL mechanism invokes the `doPersistLocked` function (Line 89). This function performs the necessary snapshotting operations under the protection of a reader lock, ensuring that the changes made during the write operation are also reflected in the backup memory-mapped file. Therefore, even if snapshotting starts during a write operation, the DCL mechanism ensures consistency between the primary and backup memory-mapped files throughout the snapshotting process (as shown in Figure 6.1).

2. It should be noted that in an unlikely situation, the snapshotting process may begin and finish while the `doPersistProtected` function is being executed. During the execution of `doPersistProtected`, the epoch advancer is unable to increment the epoch counter. The reason behind this limitation is that when an application thread is persisting a payload using the `doPersistProtected` function, it indicates

that the epoch advancer is waiting for a payload from the previous epochs to be persisted. Thus, it cannot advance the epoch counter, as all payloads from the previous epochs must be persisted before advancing. This constraint ensures that the epoch advancer cannot proceed until the ongoing payload is completely persisted during the snapshotting time.

Consider a queue implemented as a linked-list. Each node addition in the queue is associated with an increment in an epoch counter.

Inconsistency Example

- (a) The queue initially has nodes corresponding to epochs 1 to 5.
- (b) A thread starts persisting a node on epoch 6 by invoking `doPersistProtected`.
- (c) While the node for epoch 6 is being added, a snapshotting process starts and completes before the `doPersistProtected` function finishes.
- (d) The epoch advancer wants to increment the epoch counter but detects the on-going addition for epoch 6, thus waits.
- (e) Once the epoch 6 node is persisted, the epoch advancer increments the epoch counter.

As a consequence, any potential inconsistencies that arise from this scenario are limited to the last two epochs (i.e., epochs 5 and 6). Moreover, these inconsistencies are effectively handled by the recovery routine in the event of a crash failure. The recovery routine discards the payloads generated in the last two epochs and reconstructs a consistent state based on the recovered payloads. This ensures that the recovered state remains reliable and consistent.

6.2.2 Stop-the-World Snapshotting

Stop-the-world snapshotting involves a different concurrency mechanism compared to on-line snapshotting. During the snapshotting process, the snapshotter acquires a writer lock, effectively blocking all other application threads from making concurrent modifications.

Potential Inconsistency

If the snapshotting process starts after an application thread checks the `isSnapshotting` variable and sees it as `false`, the thread will not be able to reflect its changes in the backup

file due to the writer lock acquired by the snapshotter.

Resolving Inconsistencies with DCL

To address the potential inconsistency, we claim that any changes made by the application thread belong to a payload within the last two epochs. As the snapshotter holds the writer lock, the epoch advancer cannot increment the epoch counter until the snapshotting process concludes. Therefore, these changes are encapsulated within the last two epochs. In this scenario, should a crash failure occur, the recovery routine would handle any inconsistencies by discarding the payloads generated in the last two epochs. It then reconstructs a consistent state based on the recovered payloads.

By employing the DCL pattern, exploiting the behaviour of the epoch advancer, and using the exclusive writer lock, both snapshotting mechanisms mitigate potential inconsistencies and ensure the integrity and consistency of snapshots.

6.3 Correctness

6.3.1 Online Snapshotting

Theorem 6.3.1 (*Buffered-Durable Consistency for Online Snapshotting using DCL on top of Montage*). *The online snapshotting mechanism using double-checked locking (DCL) on top of Montage ensures buffered-durable consistency as per Definition 3.4.1.*

Proof. Consider the online snapshotting mechanism using DCL as described in Algorithm 14 and Algorithm 15. The process operates in the following manner:

1. When the DCL-based online snapshotting is initiated, it sets the variable *isSnapshotting* to **true** (Line 76). After the snapshot is taken, it ensures that the variable *isSnapshotting* is set back to **false** (Line 78).
2. If snapshotting starts during a write operation, the DCL mechanism checks if the variable *isSnapshotting* is **false** (Line 81). If *isSnapshotting* is **false**, it proceeds with the standard persistence operation (Line 82) and marks the chunk for persistence (Line 83). However, if snapshotting begins while this process is underway, it sets *snapshottingStarted* to **true** (Line 85).

- If an ongoing write operation fails to detect the start of snapshotting (i.e., it reads *isSnapshotting* as *false* at Line 84) and completes its persistence without updating the backup memory-mapped file, this leads to the same type of inconsistency discussed in case 2. In such cases, we rely on Montage’s recovery routine to address the situation.
3. In the scenario where the snapshotting process has started (either before the persistence operation begins or during its execution), the algorithm ensures that changes are correctly persisted using the `doPersistLocked` method. This is done under a reader lock, as seen in Line 92 and Line 95, ensuring that the changes are correctly reflected in the backup memory-mapped file (Line 94).
 4. Let e be the epoch number when the snapshotter starts copying the descriptor and metadata regions by acquiring all the locks in exclusive mode. The recovery routine manages any inconsistencies that arise during snapshotting by excluding payloads from the two most recent epochs (e and $e - 1$).

During epoch e , when snapshotting begins to copy the descriptor and metadata regions, it prevents the epoch advancer from advancing the epoch counter by acquiring all the locks. Consequently, the snapshot s_{buf} captures all payloads from epochs $e - 2$ and earlier, as proven in Theorem 5.4.1. Given the operations in the DCL algorithm, changes made during snapshotting either occur within the last two epochs or are already reflected in the backup memory-mapped files.

Therefore, utilizing s_{buf} with `MontageRecovery`, we can obtain the state of payloads up to epoch $e - 2$, yielding s_{rec} . From s_{rec} , an abstract state $\mathbf{abstract}(s_{rec})$ is derived. As s_{buf} captures all changes up to epoch $e - 2$, $\mathbf{abstract}(s_{rec})$ is consistent with the abstract state of any snapshot at the conclusion of epoch $e - 2$. Thus, the online snapshotting mechanism using DCL on top of Montage ensures buffered-durable consistency as described in Definition 3.4.1. $\square$

6.3.2 Stop-the-World Snapshotting

Theorem 6.3.2 (*Buffered-Durable Consistency for Stop-the-World Snapshotting using DCL on top of Montage*). *The stop-the-world snapshotting mechanism using DCL on Montage ensures buffered-durable consistency as defined in Definition 3.4.1.*

Proof. Consider the stop-the-world snapshotting mechanism using DCL as detailed in Algorithm 14 and Algorithm 15.

1. At the beginning of the snapshotting process, a writer lock is acquired (Line 7). This action ensures that other threads are blocked from making concurrent modifications while the snapshotting is in progress.
2. If a thread evaluates the *isSnapshotting* flag as **false** prior to the start of the snapshotting process (Line 81), then its modifications will not be reflected in the backup file. This is due to the mechanism in Line 92, where the application thread is made to wait for the completion of snapshotting before being able to acquire the lock in shared mode.
3. Moreover, if the **takeSnapshot** function begins its execution during epoch e , it will capture all payloads from epochs as early as $e - 2$, as established by Theorem 4.4. It is crucial to understand that any modifications made by threads during this waiting period (which might not appear in the snapshot) are limited to the payloads of the most recent two epochs, as detailed in Section 6.2.2.

If the snapshotting starts and finishes during epoch e , then the snapshot s_{buf} includes all payloads up to and including epoch $e - 2$. By using s_{buf} as the input to the **MontageRecovery** routine, we obtain s_{rec} , which represents the state of the payloads up to epoch $e - 2$. From s_{rec} , the abstract state $\mathbf{abstract}(s_{\text{rec}})$ can be determined. Consequently, $\mathbf{abstract}(s_{\text{rec}})$ matches the abstract state of any snapshot taken at the end of epoch $e - 2$. From this depiction, it is clear that the stop-the-world snapshotting mechanism using DCL on top of Montage assures buffered-durable consistency as outlined in Definition 3.4.1. $\square$

This chapter provided a comprehensive examination of how DCL can be employed to manage potential inconsistencies between primary and backup memory-mapped files in on-line and stop-the-world snapshotting mechanisms. We also established the correctness of the methods based on the consistency model proposed in Definition 3.4.1. The exploration underscores the importance and effectiveness of DCL in preserving the integrity of snapshots, confirming its utility in snapshotting mechanisms, and providing reliable, consistent data recovery.

Chapter 7

Experimental Evaluation

In this chapter, we focus on the systematic experimental evaluation of our systems, assessing their performance under various conditions. Our main testing environment comprised a server equipped with four Intel(R) Xeon(R) Gold 6230 processors running Ubuntu 20.04.6 LTS. Each processor has 20 physical cores and 40 hyperthreads, summing up to a total of 160 hyperthreads. For thread pinning, for up to 20 threads, we utilized only one socket without hyperthreading. Between 20 and 40 threads, we continued to use a single socket but enabled hyperthreading. For thread counts ranging from 40 to 80, we activated the second socket. When we needed to handle more than 80 threads, up to 90, we employed oversubscription on the first socket. The number 90 was chosen as the cap to maintain consistency with the original study, enabling direct comparisons with [72].

We conducted the performance tests of two different data structures, a queue and a hash map. We used Montage, which is available on [73], and is based on the original paper [72]. For these data structures, we used the same benchmark that was used in [72]. Specifically, for the hashmap, a linked list for each bucket with a lock assigned was used to ensure thread safety; for the queue, only one lock was used. We preloaded 0.5 million elements into 1 million hash buckets for the hash map and 2000 elements of 1 KB each for the queue. The results were averaged over five trials for each data point, with a standard deviation of less than 2% of the mean. Our detailed evaluation encompasses the variation of chunk size, throughput analysis, assessment of snapshotting time, and exploration of the influence of epoch length on performance. Additionally, we compared Montage’s performance with the other possible implementation techniques, as detailed in Appendix A. The code is open-sourced under an institutional Git repository at the following link: <https://git.uwaterloo.ca/mmoridi/montage-uw>. This repository is forked from the open-source version of Montage available on <https://github.com/urcs-sync/Montage>.

For the hashmap, we considered three different workloads:

- Write-dominant, with 0% get, 50% insert, and 50% remove (labelled as g0i50r50)
- Read-dominant, with 90% get, 5% insert, and 5% remove (labelled as g90i5r5)
- Balanced, with 50% get, 25% insert, and 25% remove (labelled as g50i25r25)

For the queue, we considered a specific workload, comprising 50% enqueue and 50% dequeue (labelled as 50enq50deq).

7.1 Finding the Optimal Chunk Size

The experiments aim to identify the optimal chunk size for snapshotting algorithms, considering system throughput and snapshotting duration.

Parameters Explored

- **Chunk Size:** Different chunk sizes were considered to understand their impact on system performance.
- **Thread Count:** Various numbers of threads were used to investigate the influence on performance.
- **Snapshotting Technique:** A multi-threaded stop-the-world snapshotting mechanism without double-checked locking was employed.
- **Backup Medium:** DELL 1TB 7.2K RPM NLSAS 12Gbps 512n 2.5in Hot-Plug Hard Drive.

Methodology

1. **Throughput Evaluation:** The system throughput was assessed with snapshotting enabled. During the initialization phase, all the chunks are copied to the provided backup address. After that, during execution, the system takes a snapshot three times every five seconds.

2. **Metrics:** The following metrics were measured for each experiment:

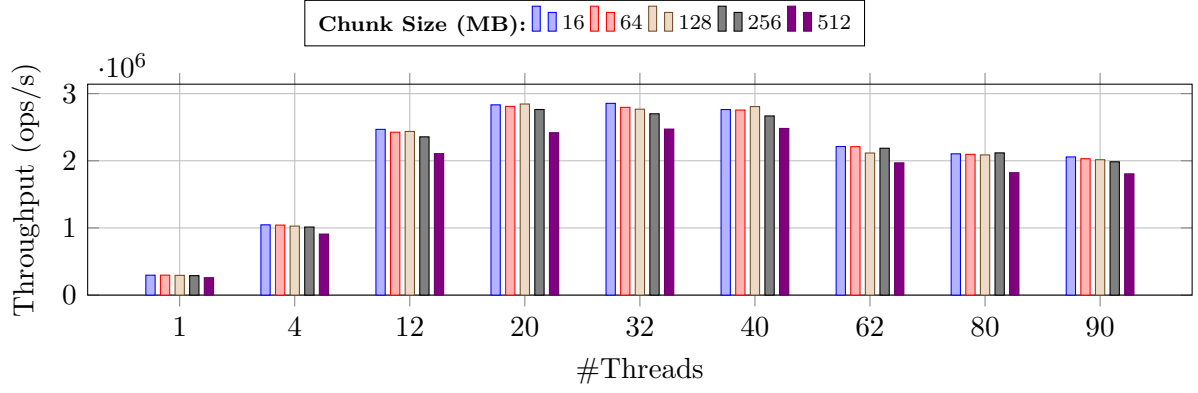
- System throughput: It refers to the number of data structure operations per second, such as enqueue and dequeue operations for a queue data structure.
- Average snapshotting duration
- Percentage of altered chunks for each snapshot. The lower the percentage, the less data needs to be copied.

Figure 7.1a’s findings from the write-dominant load on the hash map reveal a trend of declining performance as chunk size increases. Smaller chunk sizes result in more chunks being created, which improves performance. Figure 7.1c demonstrates that decreasing the chunk size results in a lower percentage of chunks having changed since the last snapshot. Due to the reduced amount of data that needs to be copied, this improves performance and decreases snapshotting time (figure 7.1b).

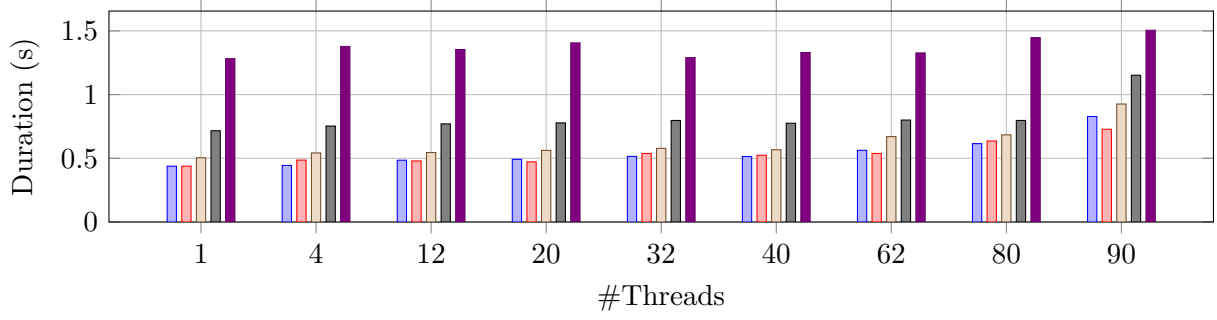
However, there is a small performance loss when increasing from 20 to 40 threads, which hyper-threading may be responsible for. However, when the number of threads is increased from 40 to 80, there is a noticeable decline in performance since having two different NUMA nodes while executing a write-dominant load causes remote cache misses because the other NUMA node’s cache lines are invalidated. Going from 80 to 90 threads further worsens performance due to oversubscription on NUMA node 1.

In Figure 7.3a, a trend is evident: as the chunk size increases, the system’s performance decreases, irrespective of the number of threads. Additionally, Figure 7.3c reveals that by increasing the chunk size, the average percentage of changed chunks also increases. This implies that the snapshotter needs to copy a larger amount of data with each snapshot. Consequently, this leads to an increased copying time, which extends the duration of snapshotting, as illustrated in Figure 7.3b. Furthermore, when analyzing the impact of varying thread counts while keeping the chunk size consistent, it is apparent that a greater number of threads correlates to reduced efficiency. This can be attributed to the queue’s single-lock mechanism, which becomes a bottleneck when multiple threads are in operation.

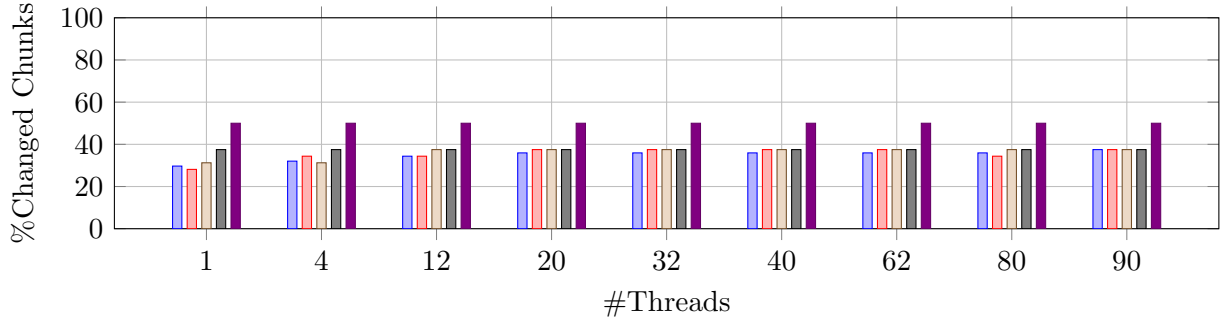
Figure 7.2a shows a surprising fact that the 64 MB chunk size program outperforms the 16 MB chunk size, and also leads to shorter snapshotting time, as demonstrated in Figure 7.2b. Additionally, Figure 7.2a shows that greater performance appears from raising the number of threads for a given chunk size. This improved performance is caused by fewer coordination efforts and updates across NUMA nodes.



(a) Throughput Analysis

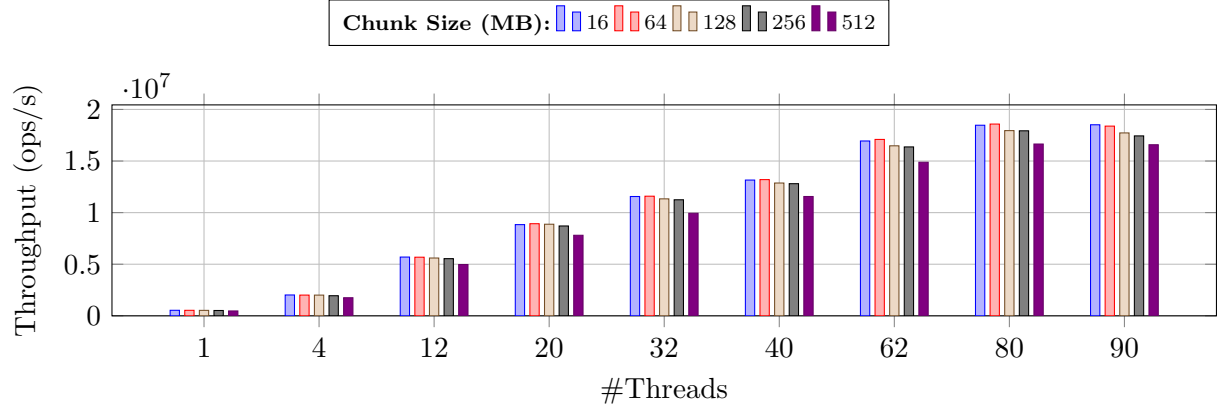


(b) Snapshotting Time

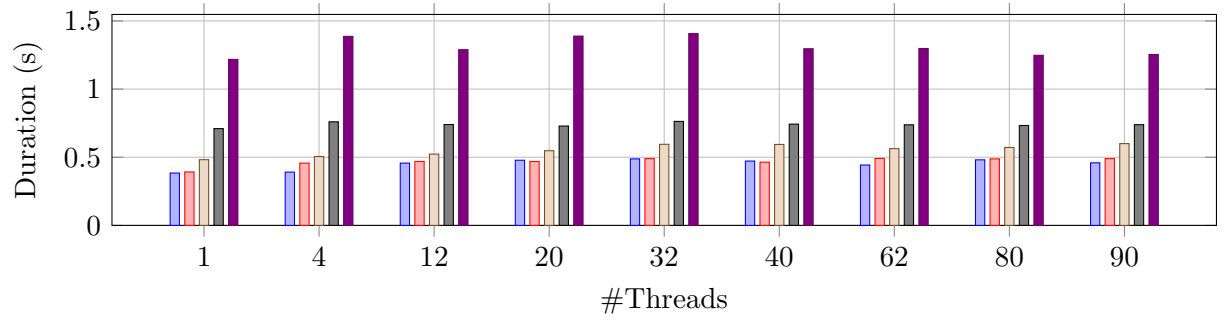


(c) Average percentage of changed chunks

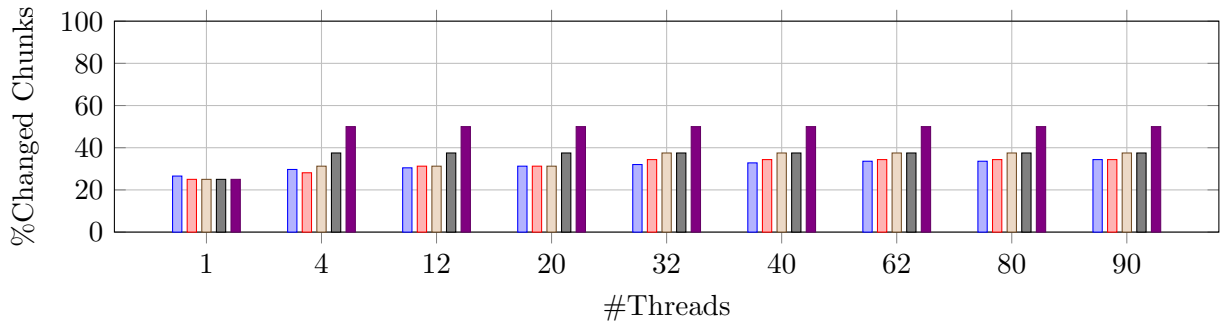
Figure 7.1: Data structure: Hash map (g0i50r50). Algorithm: Multi-threaded stop-the-world snapshotting without double-checked locking.



(a) Throughput Analysis



(b) Snapshotting Time



(c) Average percentage of changed chunks

Figure 7.2: Data structure: Hash map (g90i5r5). Algorithm: Multi-threaded stop-the-world snapshotting without double-checked locking.

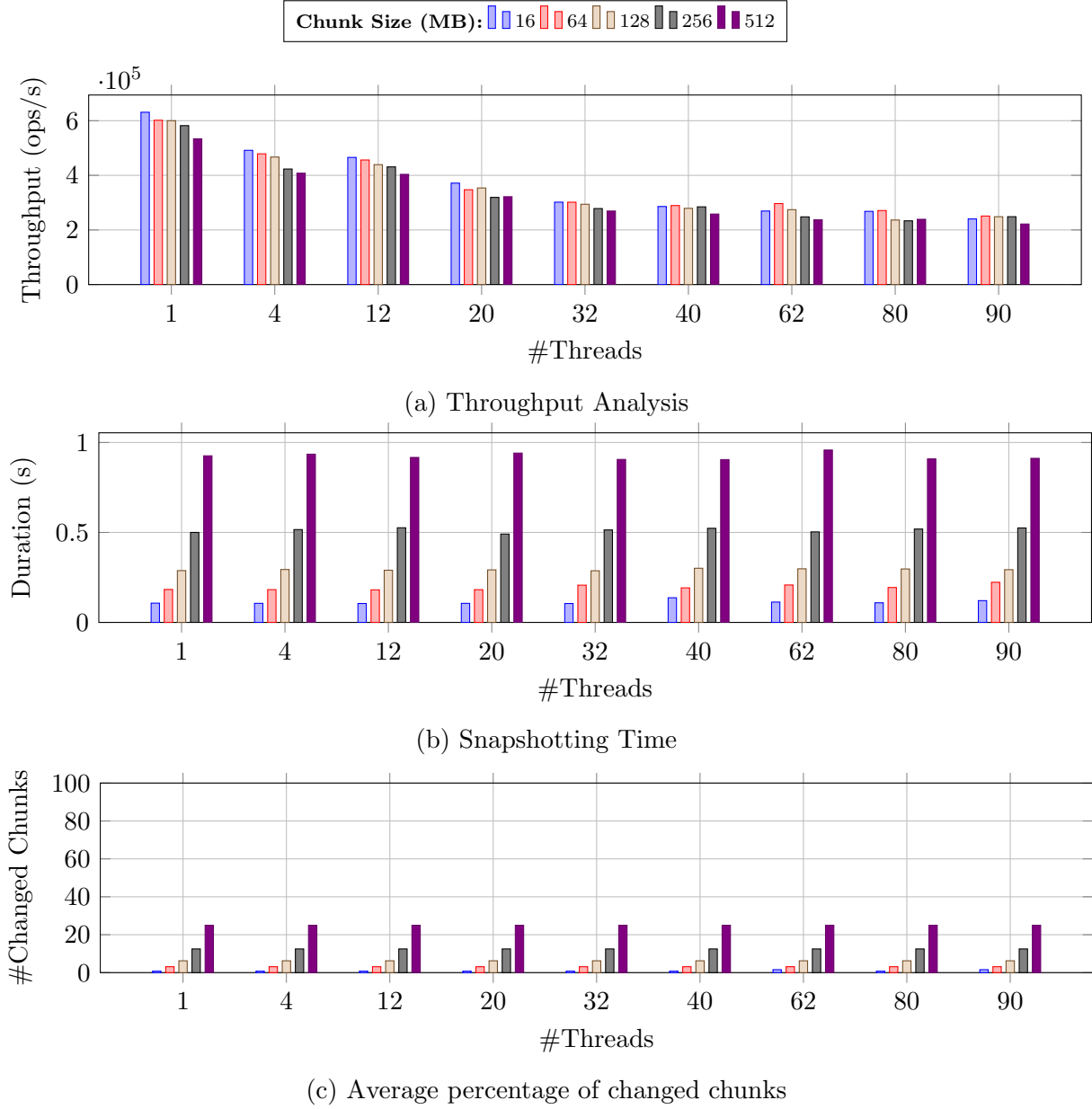


Figure 7.3: Data structure: Queue (enq50deq50). Algorithm: stop-the-world snapshotting without double-checked locking.

7.2 Throughput Analysis

The diagrams in this section provide a comparison of the performance of various snapshotting algorithms. The duration required for snapshotting includes the complete cycle of the process, which runs every five seconds three times, using a chunk size of 16 MB.

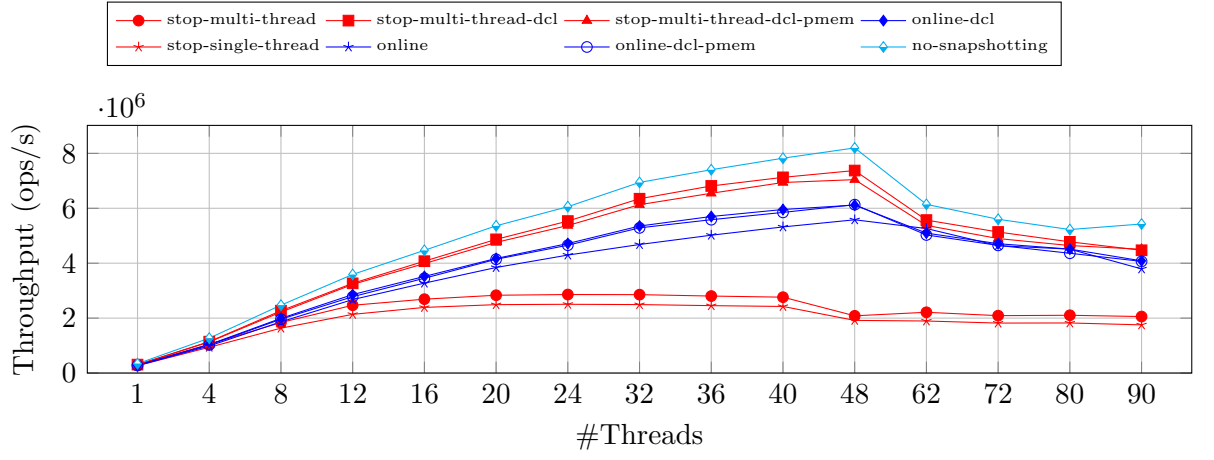
The algorithms represented in the diagram include both stop-the-world (starting with **stop** in the figure) and online snapshotting (starting with **online**) techniques, either equipped with double-checked locking (indicated by **dcl** in the name) or without it. Additionally, stop-the-world snapshotting is classified into two types - single-threaded (indicated by **single-thread**) and multi-threaded (indicated by **multi-thread**) snapshotting.

Given that snapshots are being created, the selection of storage media becomes important. Our project anticipates potential failures in one of the persistent memory modules that could be beyond repair. However, we can use another persistent memory module or an HDD to store our snapshots, as evidenced in the previous section's trials. The diagrams also display the performance of double-checked locking versions that use another PMEM module for storing backup files (ending with **pmem**).

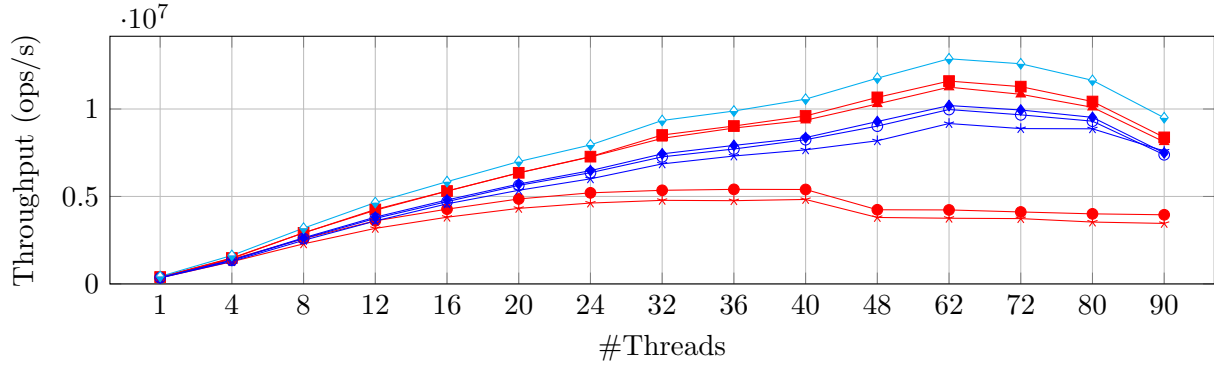
According to Figure 7.4a, the throughput of the **no-snapshotting** (Montage without any memory-mapped snapshotting) and the **stop-multi-thread-dcl** method shows that the gap lessens as the number of threads increases; at 80 threads, the difference is less than 10%. Furthermore, a clear trend across all three charts in Figure 7.4 shows that multi-threaded variants of stop-the-world snapshotting with DCL enabled outperform all other versions. It should be noted that, despite online snapshotting lagging behind the multi-threaded version of stop-the-world snapshotting, it still maintains responsiveness during snapshotting (explored in Section 7.2.1), something the stop-the-world variant cannot provide. In addition, online snapshotting without DCL performs better than the stop-the-world versions without DCL.

The effectiveness of snapshotting methods with double-checked locking suggests possible advantages in minimizing lock contention. The use of persistent memory as the backup medium in both online and stop-the-world snapshotting methods results in marginally inferior performance. A more thorough examination would be required to fully comprehend the performance variations between HDD-based and persistent memory-based techniques.

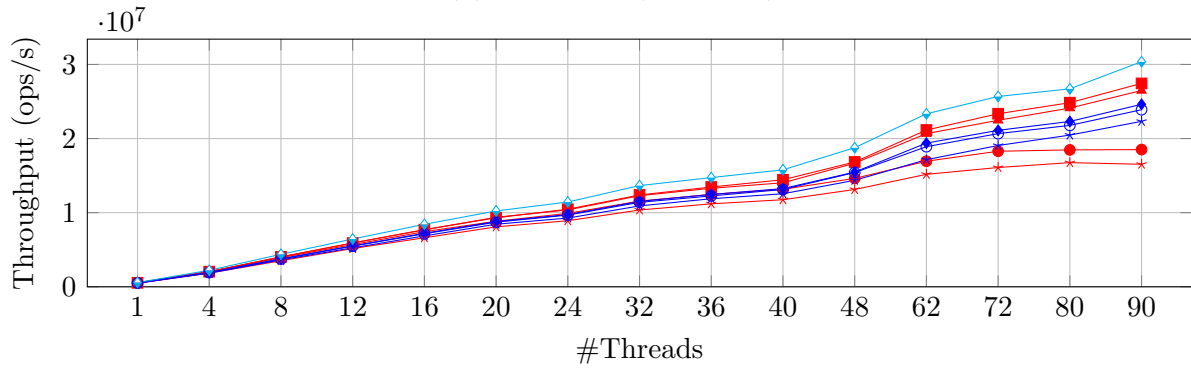
As per Figure 7.5 related to the queue (with 2000 elements) data structure, which typically modifies only a single chunk, there is not a significant difference in performance between the various versions of snapshotting mechanisms and the **no-snapshotting**. This indicates a scalability issue with the queue data structure. Primarily, increasing the number



(a) Hash map (g0i50r50)



(b) Hash map (g50i25r25)



(c) Hash map (g90i5r5)

Figure 7.4: Average throughput for hash map

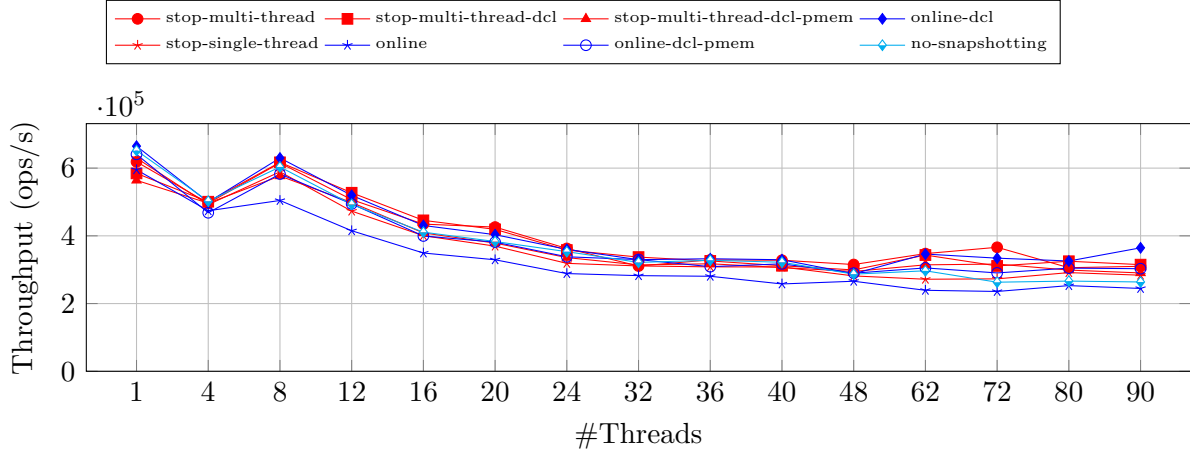


Figure 7.5: Average throughput for queue (enq50deq50)

of threads does not enhance performance, and the threads routinely wait for one another. Even with the snapshotting enabled, the performance remains the same.

7.2.1 Responsiveness

In this section, we aim to evaluate the efficacy of our online snapshotting mechanism. To achieve this, we executed our algorithm on a hashmap data structure across three distinct workloads. We define a *dual write* as an operation where modified memory locations must be written back to both the primary and backup persistent memory-mapped files during the snapshotting process. This operation occurs when an application thread tries to modify a chunk that has already been copied.

Our system addresses these dual writes by updating the backup file and promptly responding to client requests (refer to Section 5.2). Consequently, our program’s responsiveness improves since it does not delay client requests, unlike the stop-the-world snapshotting. In the case of stop-the-world snapshotting, no dual writes are handled as the snapshotting process pauses the program, effectively suspending any incoming requests for its duration. Therefore, for algorithms employing this method, the throughput for write operations on copied chunks during snapshotting is represented as horizontal lines at the value zero in Figure 7.6. This indicates the method’s inability to process client requests during snapshotting. In simpler terms, while our online snapshotting continues to address client demands, stop-the-world snapshotting suspends all such requests.

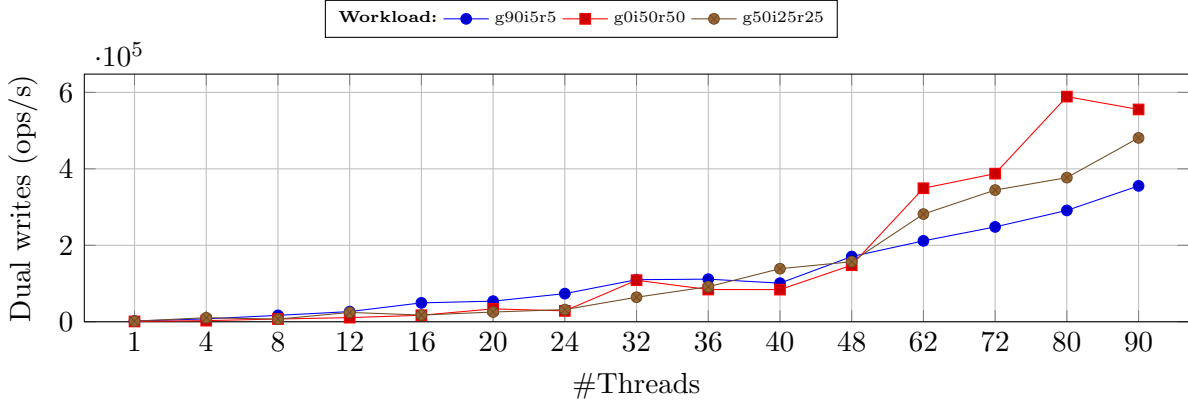


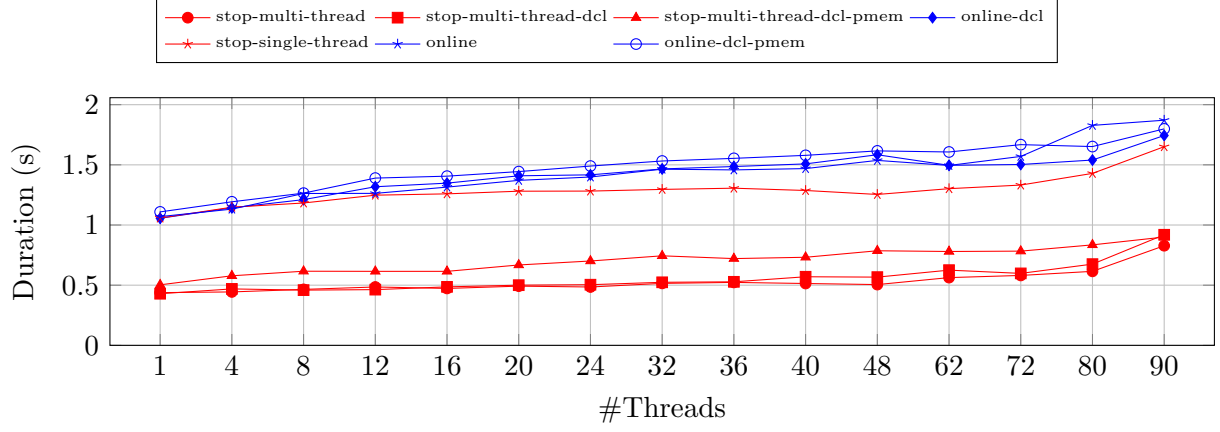
Figure 7.6: Average number of conflicts of a hashmap during online snapshotting

Referencing Figure 7.6, we observe a clear correlation: as the number of threads increases, our algorithm efficiently manages more dual writes. This demonstrates our program’s enhanced responsiveness to client requests, especially as the thread count rises. It is noteworthy to mention that the write-dominant workload incurs a higher number of dual writes. However, this does not necessarily imply superior performance compared to the read-dominant workload, as the majority of operations in the latter do not modify memory-mapped files and thus bypass our snapshotting mechanism, accessing the memory-mapped files directly.

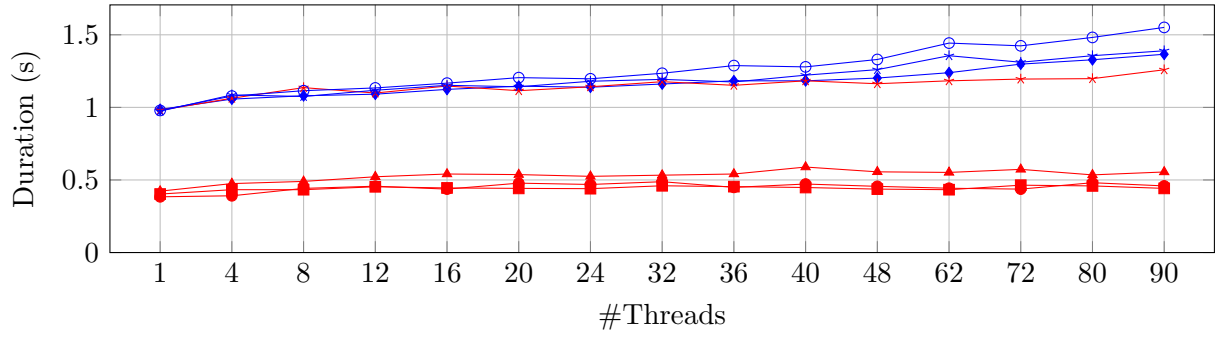
7.3 Snapshotting Time

In this section, we will concentrate on evaluating the total snapshotting duration for the aforementioned snapshotting variants.

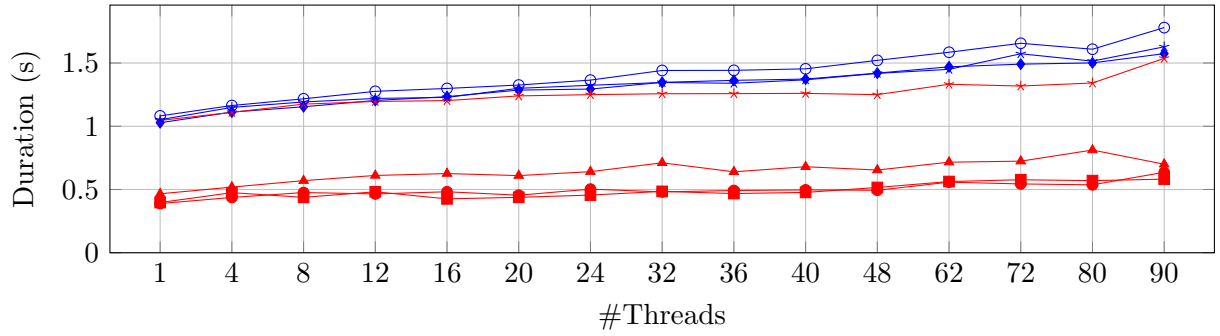
As depicted in Figure 7.7, the multi-threaded variants of the stop-the-world snapshotting method consistently show the shortest snapshotting durations across all thread counts, aligning with the high performance of the DCL versions in the throughput analysis. This can be credited to the lower contention for the reader locks facilitated by double-checked locking. Also, it is important to note that, while multi-threaded stop-the-world snapshotting without DCL demonstrates lower snapshotting times than all the other variants, it has difficulty obtaining the reader locks when snapshotting is not ongoing. Furthermore, online snapshotting methods show longer snapshotting times in comparison to all the stop-the-world methods, which is anticipated, as online snapshotting methods permit



(a) Hash map (g0i50r50)



(b) Hash map (g90i5r5)



(c) Hash map (g50i25r25)

Figure 7.7: Average snapshotting time for hash map

the system to proceed with other tasks during the snapshotting process, which may extend the snapshotting duration.

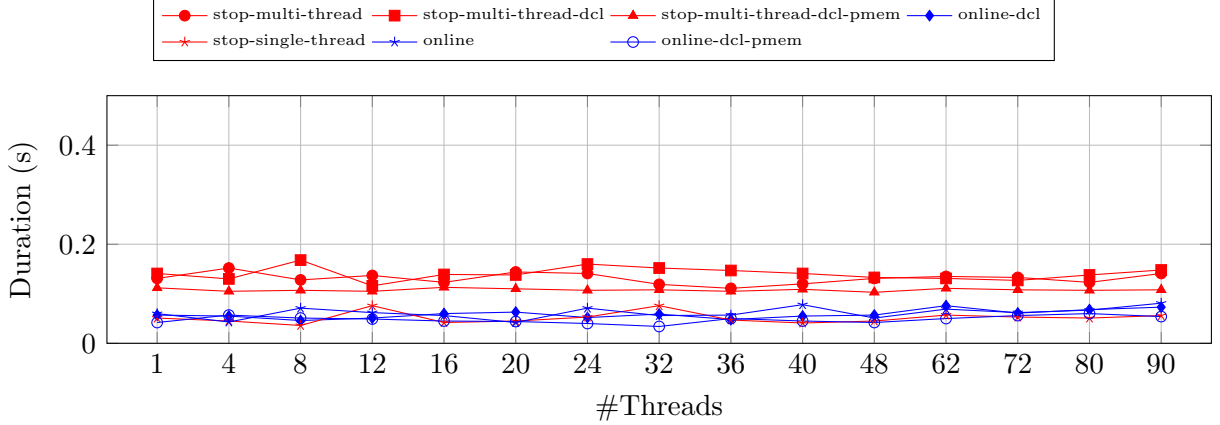


Figure 7.8: Average snapshotting for queue (50enq50deq)

Based on Figure 7.8, the snapshotting times for the stop-the-world methods are slightly higher than those of the online method across all thread counts, although the difference is minimal. We have observed that in the case of a queue, the number of altered chunks is typically one. Consequently, multi-threaded variations do not benefit from parallelism as only one chunk needs to be duplicated.

7.4 Impact of Write-Backs

In this section, we aim to explore the effect of write-backs on the performance of the Montage. Buffered durable linearizability mirrors the behaviour of conventional file systems where operations are allowed to return before achieving persistence. If an operation needs to be persisted before returning to the client, it uses the “*Direct Write-Back*” (DirWB) method. DirWB conducts a write-back and flushes all written data immediately after each update. In the charts, a bar represents the strictly durable implementation of Montage (Strict). In this variation, the `sync` function is invoked after each operation. When `sync` is triggered on epoch e , it waits for all active operations on epochs $e - 1$ and e , and then increases the epoch counter by two. As a result, if an operation completes, the epoch jumps by two. This ensures visibility even if a crash failure occurs and the recovery routine omits payloads from the last two epochs.

Explanation of Figure Legends

- **DirWB(#)**: Represents the “*Direct Write-Back*” approach. The number ‘#’ indicates the epoch length in milliseconds.
- **Buffered(100)**: Montage’s buffered method with a fixed epoch length of 100ms.
- **Strict**: The strictly durable implementation of Montage, invoking ‘`sync`’ after every operation.

In order to investigate the impact of epoch length in systems where each operation completes a write-back before returning, we conducted an experiment with Montage. We adjusted the epoch length and compared it to a buffered version of Montage that has an epoch length of 100 ms. This value is the optimal one based on our experiments, also reported in [72]. As the figures illustrate, the buffered version of Montage outperforms the Direct Write-Back (DirWB) versions, especially when a higher number of threads are used, across write-dominant and balanced workloads.

Interestingly, the influence of epoch length on DirWB performance varies depending on the workload and number of threads. For instance, Figure 7.9a reveals that with fewer than 40 threads - limited to one socket - performance increases with longer epochs, suggesting that a higher number of the payload within the same epoch boost the system performance. However, when the thread count rises from 40 to 90 - employing another NUMA node - shorter epochs yield better Montage performance. This shift implies that as more payloads fall within the same epoch, an increase in remote cache misses between the two nodes leads to lower performance than with longer epochs. The pattern of shorter epochs enhancing performance also holds true for balanced workloads.

In a read-dominant environment, shorter epochs with the DirWB implementation indicate that write-backs are committed more frequently, ensuring that the latest data updates are reflected in persistent memory. This promotes data consistency across caches and ensures that the latest values are available to all threads, enhancing cache hits and reducing data retrieval times. On the other hand, the buffered version introduces a delay in write-backs due to its buffering mechanism. While this might optimize write operations, it could cause a slight delay in distributing the most recent values among threads. As a result, in scenarios dominated by read operations, the DirWB with shorter epochs can potentially achieve higher throughput compared to its buffered counterpart.

As depicted in Figure 7.10, for the queue data structure, the buffered version consistently outperforms the DirWB variants. The advantage of the buffered version is its

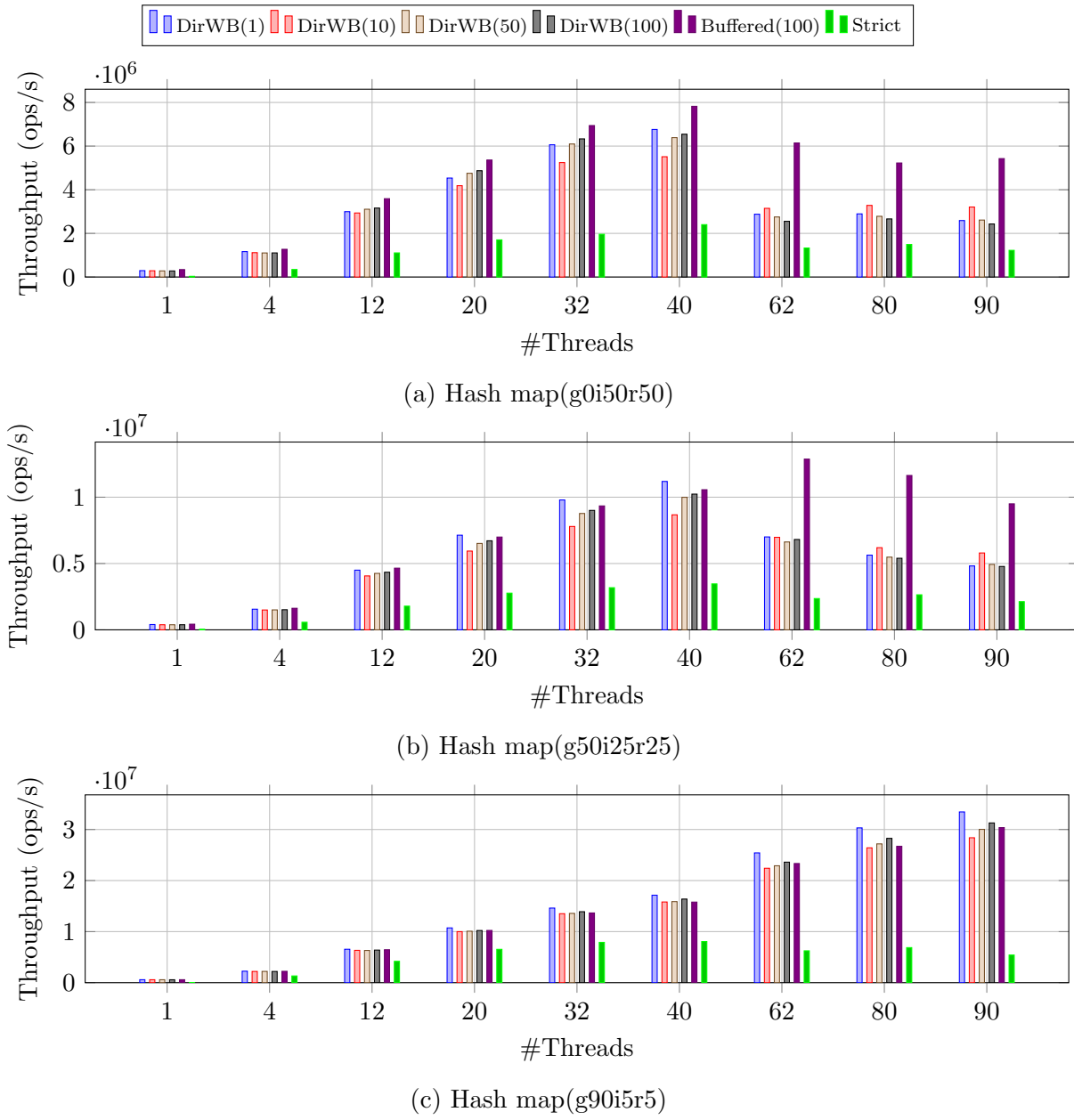


Figure 7.9: Impacts of cache-line write-backs on hashmaps

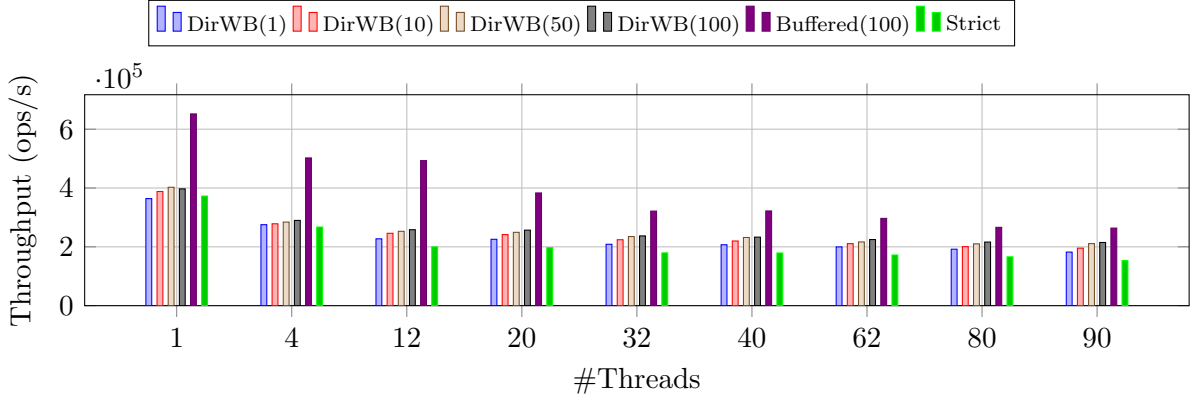


Figure 7.10: Impacts of cache-line write-backs on queue(enq50deq50)

capacity to aggregate several changes, notably at high contention points such as the head and tail pointers, into one write-back operation. This aggregation enhances I/O efficiency to persistent memory, resulting in improved performance.

As depicted in Figures 7.9 and 7.10, the strictly durable implementation lags significantly in performance compared to other variations—yielding only about one-fifth to one-third of the throughput of the buffered variations. This stark difference underscores the inefficiency of the `sync` function. This performance gap is attributed to the high contention on the epoch counter: with numerous threads trying to increment this global counter, they end up in a wait state, leading to a bottleneck. Even at lower thread counts, where contention on the epoch counter is not a factor anymore, this implementation shows noticeably reduced performance compared to the buffered version. The reason lies in the `clwb` implementation on the 2nd generation Xeon(R) processors we used. Specifically, it not only writes the cache lines back to persistent memory but also flushes them. Consequently, every invocation of `sync` invalidates the cache lines that are explicitly written back. This flushing mechanism directly impacts the performance, leading to diminished throughput.

Overall, the section demonstrates that the buffered version of Montage consistently outperforms the Direct Write-Back (DirWB) method across diverse workloads and thread configurations. While DirWB provides immediate persistence, its performance is variable, influenced by factors such as epoch length, thread count, and workload type. Conversely, the strictly durable implementation is plagued by marked inefficiencies, particularly in high contention scenarios, underscoring the superiority of the buffered approach in the contexts examined.

7.5 Take-Aways

Throughout this chapter, a detailed examination of snapshotting algorithms was conducted, in different experimental conditions.

1. **Multi-threaded Snapshotting Performance:** The multi-threaded variants of stop-the-world snapshotting with double-checked locking were shown to be robust and efficient compared to other methods.
2. **HDD vs. Persistent Memory:** There remains a need for further research to understand the performance differences between HDD-based and persistent memory-based snapshotting techniques.
3. **Advantages of Online Snapshotting:** While online snapshotting might exhibit reduced performance in certain scenarios, it provides the advantage of maintaining system responsiveness during the snapshotting process, a feature not available in the stop-the-world method.
4. **Impacts of Write-Back and Strict Durable Linearizability:** The buffered version of Montage consistently outperformed the Direct Write-Back method across various workloads and thread configurations. The strictly durable implementation, however, exhibited marked inefficiencies, especially under high contention scenarios.

In conclusion, these findings emphasize the importance of choosing the correct snapshotting techniques based on the specific requirements of both the data structure and the workload characteristics.

Chapter 8

Related Work

In recent times, research on persistent data structures has witnessed substantial growth. This encompasses studies on varied components such as B-tree indices in databases and file systems [29, 33, 43, 52], RB trees [69], skip lists [39, 15], hash maps [55, 62, 77], and queues [21, 38, 50].

Durable linearizability has emerged as the key correctness criterion for persistent data structures [21, 30, 48, 77], building on the well-established concept of linearizability [26] for concurrent structures. Izraelevitz et al. [30] offer a mechanical construction for converting any nonblocking concurrent structure into a correct persistent version, while David et al. [19] describe techniques for eliminating redundant write-backs and fences, improving performance significantly. In addition to custom data structures, various groups have developed systems to ensure failure atomicity in lock-based critical sections [13, 27, 31, 45]. These systems adhere to durable linearizability by ensuring operations persist before returning to the calling thread.

The Dalí hash map [55] postpones persistence to reduce overhead; however, it depends on a flush-all-cache instruction, which is only available in privileged mode on x86 and can lead to unnecessary evictions. Medley [10], another work related to Montage, implements nonblocking transaction composition (NBTC) to provide a high-concurrency alternative for transactional data structures. When combined with **nbMontage** to create **txMontage**, it delivers ACID transactions and high throughput for periodically persistent data structures, aligning with Montage’s objectives. **nbMontage** introduces a completely lock-free version of the Montage system for persistent data structures. They manage to significantly outpace earlier general-purpose systems and often achieve at least half the throughput of similar non-persistent structures stored in DRAM [9].

8.1 Snapshotting

To the best of our knowledge, one of the most commendable efforts to facilitate snapshotting in a persistent memory system is embodied by the Pronto system. The Pronto system, created by Memaripour et al. [48], uses a volatile memory allocator and asynchronous semantic logging (ASL) to create snapshots and support various linearizable concurrent data structures. However, it cannot guarantee complete protection against persistent memory hardware failures, as these snapshots are stored on PMEM without backup on another PMEM module or HDD. Additionally, there is a substantial body of research on in-memory databases and consistency models for snapshots that will be explored in the following subsections.

8.1.1 Consistency Models

Prior to asserting the correctness of a snapshot, it is imperative to establish a formal definition for the concept of correctness. Considerable research, including the works by Mattern and Friedemann [47] and Schwarz and Mattern [63], are focused on asynchronous distributed systems where the principal sense of time is causality [35], making it often difficult to identify the system’s state at any given time [16]. Ottmann et al. [56] build on top of the work of Vömel and Freiling [68]. They present two new definitions of atomicity, referred to as “*instantaneous*” and “*quasi-instantaneous*” consistency. In Chapter 3, we present that none of these definitions is a sufficient consistency model for a buffered-durable model.

8.1.2 Checkpointing

ThyNVM, presented by Ren et al. [59], offers a solution to the challenges posed by DRAM + NVM hybrid memory architectures, aiming to guarantee crash consistency without necessitating software alterations. To achieve this, ThyNVM employs a dual-scheme checkpointing method that periodically captures the state of the main memory. It adjusts the granularity of these snapshots based on the spatial locality of memory updates. Meanwhile, the dual-page checkpointing system [74] presents an approach focused on data persistence by using a dual-page mapping scheme with two physical pages, ensuring efficient cache-line level mapping to maintain checkpoint consistency. However, despite its efficiency, the method can result in additional write operations to the NVM. While ThyNVM’s appeal lies in its software transparency, more comprehensive systems like Whole-System Persistence

[53] also consider device states, providing a broader scope of crash consistency. However, ThyNVM’s strength remains its potential for straightforward real-world implementation and adoption.

8.1.3 In-Memory Databases

There are various snapshot algorithms used in in-memory database systems. These algorithms are crucial in maintaining data consistency and allowing for effective data recovery. In the following, you can find a brief overview of these algorithms based on [37]. In this context, a “*client thread*” is a thread that is performing operations on behalf of a client. A “*snapshotter thread*”, on the other hand, would be a thread that is taking snapshots of the data in the database.

- **Naive Snapshot (NS)** [7, 61]: This technique forms a copy of the entire dataset, causing the client thread to pause during this process. Once the in-memory snapshot is created, the client thread is reactivated. The snapshotter thread can asynchronously browse or scan the snapshot data. Even though clients can read the most recent data throughout this procedure, it can be slow and consume a lot of resources, especially for larger datasets. This is due to the fact that client threads are paused while the snapshot is being created. This is reminiscent of our stop-the-world technique but without using the chunking mechanism.
- **Copy-on-Update (COU)** [11]: This method utilizes an extra data structure to create a duplicate of the primary dataset and a bit array to log the row update statuses. When a client makes changes to a row for the first time, a duplicate row is made and the respective bit is set to represent the pre-update state of the row. While this approach copies only the changed data, which is more efficient than the NS method, it still necessitates distinct locks between the client thread and the snapshotting thread for each row, which could potentially diminish performance.
- **Zigzag (ZZ)** [12]: This approach uses two bit arrays to indicate which copy the client should modify and read. In contrast to the COU method, the ZZ method does not need a mutex lock, which is suitable for smaller datasets. However, for larger datasets, the bit arrays can occupy a substantial amount of memory, which can be problematic.
- **Ping-Pong (PP)** [12]: This technique involves two versions of the data; one is used for updates and the other for noting the progressive snapshot. This method

eliminates latency spikes completely as the client thread writes to both copies, and the snapshotter thread can access the progressive snapshot asynchronously. However, each write operation has to modify both the primary data and the updated copy, which could result in performance degradation.

- **Hourglass (HG)** [37]: This is an advanced version of the previous snapshot techniques. It merges the Zigzag bit array marking and the Ping-Pong pointer swapping to prevent latency spikes while keeping a small memory footprint.
- **Piggyback (PB)** [37]: This is an additional advancement that enables full snapshots with pointer swapping by incorporating a background thread that copies fresh data from one dataset to another. It uses a two-bit array to show from which dataset the client thread should read.

The techniques described for in-memory databases are optimized for transient data and often introduce fine-grained operations. Montage requires snapshot methods that guarantee both speed and durability across non-volatile storage, challenges not addressed by the aforementioned in-memory database techniques. Additionally, we adopt an approach similar to the Ping Pong method during online snapshotting to ensure snapshot consistency by the end of the snapshotting process.

Chapter 9

Conclusion and Future Work

In conclusion, our work marks significant strides in enhancing the fault tolerance of Montage. We achieved this by establishing a robust definition for snapshot correctness, termed “*Buffered-Durable Consistency*”. Additionally, we implemented both *stop-the-world* and *online* snapshotting mechanisms. The performance of these two mechanisms was further enhanced by utilizing *double-checked locking* to minimize reader lock acquisitions, leading to an improvement in overall system performance. These strategies are designed to ensure data consistency, durability, and efficient recovery, providing a solid foundation upon which persistent memory applications can be built and operated.

Nonetheless, each chapter of our exploration revealed potential trade-offs and areas for further refinement and research. For instance, the current state of our online snapshotting mechanism limits its applicability to large allocations. Moreover, our understanding of the performance differences between HDD-based and PMEM-based backup media is still nascent. These areas of growth do not undermine our achievements but serve to illuminate the path for future exploration. Moreover, several promising opportunities for optimization and enhancement have been identified. For example, our snapshotting algorithms are not currently NUMA-aware, but tailoring them to accommodate this could result in significant performance improvements. Also, we observed differences in performance between the two snapshotting mechanisms: *stop-the-world* exhibited higher overall throughput compared to *online*. Further research could explore and quantify these differences, including latency measurements, to provide a more comprehensive understanding of their respective strengths and weaknesses. Additionally, the creation of a verifier for buffered durably linearizable platforms could allow for more efficient program correctness testing.

By focusing on these areas, we strive to continue expanding the robustness, fault tol-

erance, and overall performance of our snapshotting mechanisms. This will contribute significantly to the development and management of persistent memory applications.

References

- [1] Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, 2003.
- [2] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent memory: A survey of programming support and implementations. *ACM Computing Surveys (CSUR)*, 54(7):152, 2022.
- [3] Ohad Ben-Baruch and Srivatsan Ravi. Separation and equivalence results for the crash-stop and crash-recovery shared memory models. *arXiv preprint arXiv:2012.03692*, 2020.
- [4] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, page 253–264, 2019.
- [5] R. Berryhill, W. Golab, and M. Tripunitara. Robust shared objects for non-volatile main memory. In *International Conference on Principles of Distributed Systems*, Rennes, France, 2015.
- [6] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. Makalu: fast recoverable allocation of non-volatile memory. In *Proc. of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, page 677–694, 2016.
- [7] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent advances in checkpoint/recovery systems. In *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, pages 282–289, 2006.
- [8] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. Understanding and optimizing persistent memory allocation.

In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, page 1–14, 2020.

- [9] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. Fast nonblocking persistence for concurrent data structures. In *Fast Nonblocking Persistence for Concurrent Data Structures*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [10] Wentao Cai, Haosen Wen, and Michael L. Scott. Transactional composition of non-blocking data structures, 2023.
- [11] T. Cao. *Fault Tolerance for Main-Memory Applications in the Cloud*. PhD thesis, Cornell Univ., Ithaca, NY, USA, 2013.
- [12] T. Cao, M.A.V. Salles, B. Sowell, Y. Yue, A.J. Demers, J. Gehrke, and W.M. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. ACM SIGMOD Int. Conf. Manage. Data*, pages 265–276, 2011.
- [13] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, page 433–452, 2014.
- [14] S. Chen and Q. Jin. Persistent B $\pm$ trees in non-volatile main memory. *Proc. of the VLDB Endowment*, 8(7):786–797, 2015.
- [15] Sakib Chowdhury and Wojciech Golab. A scalable recoverable skip list for persistent memory. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, page 426–428, 2021.
- [16] C. Chu and M. Brockmeyer. Predicate detection modality and semantics in three partially synchronous models. In *7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008*, page 444–450, 2008.
- [17] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 105–118, 2011.
- [18] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, page 259–269, 2018.

- [19] T. David, A. Dragojević, R. Guerraoui, and I. Zablatchi. Log-free concurrent data structures. In *Usenix Annual Technical Conference (ATC)*, page 373–386, 2018.
- [20] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablatchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, page 373–385, 2018.
- [21] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, page 28–40, 2018.
- [22] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [23] R. Guerraoui and R. Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems*, Santa Fe, NM, 2004.
- [24] Jim Handy. What is snia’s persistent memory programming model?, 2019.
- [25] S. Hara, M. D. Hill, and M. M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 775–788, 2020.
- [26] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [27] Terry Ching-Hsiang Hsu, Helge Brügger, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *12th European Conference on Computer Systems (EuroSys)*, page 468–482, 2017.
- [28] J. Huang, M. Liu, K. S. McKinley, and F. X. Lin. Nvoverlay: Enabling efficient and scalable high-frequency snapshotting for non-volatile memory. *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021.
- [29] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent B±tree. In *16th Usenix Conf. on File and Storage Technologies (FAST)*, page 187–200, 2018.

- [30] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing (DISC)*, page 313–327, 2016.
- [31] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 427–442, 2016.
- [32] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.
- [33] W.-H. Kim, J. Seo, J. Kim, and B. Nam. clfb-tree: Cacheline friendly persistent b-tree for nvram. *ACM Transactions on Storage*, 14(1):5:1–5:17, 2018.
- [34] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *ACM SIGPLAN Notices*, 51(4):385–398, 2016.
- [35] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.
- [36] 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 Conf. on File and Storage Technologies (FAST)*, page 257–270, 2017.
- [37] Liang Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian. A comparative study of consistent snapshot algorithms for main-memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):316–330, 2021.
- [38] Nan Li and Wojciech Golab. Brief announcement: Detectable sequential specifications for recoverable shared objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, page 557–560, 2021.
- [39] Zhenxin Li, Bing Jiao, Shuibing He, and Weikuan Yu. Phast: Hierarchical concurrent log-free skip list for persistent memory. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3929–3941, 2022.

- [40] fork(2) - linux manual page. <http://man7.org/linux/man-pages/man2/fork.2.html>. Accessed: 2023-07-09.
- [41] mmap(2) - linux manual page. <http://man7.org/linux/man-pages/man2/mmap.2.html>. Accessed: 2023-07-09.
- [42] msync(3) - linux manual page. <https://linux.die.net/man/3/msync>. Accessed: 2023-09-13.
- [43] M. Liu, J. Xing, K. Chen, and Y. Wu. Building scalable NVM-based B+tree with HTM. In *Intl. Conf. on Parallel Processing (ICPP)*, page 101:1–101:10, 2019.
- [44] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 329–343, 2017.
- [45] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st International Symposium on Microarchitecture (MICRO)*, page 258–270, 2018.
- [46] P. Mahapatra, M. D. Hill, and M. M. Swift. Don’t persist all: Efficient persistent data structures, 2019.
- [47] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, page 215–226, 1989.
- [48] Amirsaman Memaripour, Joseph Izraelevitz, and Steven 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*, page 789–806, 2020.
- [49] Microsoft. *Database Snapshots (SQL Server)*, 2021.
- [50] Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech Golab. A closer look at detectable objects for persistent memory. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*, page 56–64, 2022.

- [51] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.
- [52] M. Nam, H. Cha, K. Jin, J. Seo, and B. Nam. B3-tree: Byte-addressable binary B-tree for persistent memory. *ACM Trans. on Storage*, 16(3):17:1–17:27, 2020.
- [53] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410. ACM, 2012.
- [54] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology*, page 689–694, 2015.
- [55] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dal’i: A periodically persistent hash map. In *International Symposium on Distributed Computing (DISC)*, 2017.
- [56] Jenny Ottmann, Frank Breiting, and Felix Freiling. Defining atomicity (and integrity) for snapshots of storage in forensic computing. In *Proceedings of the Digital Forensics Research Conference Europe (DFRWS EU) 2022*, 2022.
- [57] 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 *Intl Conf on Management of Data (SIGMOD)*, page 371–386, 2016.
- [58] Han Jie Qiu, Sihang Liu, Xinyang Song, Samira Khan, and Gennady Pekhimenko. Pavise: Integrating fault tolerance support for persistent memory applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2023.
- [59] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685. IEEE, 2015.
- [60] Douglas C. Schmidt and Tim Harrison. Double-checked locking. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

- [61] B. Schroeder and G.A. Gibson. Understanding failures in petascale computers. *J. Physics: Conf. Series*, 78:012022–012032, 2007.
- [62] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *3rd VLDB Workshop on In-Memory Data Management and Analytics (IMDM)*, page 4:1–4:8, 2015.
- [63] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.
- [64] Snia nvm programming model.
- [65] Scott D Stoller. Detecting global predicates in distributed systems with clocks. *Distributed Computing*, 13:85–98, 2000.
- [66] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th Usenix Conf. on File and Storage Technologies (FAST)*, page 61–75, 2011.
- [67] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 91–104, 2011.
- [68] S. Vömel and F.C. Freiling. Correctness, atomicity, and integrity: Defining criteria for forensically-sound memory acquisition. *Digital Investigation*, 9:125–137, 2012.
- [69] C. Wang, Q. Wei, L. Wu, S. Wang, C. Chen, X. Xiao, J. Yang, M. Xue, and Y. Yang. Persisting RB-tree into NVM in a consistency perspective. *ACM Trans. on Storage*, 14(1):6:1–6:27, 2018.
- [70] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, page 461–472, 2018.
- [71] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. Montage: A general system for buffered durably linearizable data structures. *CoRR*, abs/2009.13701, 2020.
- [72] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A fast, general system for buffered persistent data structures. In

Proceedings of the 50th International Conference on Parallel Processing, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery.

- [73] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. Montage: A fast, general system for buffered persistent data structures. <https://github.com/urcs-sync/Montage>, 2023. Accessed: 2023.
- [74] Song Wu, Fang Zhou, Xiang Gao, Hai Jin, and Jinglei Ren. Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications. *ACM Transactions on Architecture and Code Optimization*, 15(4), jan 2019.
- [75] 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 Conf. on File and Storage Technologies (FAST)*, page 167–181, 2015.
- [76] X. Zheng and V.K. Garg. An optimal vector clock algorithm for multithreaded systems. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019*, page 2188–2194, 2019.
- [77] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank. Efficient lock-free durable sets. *Proc. of the ACM on Programming Languages*, 3(OOPSLA):128:1–128:26, 2019.

APPENDICES

Appendix A

Alternative Implementation Techniques

The performance of different data structures was assessed in our study. Among the persistent implementations, we considered:

- Dalí [55]
- Friedman [21]
- SOFT [77]
- MOD [25]
- NVTraverse [22]

We also explored transient implementations, including those that operate on DRAM (shown with DRAM(T)) and NVM (shown with NVM(T)). Additionally, we examined a variation of Montage (shown with Montage(T)) that continues to store payloads in NVM but omits all persistence operations. For Montage, we selected parameters that yielded the best performance (based on [72]), such as an epoch length of 10 ms, buffering up to 64 write-backs in each thread during each epoch (when these buffers reach their capacity, the entries that were added earliest are written back to free up space), and using a single background thread to advance the epoch and perform any remaining write-backs.

To conduct a comprehensive analysis, we assessed the performance of these hash tables across three distinct workloads: write-dominant (0% read, 50% insert, 50% delete) as depicted in Figure A.1a, balanced (50% read, 25% insert, 25% delete) as shown in Figure A.1b, and read-dominant (90% read, 5% insert, 5% delete) as presented in Figure A.1c.

Montage shines under the write-dominant and balanced workloads. In Figure A.1b, it surpasses the NVM transient variation at higher thread counts, further demonstrating

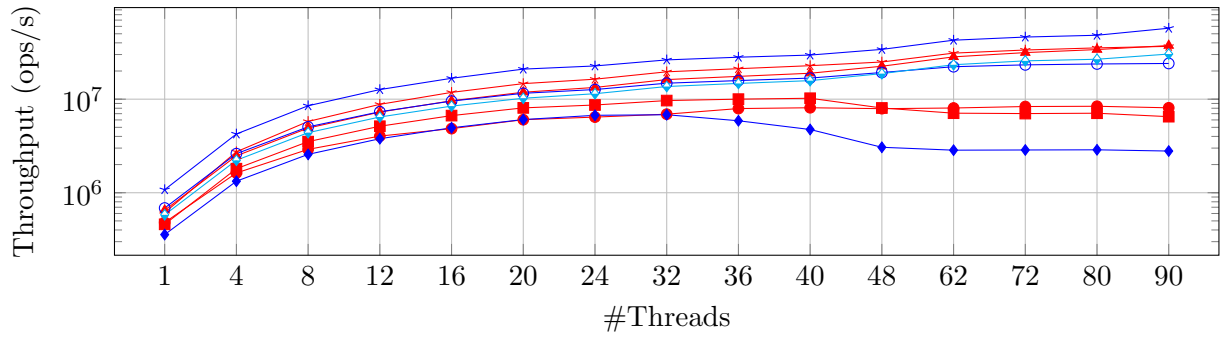
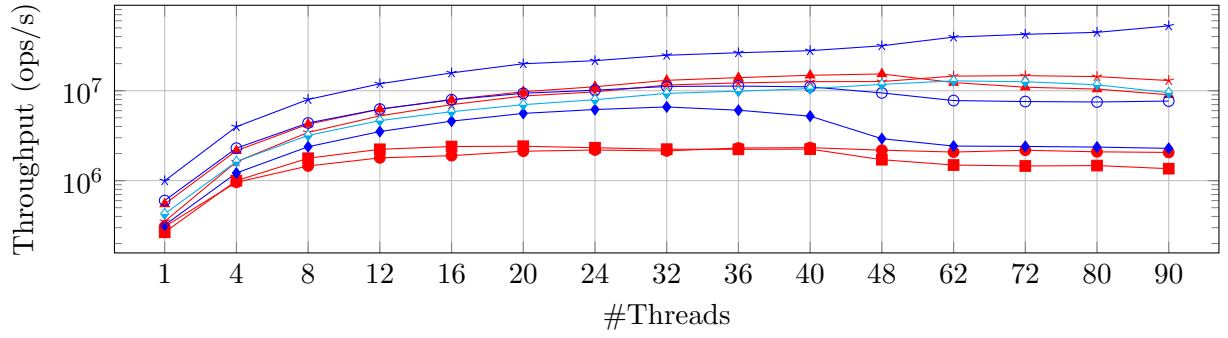
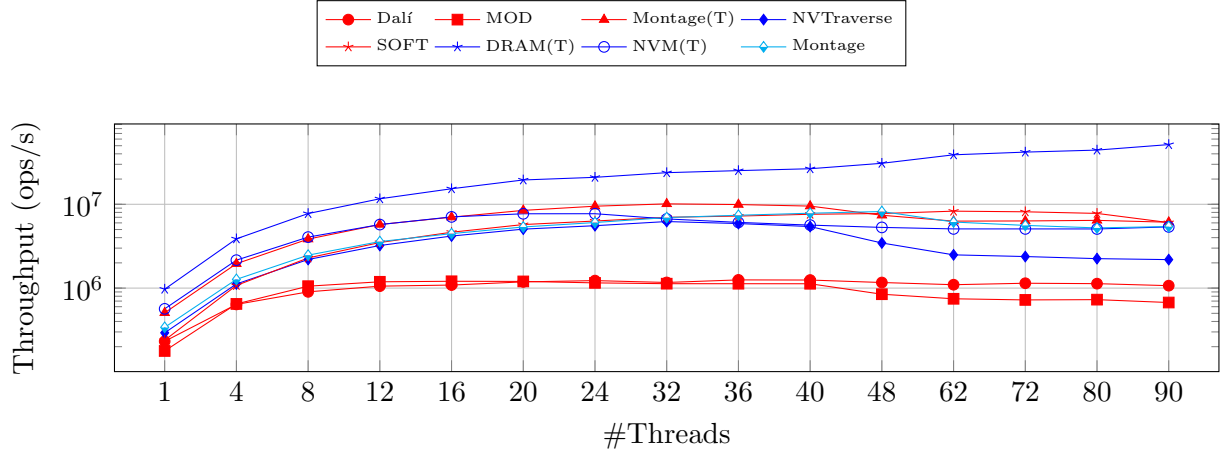
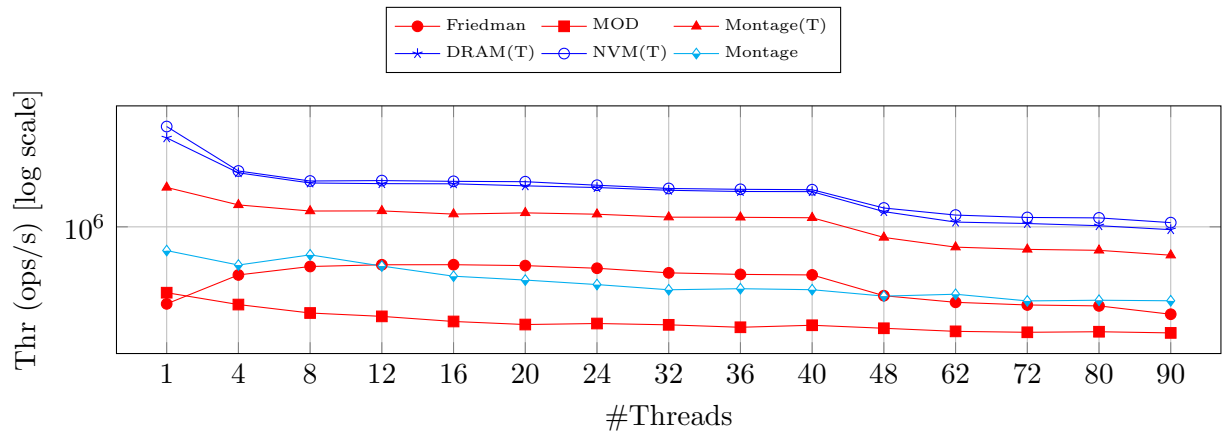


Figure A.1: Throughput of various implementations of a hash map (log-scale y-axis)

the effectiveness of the Montage implementation. Interestingly, Montage exhibits superior performance when compared to SOFT at lower thread counts, but as the number of threads increases, SOFT takes the lead. The performance trend shifts slightly for read-dominant workloads. Although the DRAM transient variation maintains its top position, the SOFT implementation demonstrates impressive scaling capability. At higher thread counts, SOFT outpaces Montage, further solidifying its strong performance under read-heavy workloads. However, SOFT keeps a complete copy of the data in DRAM and only operates on that, which is only applicable for sets and mappings. Consistently, NVTraverse, Dalí, and MOD implementations lag behind other configurations across all tested workloads and thread counts. In conclusion, the Montage hash table has demonstrated solid performance across the board, especially under write-dominant conditions. Nonetheless, SOFT and the transient hash table (DRAM variant) also stand out with strong results, particularly in read-dominant scenarios.

In a 1:1 enqueue:dequeue workload for the queue data structure, Montage exhibits a common trend of decreasing throughput as the number of threads increases, indicating scalability challenges in highly parallel, multi-threaded environments. This is a pattern shared with other implementations such as Friedman and MOD, where Montage consistently shows competitive performance. Although certain implementations initially outperform Montage, they also follow a similar trend of performance decline as the thread count grows. Even those implementations that initially showcase superior performance, like NVM(T) and DRAM(T), also face a decrease in throughput under higher concurrency. This underlines that scalability under high concurrency is a widespread challenge across all tested queue implementations, not limited to Montage.



(a) Queue (enq50deq50)

Figure A.2: Average throughput of alternative implementations of a queue