Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Flat combining for Non-Volatile Main Memory

Luis Morgenstern

Course of Study: Informatik

Examiner: Prof. Dr. Christian Becker

Supervisor: Simon König, M.Sc.

Commenced: September 12, 2024

Completed: March 12, 2025

Abstract

The availability of non-volatile random-access memory (NVRAM) spurred interest in its potential as a foundation for high-performance, persistent data structures. Our work focuses on applying and evaluating the *flat combining* synchronization paradigm on persistent memory systems. We present a novel approach, named persistent flat combining (PFC), along with a simplified version called fast persistent flat combining (FPFC), which is more efficient but has more limited use cases. Our algorithms combine operations from multiple threads into batches, leveraging not only the well-known benefits of general software combining but also the unique characteristics of NVRAM to develop a high-performance, persistent resizable array together with a persistent memory allocator. Our experimental evaluation, conducted on Intel's Optane persistent memory platform, demonstrates the effectiveness of our implementation and shows substantial performance improvements over traditional locking mechanisms and other state-of-the-art persistent software combining techniques. We show that at high concurrency levels, our approach outperforms detectable flat combining (DFC) by a factor greater than 10 and *PBstack* (a persistent stack based on *PBcomb*) by a factor greater than 2.

Kurzfassung

Die Verfügbarkeit non-volatile random-access memory (NVRAM) weckte das Interesse an seinem Potenzial als Grundlage für hochperformante, persistente Datenstrukturen. Unsere Arbeit konzentriert sich auf die Anwendung und Evaluierung des *flat-combining* Synchronisationsparadigmas auf persistenten Speichersystemen. Wir präsentieren einen neuen Ansatz, gennant persistent flat combining (PFC), sowie eine vereinfachte Version namens fast persistent flat combining (FPFC), die effizienter ist, aber enger begrenzte Anwendungsmöglichkeiten hat. Unsere Algorithmen kombinieren Operationen von mehreren Threads zu Stapeln und nutzen dabei nicht nur die bekannten Vorteile des allgemeinen Software-Combinings, sondern auch die einzigartigen Eigenschaften von NVRAM, um ein hochleistungsfähiges, persistentes, größenveränderliches Array zusammen mit einem persistenten Allocator zu entwickeln. Unsere experimentelle Auswertung, die auf Intels Optane persistent memory platform durchgeführt wurde, demonstriert die Effektivität unserer Implementierung und zeigt erhebliche Leistungsverbesserungen gegenüber traditionellen Locking-Mechanismen und anderen aktuellen persistenten Software-Combining-Techniken. Wir zeigen, dass unser Ansatz bei hoher Parallelität detectable flat combining (DFC) um einen Faktor von mehr als 10 und *PBstack* (ein auf *PBcomb* basierender persistenter Stack) um einen Faktor von mehr als als 2, übertrifft.

Contents

1 Introduction **15**

2 Background **17**
2.1 Non-Volatile Random-Access Memory (NVRAM) 17
2.2 Guarantees in case of Errors 18
2.3 Flat Combining 19

3 Related Work **21**

4 Research Objectives **25**

5 Design and Implementation **27**
5.1 Flat Combining Algorithm 27
5.2 Persistent Resizable Array 30
5.3 Persistent Buddy Allocator 36
5.4 Platform-Specific Optimizations 42

6 Evaluation **47**

7 Conclusion and Outlook **53**

Bibliography **55**

List of Figures

6.1 *Push-pop* benchmark results for *PFC*, *FPFC*, *DFC*, and *PBstack* 48
6.2 *Push-pop* benchmark results for *PFC* with and without futex lock mechanism and non-temporal stores 49
6.3 *Push-pop* benchmark results for *FPFC* with and without futex lock mechanism and non-temporal stores 50
6.4 Mixed *swap* and *push-pop* benchmark results for *PFC*, *FPFC*, and *Mutex* 51
6.5 Pure *swap* benchmark results for *PFC*, *FPFC*, and *Mutex* 51
6.6 Mixed *get* and *push-pop* benchmark results for *PFC*, *FPFC*, and *Mutex* 52
6.7 Pure *get* benchmark results for *PFC*, *FPFC*, and *Mutex* 52

List of Algorithms

5.1 FC types and member variables 29
5.2 FC request procedure 30
5.3 FC combine procedure 31
5.4 FC combine-push-pop procedure 32
5.5 FC wait and release procedures 33
5.6 NV-Vector types and member variables 34
5.7 NV-Vector push and pop procedure 35
5.8 NV-Vector reserve procedure 36
5.9 NV-Vector swap procedure 37
5.10 NV-Vector recover procedure 38
5.11 Allocator types and member variables 40
5.12 Allocator request procedure 41
5.13 Allocator release procedure 42
5.14 Allocator recover procedure 43
5.15 Allocator auxiliary procedures (1) 44
5.16 Allocator auxiliary procedures (2) 45

List of Abbreviations

AFC asymmetric flat combining. 22

CAS compare-and-swap. 19

ccNUMA cache-coherent non-uniform memory access. 22

DFC detectable flat combining. 3, 5, 23

DSM distributed shared memory. 22

FPFC fast persistent flat combining. 3, 5, 23

HDD hard disk drive. 17

NUMA non-uniform memory access. 22

NVRAM non-volatile random-access memory. 3, 5, 15

PCM phase-change memory. 17

PFC persistent flat combining. 3, 5, 16

PTM persistent transactional memory. 23

PWB persistent write-back. 18

RAM random-access memory. 15

ReRAM resistive random-access memory. 17

RMR remote memory reference. 22

SSD solid-state drive. 17

1 Introduction

Modern data-intensive applications have to overcome the inherent challenge of delivering high performance, while ensuring strong data durability. On the one hand, fast access to data is crucial for executing computationally intensive tasks effectively. On the other hand, ensuring that critical data is not lost in case of failures or power losses is essential to maintain system reliability and consistency. Different memory and programming concepts offer varying trade-offs between speed and durability.

Traditional volatile memory systems offer the speed required for operation-intensive workloads but fall short when it comes to data persistence. Conversely, conventional persistent storage often incurs significant latency overheads. This dichotomy has spurred the need for innovative memory technologies that bridge the gap between performance and durability, thereby enabling systems to simultaneously meet the demands for speed and reliability. non-volatile random-access memory (NVRAM) emerged as a promising solution in this context. By combining the high-speed characteristics of random-access memory (RAM) with inherent data persistence, NVRAM allows data-intensive applications to operate efficiently without sacrificing durability.

Despite the potential benefits of NVRAM, challenges arise when integrating it into existing software stacks. One such challenge in transitioning legacy software to NVRAM comes from the fact that most of the existing systems are designed under the assumption of volatile main memory. These systems lack built-in mechanisms for ensuring data durability, and many of them fail to address issues such as crash consistency and recovery. Traditional synchronization and data management methods often fall short when applied to persistent memory systems, due to the unique challenges of ensuring consistency and integrity after unexpected failures. Ensuring efficient concurrency control, maintaining data consistency in the presence of failures, and achieving scalability under high parallelism remain challenging goals and often necessitate the development of new programming models and algorithms that can manage persistence at a finer granularity while maintaining efficiency. In balancing the dual requirements of high performance and reliable persistence, novel solutions are developed that redefine data structure design and memory management strategies. This thesis explores concurrent programming techniques that leverage the unique capabilities of NVRAM to deliver both requirements. We propose a solution based on an approach called *flat combining* [HIST10a] and adapt it to the specific characteristics of NVRAM.

Flat combining is a synchronization technique that aggregates multiple concurrent operations into a single batched execution, thereby significantly reducing lock contention and the cost of memory coherence. In this approach, individual threads do not try to obtain a lock for each operation; instead, if the lock is already held, they post their operation requests to a shared combining data structure. The thread holding the lock is the designated combiner thread for this batch of operations and sequentially collects and executes the pending operations within a single critical section. By processing the operations in a batch, the mechanism leverages cache locality, minimizes the frequency of atomic operations, and reduces costly cache invalidations across cores. Further

optimizations were proposed, such as *elimination* [RAB+21], where some operations cancel each other out (e.g. on a stack, a push operation can be canceled by a following pop operation), reducing the number of accesses to the data structure even further.

In the context of NVRAM, this batching approach also reduces the need for frequent persistence instructions, significantly lowering performance costs tied to data durability. When lock contention, cache coherence, and data persistence overheads are combined, concurrency can be severely bottlenecked. By grouping requests and handling them in a single critical section, the number of persistence instructions can be drastically reduced, compared to a naive approach where each operation must be persisted separately. As a result, *flat combining* can be especially effective in preserving high throughput under heavy concurrent loads, while maintaining the consistency guarantees that persistent memory demands.

Throughout the chapters, we will discuss related work, present the implemented data structures and synchronization strategies, and evaluate their effectiveness on real hardware. The thesis aims to provide not only a theoretical foundation for applying *flat combining* on NVRAM but also practical insights into how these techniques perform in real-world scenarios. Ultimately, the goal is to demonstrate that well-engineered concurrent data structures, optimized for persistence and fast recovery, can achieve substantial improvements over generalized synchronization and persistence constructions on modern persistent memory platforms.

In our work, we present a new approach to software-combining on NVRAM, we call persistent flat combining (PFC), which functions as drop-in replacement for a C++ `std::vector`. For this, we implement a resizable array data structure in C++, integrating it with Intel's Optane persistent memory. We apply the flat combining approach to reduce persistence and synchronization overhead, thereby improving the performance of concurrent operations on our data structure. Our decision to implement the underlying data structure as a resizable array, necessitates the need for dynamically sized memory allocations. We therefore also present a custom persistent memory allocator.

Extensive experiments were conducted to compare the performance of our flat combining implementation against traditional locking mechanisms and other state-of-the-art software combining implementations on NVRAM. Our findings indicate that, under high concurrency, the PFC algorithm significantly reduces contention and improves throughput. Detailed performance evaluations revealed lower latency and enhanced scalability, while maintaining a robust recovery behavior in the presence of failures. Our results also show a substantial performance improvement over previous work by Rusanovsky et al. [RAB+21] and Fatourou et al. [FKK22b], demonstrating the effectiveness of our approach compared to other state-of-the-art software combining techniques.

2 Background

In this chapter, we introduce key concepts of persistent memory and concurrent programming. We outline the benefits of NVRAM, e.g. its low latency and byte-addressability, but also the unique durability challenges, compared to traditional volatile memory. Next, we discuss essential consistency models, such as *durable linearizability* and *detectability*, detailing how modern systems ensure operation consistency despite failures. Finally, we examine the *flat combining* synchronization paradigm, which aggregates operations to reduce contention and improve scalability. This chapter provides the necessary background for the design and implementation of our proposed algorithm.

2.1 Non-Volatile Random-Access Memory (NVRAM)

NVRAM represents a significant advancement in memory technology, combining the high-speed access of traditional volatile RAM with the persistence of non-volatile storage devices such as solid-state drives (SSDs) and hard disk drives (HDDs). Unlike conventional RAM, NVRAM retains its data even after a power loss, making it an attractive solution for applications requiring both, high speed and data persistence.

The concept of non-volatile memory has been explored for several decades, with early implementations including magnetic-core memory in the 1950s [Aue52] and flash memory in the 1980s [MI85]. However, these technologies were either too slow or too expensive for widespread use as main memory. Recent advancements in materials science and semiconductor manufacturing have enabled the development of NVRAM technologies, such as phase-change memory (PCM) and resistive random-access memory (ReRAM), which offer the potential for high-speed, high-density, and cost-effective non-volatile memory solutions.

NVRAM offers several advantages over traditional storage solutions. It provides faster access times compared to SSDs and HDDs, significantly enhancing the performance of applications that require frequent data access, while its persistence ensures that data is not lost in the event of a power failure, making it ideal for critical applications requiring high reliability and data integrity. Beyond raw speed, NVRAM eliminates many of the latency bottlenecks associated with mechanical or flash-based storage through its byte-addressability, which allows for more granular data management and efficient random access patterns that improve system responsiveness. Furthermore, its higher durability minimizes the wear and tear issues commonly experienced with flash memory, resulting in a longer lifespan and reduced maintenance overhead. Collectively, these factors make NVRAM an interesting option for modern computing environments where both high performance and data persistence are essential.

2.2 Guarantees in case of Errors

Despite its many advantages, today's NVRAM platforms involve inherent trade-offs, that must be dealt with when designing software for these systems. Current processors use a hierarchy of memory technologies, ranging (in decreasing order of speed) from registers to caches of different levels, to (possibly non-volatile) memory. Although registers and caches provide access speeds that are orders of magnitude faster than NVRAM, their volatile nature makes it challenging to ensure that every critical update is reliably persisted. This duality means that software running on NVRAM must carefully orchestrate cache flushing and memory ordering techniques to ensure both optimal performance and durable data integrity. Multiple abstract models have been proposed to describe the persistence behavior provided by real-world NVRAM systems. By splitting a thread's execution into epochs, where each epoch is a sequence of instructions separated by a persist barrier instruction (also called fence instruction), *epoch persistency* [CNF+09] [PCW14] provides a simple model to reason about the persistence behavior of a system. Memory accesses within an epoch are guaranteed to be persisted before any memory access in the following epoch is persisted, while memory accesses within the same epoch are not ordered with respect to each other and can be persisted concurrently. Although this model is easy to work with, current hardware does not implicitly write back modified data from the cache to the persistent domain, but requires explicit instructions, so-called persistent write-backs (PWBs), to do so. In this model, called *explicit epoch persistency* [IMS16], PWB instructions start the write-back of a cache line to memory, while a fence instruction waits until all previous PWBs have completed. In the Intel x86 architecture, the concept of the PWB instruction is implemented by the `clflush`/ `clflushopt`/`clwb` op-codes and the persist barrier instruction by the `mfence`/`sfence`/`lfence` op-codes [Int24].

The persistent nature of NVRAM also introduces new challenges when designing algorithms which tolerate system failures. In volatile memory systems, it is often sufficient to ensure *linearizability* [HW87] of operations on a data structure, meaning that the system behaves as if operations are executed one after the other in a sequential order i.e. an operation takes effect instantaneously, between its invocation and its response. Since NVRAM retains data, even in the event of a system crash, stricter guarantees are required to ensure a consistent state after recovery. Several models have been proposed to formalize correctness guarantees, taking the possibility of crashes into account. *Persistent atomicity* [GL04] guarantees the consistency of a data structure after a crash, by requiring operations on an object to either take effect (linearize) or have no effect (abort) before any other operation of the calling thread is invoked. The downside of this approach is the lack of locality, i.e. compositions of multiple histories of operations, satisfying *persistent atomicity*, do not necessarily satisfy *persistent atomicity* themselves. *Recoverable linearizability* [BGT16] relaxes the requirements, allowing multiple pending operations in one thread, as long as they do not operate on the same object. *Recoverable linearizability* provides locality but can lead to an inversion of the order of operations, in case of a crash. Both correctness guarantees act under a broad failure model, where threads can crash independently of each other and recover while the other threads continue to operate. Real-world systems often provide a more restricted failure model, where only the whole system can crash, and all threads recover simultaneously after a restart. Izraelevitz et al. [IMS16] showed that under such a restricted failure model, *persistent atomicity* and *recoverable linearizability* are equivalent, and they called this combined condition *durable linearizability*. Informally, a history of operations where each operation, which received a response, is linearized and all operations, which did not receive a response yet, are either linearized or aborted, is *durable linearizable* and by extension *persistent atomic* and *recoverable linearizable*. Sometimes,

the guarantees provided by *durable linearizability* are not sufficient, because it is not possible to detect if an operation was linearized or aborted after a crash, it is hard to ensure operations occur exactly once. *Detectability* [FHMP18] allows detecting, if an operation was aborted or linearized after a crash, giving the possibility to replay aborted operations. *Durable linearizability* combined with *detectability* (also called *detectable recoverability* [ABF+22] [FKK22b] [CJRK23]) allows a straightforward implementation of occurs-exactly-once semantics. *Detectable recoverability* provides stronger guarantees than *durable linearizability* alone but also requires system support, in the form of an additional non-volatile state, given from outside the operation, which can be written from within the operation [BHR20].

2.3 Flat Combining

Flat combining is a synchronization paradigm in which multiple operations from different threads are consolidated into a single combined step. A designated combiner retrieves these operations from a shared request list and executes them sequentially, thereby reducing the need for each thread to acquire its own lock. Periodic routines remove inactive entries and adaptive strategies adjust to varying levels of concurrency.

Early work in software combining dates back to the 1980s, where Yew, Tzeng, et al. [YT+87] explored the possibility to reduce contention in multiprocessor machines, by combining multiple accesses in software. The authors propose a solution, called a combining tree, where a shared variable is modeled as a tree structure, in which each node represents the combined state of its children. The processes, accessing the shared variable, are assigned a leaf of the tree and operations can be performed on this leaf, while multiple modifications to a subtree can be combined into a single update of the parent node. This approach reduces contention by decreasing the number of concurrent accesses to shared memory locations.

Later on, Oyama et al. [OTY99] presented a generic combining technique, sometimes called *OyamaAlg*, suited for modern multicore processors. In their algorithm, a single lock flag guards the shared data structure and a list of pending operations, called context list. The lock flag can be in one of three states: `free`, `locked`, or `conflict`. When a thread is executing an operation on the object, it executes a compare-and-swap (CAS) instruction, setting the lock flag to `locked` if it is set to `free` at the moment. If the flag was already set to `locked` or `conflict`, the current operation is appended to the context list and the lock flag set to `conflict`, if it wasn't already. Lastly, the thread waits for the lock to be released by spinning on the lock flag, until it is set to `free` again. If the CAS instruction succeeds, the thread has exclusive access to the underlying object and executes its operation. After the operation is completed, another CAS instruction sets the lock flag back to `free`, given that no other thread has set it to `conflict` in the meantime. If this second CAS fails, the thread executes all pending operations present in the context list, combining all operations other threads tried to execute while the lock was held.

In 2010, Hendler et al. [HIST10a] introduced the notion of *flat combining* describing an improvement over the technique presented by Oyama et al. [OTY99]. There is still a lock flag, guarding the shared data structure and a list of pending operations, here called announcement or publication list. This list consists of an entry for each thread, recently accessing the shared object and each thread has a thread-local variable, referencing its entry in this list. If a thread is unable to acquire the lock, it writes its operation to its thread-local entry and spins on the result field of this entry. The thread

holding the lock, also called the combiner thread, executes all pending operations, by iterating over the publication list, writing the result of each operation to the result field of the corresponding entry. Every so often, the combiner thread removes inactive entries from the publication list, to prevent threads, which have not accessed the shared object recently, from accumulating in the list. *Flat combining* mainly differs from the technique presented by Oyama et al. [OTY99] by the use of thread-local storage for the list of pending operations. This design significantly reduces cache-coherence traffic, as each thread has its dedicated space in memory, thereby reducing contention on the head of the list, which is a hotspot in *OyamaAlg* [HIST10a]. A drawback of this approach is the additional overhead introduced by the periodic removal of inactive entries and the necessity for the combiner thread to iterate over potentially empty entries in the publication list, whenever a pending request exists in this list.

Flat combining directly addresses several challenges in concurrent programming. It minimizes lock contention by aggregating thread requests, which prevents excessive lock acquisitions and reduces expensive cache-coherence traffic. The combiner thread is able to detect and eliminate redundant operations, such as canceling a push with a pop operation on a stack, thereby boosting efficiency. In highly concurrent systems, where numerous threads compete for shared resources, *flat combining* scales effectively by reducing synchronization overhead. Instead of every thread attempting to acquire contended lock periodically, threads merely write a request to a local memory location, allowing the combiner to process them in batches. This not only reduces the frequency of costly locking operations, but also leverages better cache utilization and minimizes contention hotspots. Moreover, by consolidating work, the system can better exploit CPU pipelines and reduce idle waiting times, leading to improved throughput and responsiveness even under extreme concurrency.

In the context of NVRAM, the benefits of *flat combining* are particularly significant. Non-volatile memory operations can be costly due to persistence instructions. By batching operations, *flat combining* reduces the frequency of these expensive instructions, conserving system resources and reducing latency. This approach is well-suited for high throughput while maintaining data durability. Software combining in general, and *flat combining* in particular, proves advantageous in modern NVRAM systems because cache flushes to the persistent domain remain expensive compared to operations in volatile RAM. By aggregating multiple thread requests, *flat combining* minimizes the number of cache flushes needed, thus reducing the overhead associated with ensuring data consistency and durability. This not only improves performance by decreasing synchronization costs, but also enhances scalability under high-concurrency workloads.

Moreover, software combining in general has shown its effectiveness in modern non-volatile memory systems. Implementations like those described in [RAB+21] and [FKK22b] demonstrate that software combining can enhance performance in environments where persistence and concurrency are critical. Their findings indicate that through combining techniques, data structures can maintain strong guarantees like *durable linearizability* and even *detectable recoverability* while achieving high throughput and low latency, even under heavy concurrent loads. These results underline the potential of software combining to mitigate persistence costs and ensuring efficient operation in persistent memory systems.

3 Related Work

The reduction of hotspot accesses to shared variables by combining multiple operations was first explored in the 1980s in network hardware. Yew, Tzeng, et al. [YT+87] were the first to propose a software-based solution to this problem, based on tree structures. These so called combining trees represent a shared variable, where operations act on the leaves of the tree the operations on multiple children of a node are combined into a single operation on the parent node.

Later work by Oyama et al. [OTY99] introduced a generic combining technique, using atomic CAS operations, bringing the benefits of the software combining approach to modern multicore processors. In their scheme, sometimes called *OyamaAlg*, a lock flag with the tree possible states `free`, `locked` or `conflict` is attached to the synchronized object. When executing an operation on the object, if the flag is `free`, it is set to locked and the operation is executed locally. If the flag is `locked` or `conflict`, the operation is put in a list to be later executed by the thread currently holding the lock. The waiting thread spins on the lock flag until it is set to `free` again. This approach is the foundation for a variety of modern combining techniques, including our implementation PFC. The authors show that their technique outperforms simple synchronization mechanisms, however scalability is limited, because the head of the list gets contended by all threads trying to execute CAS operations on it.

The *flat combining* technique was introduced by Hendler et al. [HIST10a], describing an improvement over *OyamaAlg* by using thread-local storage to reduce the aforementioned contention. Every thread using the object has a thread-local entry in the list of operations. This reduces the number of insertions in the list and therefore the contention on the pointer to the head of the list. On the other hand, additional overhead is introduced, by periodically removing those entries from the list whose threads had not accessed the synchronized object recently. The combining thread must also iterate over possibly empty entries in the request list, each time a pending request exists in this list. The authors show that their implementation massively outperforms *OyamaAlg* and other compared synchronization techniques under high parallelism [HIST10a]. Shortly after, Hendler et al. [HIST10b] explored scalable *flat combining* based synchronous queues, which were integrated into the Java JDK 6.0 to support high-performance thread pools. This integration demonstrated the practical applicability of *flat combining* in widely-used software libraries, providing a real-world example of its benefits. In 2016, Galimullin et al. [GKR16] evaluated different synchronization strategies for *flat combining* by implementing and comparing them in the C++ library `libcds`. In their testing, they found a version using a thread-local mutex, combined with a thread-local condition variable to be the most efficient one. Threads which were not able to acquire the global lock, wait on their thread-local condition variable. The combiner thread signals the completion of an operation by notifying each condition variable separately. Their investigation highlighted how careful tuning of synchronization approaches directly impacts execution times and CPU utilization. Our implementation of PFC uses a similar mechanism to wait for the completion of a request, but instead of a condition variable, we use the Linux futex mechanism to wait in case of high contention. *Flat combining* was further extended by Dice et al. [DMS11], who proposed *flat-combining NUMA*

locks, designed for non-uniform memory access (NUMA) and cache-coherent non-uniform memory access (ccNUMA) architectures, with shared caches for multiple clusters of cores (NUMA nodes), allowing cheap communication between cores of the same node, but not between the cores of different nodes. They combine the MCS lock algorithm [MS91] with *flat combining*, forming a hierarchical locking mechanism, where each NUMA node has its combiner thread, working like the global combiner thread in the original *flat combining* algorithm. Those combiner threads are synchronized over NUMA nodes, using the MCS lock mechanism.

Another approach to software combining was proposed by Fatourou and Kallimanis [FK11], who introduced a wait-free universal construction called *SIM*. *SIM* is similar to *flat combining*, in that each thread has a predefined space, where it can write its operation requests and one thread combines and executes these operations. The main difference is on the one hand the wait-freedom of *SIM* and on the other hand the fact that it uses a *fetch-and-add* object to manage the publication list. This allows for only a constant number of shared memory accesses, but requires *fetch-and-add* instructions on unrealistically large objects. The authors also present a practical version of the construction, called *P-SIM*, which substitutes one large *fetch-and-add* operation with multiple smaller ones, making it possible to be implemented on current hardware. This degrades the performance from $O(1)$ to $O(n+s)$ shared memory accesses, where n is the number of threads and s the size of the synchronized object. Nevertheless, the authors show that *P-SIM* outperforms various state-of-the-art synchronization techniques in their experiments, including *flat combining*. In 2012 Fatourou and Kallimanis [FK12] introduced three more combining algorithms, called *CC-Synch*, *DSM-Synch* and *H-Synch*. *CC-Synch* and *DSM-Synch*, similar to *flat combining*, announce their operations in publication entries and if they are not the combiner thread, spin on the result field inside their corresponding entry. In contrast to *flat combining*, the decision if a thread becomes the combiner is not made by a global atomic variable, but the order of the publication list itself, such that the thread associated with the first entry in the list combines the pending operations. *CC-Synch* is tailored towards systems with coherent caches and performs $O(h+t)$ remote memory references (RMRs), where h is the number of requests the combiner may execute and t the size of the accessed data during these requests. *DSM-Synch* is designed for distributed shared memory (DSM) architectures, without a shared cache, and performs $O(d \cdot h)$ RMRs, where d is the average number of RMRs in a single request. *H-Synch* is a hierarchical version of *CC-Synch*, similar to the *flat-combining NUMA locks* in [DMS11]. Unlike *flat-combining NUMA locks*, which makes use of an MCS lock [MS91], *H-Synch* uses a CLH lock, as described in [MLH94]. The authors show their implementations outperforming other state-of-the-art techniques, including the original *flat combining* approach [HIST10a], *P-SIM* [FK11] and *flat-combining NUMA locks* [DMS11].

With asymmetric flat combining (AFC), Gorelik and Hendler [GH13] introduced a combining algorithm, based on the *flat combining* technique, allowing for wait-free operations. They describe a queue, where enqueue operations are written to a thread-local publication, without waiting for the combiner to execute their operation while only threads, wanting to dequeue an element, have to wait for the combiner to finish. The main difficulty, introduced by this approach, is the need to guarantee *linearizability*, which is done by giving each enqueue operation a timestamp, used to order the operations in the publication list. This consolidation step can get expensive under high load, so the authors propose a parallel version, where multiple threads, waiting for the combiner to finish their dequeue operation, can execute the consolidation step concurrently. In their experimental results, they show that AFC outperforms the classical *flat combining* algorithm [HIST10a] as well as *H-Synch* [FK12].

The rise of non-volatile memory technologies has spurred research into persistent and transactional data structures and synchronization mechanisms. Persistent transactional memory systems aim to integrate atomicity and durability into memory operations. By using techniques such as logging and shadow copying, these systems ensure that modifications to non-volatile data are either fully committed or completely rolled back, even in the event of a failure. In Coburn et al. [CCA+11] proposed *NV-Heaps*, a persistent object system that leverages NVRAM to provide fast and safe persistent objects. They present a framework for building persistent data structures, guaranteeing pointer safety and transactional consistency. Their performance evaluations show that *NV-Heaps* outperform persistence mechanisms, designed for traditional secondary storage systems, highlighting the potential of algorithms, specifically tailored towards non-volatile memory architectures. In the same year, Volos et al. [VTS11] introduced a persistent memory system, called *Mnemosyne*. *Mnemosyne* provides a simple programming model for declaring persistent data and supports efficient transactional updates. In comparison to *NV-Heaps* [CCA+11], no type-safe pointers or garbage collection is provided, but *Mnemosyne* did not require features of the underlying hardware, which were not available in commodity hardware at the time.

In 2018, Friedman et al. [FHMP18] presented three different persistent lock-free queue implementations, specifically designed for NVRAM. They define the concept of *detectability* and show their queue implementations to be *durable linearizable*. Their approach utilizes a logging mechanism, which attaches metadata to each enqueued or dequeued node, ensuring partially completed operations can be detected and finished after a crash. *Flat combining* was first applied to NVRAM by Rusanovsky et al. [RAB+21]. The authors introduced the detectable flat combining (DFC) algorithm, bringing the benefits of software combining to persistent memory systems. They show implementations of stacks and queues, using their algorithm and demonstrate that them outperforming solutions based on general-purpose persistent transactional memorys (PTMs). Their implementation is similar to the algorithm presented by us, but in contrast to PFC, the publication list lives in the persistent domain. This allows for *detectable recoverability* but introduces additional overhead. In Chapter 6, we show that PFC outperforms DFC by a wide margin. In 2022 Fatourou et al. [FKK22b] presented *PBcomb* and *PWFcomb*, two recoverable combining algorithms specifically tailored for persistent memory, where *PBcomb* is blocking and *PWFcomb* is non-blocking. Both algorithms extend ideas from *P-SIM* [FK11], using *fetch-and-add* instructions to manage the publication list. The authors show that their algorithm guarantees *detectable recoverability*, like DFC. According to their experiments, when synchronizing requests to an atomic floating-point number, *PBcomb* outperforms various other blocking synchronization techniques, including the MCS lock algorithm [MS91], *P-SIM* [FK11], *H-Synch* and *CC-Synch* [FK12]. The authors also present an implementation of a stack and queue, called *PBstack* and *PBqueue*, respectively. In further experiments, they show that *PBcomb* outperforms DFC [RAB+21] significantly, in highly parallel workloads. In our performance evaluation, we find that our implementation outperforms *PBcomb*, if a more restricted but faster version of PFC is used, which we call fast persistent flat combining (FPFC).

4 Research Objectives

The primary objective of this thesis is to design and implement data structures that operate linearizable on NVRAM. The research aims to develop and test a novel way of adapting *flat combining* to existing persistent memory architectures. The proposed solution is expected to be usable in real-world applications, meaning it should be easily integrated into existing software stacks and not require uncommon hardware features or system support. For this reason, our implementation only guarantees *durable linearizability* and renounces *detectable recoverability*, as the latter requires additional system support [BHR20]. The provided solution is implemented in C++ and uses Intel's Optane persistent memory architecture, allowing on one hand to use it as a drop-in replacement for existing data structures in a wide range of applications and on the other hand for a direct comparison with other implementations on similar hardware like the implementations of Rusanovsky et al. [RAB+21] and Fatourou et al. [FKK22b].

Our work focuses on implementing a resizable array, such as `std::vector`, that leverages the advantages of *flat combining* while maintaining transactional guarantees. The implementation is expected to be robust and efficient, capable of handling high levels of parallelism. The implemented array should be able to be used as a drop-in replacement for `std::vector`, being usable under a wide range of different workload scenarios.

Because PFC is designed as a drop-in replacement for `std::vector` in real-world systems, we had to make some compromises. Supporting multiple instances in one process and allowing removal of inactive thread publications rule out a simple thread-local variable, since C++ lacks instance-scoped thread-local storage and using pthreads `pthread_key_create` is limited to a relatively small number of keys [Lin24]. Another challenge is the support for a large number of threads, which may use the data structure only sporadically and can exit at any time. Other implementations, like DFC in [RAB+21] and *PBcomb* in [FKK22b], do not address these issues, and can therefore use optimizations, which are not possible in PFC. We also implement a more efficient version of PFC, called FPFC, which cannot easily be used as a drop-in replacement for `std::vector`.

Our design decision to use a resizable array also introduces the challenge of allocating different sized memory regions in a *durable linearizable* way. Other similar implementations, like the one in [RAB+21] and [FKK22b], use underlying data structures, which only require fixed-size memory allocations, allowing for simpler allocation techniques, such as memory pools. To allow different sized blocks of memory to be allocated, we present an implementation of a buddy allocator [Kno65], which is capable of allocating and freeing memory, while guaranteeing *durable linearizability*. Compared to fixed-size memory allocation techniques, this approach allows the underlying data structure to more closely resemble the behavior of structures, used in traditional volatile memory systems. A notable advantage is the small memory footprint of an array with few elements, while still preventing excessive allocations as the array grows in size. With fixed-size memory allocations, a large allocation size leads to high overhead for small arrays, while a small allocation size leads to many allocations/frees, when the array rapidly changes in size.

The final phase of the research involves a performance evaluation of our proposed solution. We compare the performance PFC with a naive approach using a single `std::mutex` to synchronize all operations. We also compare our implementation with two other state-of-the-art persistent software combining implementations, DFC [RAB+21] and *PBcomb* [FKK22b]. The evaluation will measure the performance of our *flat combining* implementation in different access scenarios, particularly under conditions of high parallelism. The results will be systematically analyzed to determine the effectiveness of our implementation in enhancing performance on NVRAM. Additionally, platform-specific optimizations (e.g., x86 non-temporal store instructions and Linux futexes) will be implemented and measured to identify which optimizations are beneficial in specific use cases. This approach aims to provide a nuanced understanding of the conditions under which various optimizations yield performance benefits.

5 Design and Implementation

In this chapter, we present the design and implementation of our proposed solution, called PFC. We show pseudocode excerpts and describe the data structures and synchronization mechanisms used in our implementation, to illustrate the key components of the system.

In the following sections, we first describe our combining algorithm, based on *flat combining*, which is used to synchronize concurrent operations on the underlying array. Next, we explain the implementation of the underlying array and buddy allocator with the adaptations made to ensure *durable linearizability*. Finally, we discuss the platform-specific optimizations implemented to reduce synchronization overhead and improve overall performance.

5.1 Flat Combining Algorithm

In this section, we explain our implementation of the *flat combining* algorithm. Like other *flat combining* based approaches, each thread first declares its operation by recording its operation code and arguments in an announcement list. Next, each thread tries to acquire a global lock that safeguards the sequential data structure, intending to become the combiner. Threads that do not acquire the lock wait for the combiner to process their operation by spinning on their announcement entry, until the result is written back by the combiner. The thread which acquires the lock becomes the combiner, it traverses the announcement list to collate all pending operations, applies them to the data structure, and writes back the corresponding responses into the corresponding announcement entry.

Our implementation of the *flat combining* algorithm is very similar to the original one, described by Hendler et al. [HIST10a]. The main conceptual difference is the use of *elimination*, similar to [HSY04] and [RAB+21]. If a combiner encounters multiple PUSH and POP operations, it can match them up and only apply the remaining PUSH **or** POP operations to the underlying data structure.

Compared to other combining approaches on persistent memory [FKK22b; RAB+21], our approach does not require additional persistence instructions for the announcement of operations. Only when the combiner writes back the remaining values after combining the operations, persistence instructions on the underlying vector are needed. This design choice greatly reduces the number of costly persistence instructions under workloads with many read operations (GET, SIZE, CAPACITY) and/or a balanced mix of PUSH and POP operations. The downside of this approach is that we can only guarantee *durable linearizability* but not *detectable recoverability*.

We now discuss the algorithm in more detail. Algorithm 5.1 shows the types and global variables used in the algorithm. Note that in our real-world implementation in C++, the global variables are member variables of a class, to allow for multiple PFC vectors in the same program. In sake of brevity, we omit the class declaration and put all members in the global scope.

The underlying persistent data structure is a resizable array, held in a variable called $Vector$. The implementation details of the vector are discussed in the next section (Section 5.2). $Futex$ is an atomic variable acting as the global lock, deciding which thread becomes the combiner. $PubHead$ is an atomic pointer to the head of the publication list, which is a singly linked list of publications and $LocalPub$ is a thread-local variable, holding the current thread's entry in this publication list. The variable $PassCnt$ is used as a counter tracking the number of combining rounds, which can be used for scheduling cleanups.

The type Publication represents an entry in the publication list. Each entry contains a $next$ field, containing the next publication in the list, and an age field, which is set to the $PassCnt$ of the last operation and used for cleanup purposes. The $request$ field holds the actual request or response data and contains the index of the element to retrieve (*GET*), the value to push (*PUSH*), the indices of the elements to swap (*SWAP*), or the result of the operation (*RESULT*). Depending on the type of operation, $result$ holds the size or capacity of the underlying vector (*SIZE*, *CAP*), the value of the element at the given index (*GET*), the value of the element that was popped (*POP*), $\top$ if the operation was successful (*PUSH*, *SWAP*), or $\bot$ if the operation failed.

The Request procedure shown in Algorithm 5.2 is responsible for announcing and processing an operation request within the *flat combining* framework. Initially, the procedure calls MaybeInsert to ensure that the thread's local publication is present in the shared publication list. It then writes the incoming request into its publication and invokes the procedure Wait. After returning, the procedure checks if another thread has already processed the request (i.e. the result is not $\bot$), and if so, returns that result immediately. Otherwise, the current thread is now the combiner. It increments the $PassCnt$ counter and calls MaybeInsert again to account for the possibility of the previous combiner removing the thread's publication in the cleanup routine, just before the lock was acquired. In the following call to Combine, the combiner processes all pending requests and applies them to the underlying vector. After return, the combiner checks if it is time to trigger a cleanup routine and releases the lock via Release. Finally, the combiner resets its local request tag to *NONE* and returns the computed result.

In Algorithm 5.3, we present the Combine procedure, which is responsible for processing all pending operations in the publication list. Initially, the procedure initializes three lists: $pushList$, $popList$, and $swapList$, which hold the pending push, pop, and swap operations, respectively. The combiner then iterates over the publication list, processing each operation according to its type. If the operation is a *SIZE*, *CAP* or *GET* request, the combiner immediately computes the result and updates the publication accordingly. If the operation is a *PUSH* or *POP* request, the combiner checks if there are pending operations of the opposite type in $pushList/popList$ and combines the operations with a call to *CombinePushPop* described in Algorithm 5.4. Otherwise, the operation is added to the corresponding list. If the operation is a *SWAP* request, it is simply added to the $swapList$. After iterating over the entire publication list, the combiner checks if there are any pending operations in $swapList$, $pushList$, or $popList$ and calls the corresponding procedure on the underlying vector.

Taking and releasing the global lock is handled by the Wait and Release procedures, shown in Algorithm 5.5. The lock variable $Futex$ can be in three states: *FREE*, *TAKEN*, and *CONTENDED*. If the lock is *FREE*, the thread tries to acquire the lock and become the combiner. Else if the lock is *TAKEN*, the thread spins for $SpinCount$ iterations, waiting for the combiner to process its operation. If the combiner has not finished after $SpinCount$ iterations, the lock is set to *CONTENDED*. A contended lock causes the thread to go to sleep and wait for the combiner to wake it up. This is done

Algorithm 5.1 FC types and member variables

```
1:  enum RequestTag { NONE, RESULT, SIZE, CAP, GET, PUSH, POP, SWAP }

2:  enum FutexState { FREE, TAKEN, CONTENDED }

3:  union Result
4:      index : int
5:      value : value ∪ ⊥ ∪ ⊤
6:  end union

7:  union RequestData
8:      result : Result
9:      get : int
10:     push : value
11:     swap : int × int
12: end union

13: struct Request
14:     tag : atomic<RequestTag>
15:     data : RequestData
16: end struct

17: struct Publication
18:     request : Request
19:     age : int
20:     next : Publication ∪ ⊥
21: end struct

22: global
23:     thread_local LocalPub : Publication
24:     PubHead : atomic<Publication> ∪ ⊥
25:     PassCnt : int
26:     Futex : atomic<FutexState>
27:     Vector : Vector
28: end global

29: constant
30:     CleanupCount : int
31:     SpinCount : int
32: end constant
```

Algorithm 5.2 FC request procedure

```
 1: procedure Request(req : Request)
 2:     MaybeInsert
 3:     LocalPub.request ← req
 4:     result ← Wait
 5:     if result ≠ ⊥ then
 6:         return result
 7:     end if
 8:     PassCnt ← PassCnt + 1
 9:     MaybeInsert
10:     Combine
11:     if PassCnt ≡ 0  mod CleanupCount then
12:         Cleanup
13:     end if
14:     Release
15:     LocalPub.request.tag ← NONE
16:     return LocalPub.request.data.result
17: end procedure
```

by calling the FutexWait and FutexWake procedures, using the Linux futex mechanism. Further details on the futex mechanism can be found in Section 5.4. Since it is possible that the combiner removed the thread's publication in the cleanup routine, the thread must call MaybeInsert regularly, to check if its publication is still present in the publication list.

5.2 Persistent Resizable Array

In this section, we describe the implementation of the resizable array, living in the persistent domain. The vector is implemented as a contiguous array with variable size and capacity, similar to the `std::vector` in the C++ standard library.

The data types and variables used in the vector implementation are shown in Algorithm 5.6. The vector is represented by the global variables $Data$, Cap, and $Size$, which hold the allocated memory, the capacity (number of elements the vector can hold without reallocation), and the size (number of valid elements in the vector), respectively. As in the description of the flat combining algorithm, the global variables are member variables of a class in the real-world implementation, but omitted here for brevity.

The vector supports six operations: *SIZE*, *CAP*, *GET*, *PUSH*, *POP*, and *SWAP*. The operations *SIZE*, *CAP*, and *GET* are read-only operations, returning the size, capacity, and the value of an element at a given index, respectively. Those operations are trivial to implement and do not require any persistence instructions or prudence, when adapted to NVRAM. Given enough capacity, *PUSH* and *POP* operations are also straightforward to implement. *POP* simply decreases the size, while *PUSH* first writes the new values to the memory after the last element and then increases the size. If this order is abided by, both kinds of operation are *durable linearizable*. Only *SWAP* operations and *PUSH* operations, exceeding the capacity, require special care. It is not possible to atomically write

Algorithm 5.3 FC combine procedure

```
 1: procedure COMBINE
 2:     pushList ← ∅, popList ← ∅, swapList ← ∅
 3:     pub ← PubHead
 4:     repeat
 5:         pub.age ← PassCnt
 6:         if pub.request.tag = SIZE then
 7:             pub.request.data.result.index ← Vector.size
 8:             pub.request.tag ← RESULT
 9:         else if pub.request.tag = CAP then
10:             pub.request.data.result.index ← Vector.capacity
11:             pub.request.tag ← RESULT
12:         else if pub.request.tag = GET then
13:             pub.request.data.result.value ← VECTORGET(pub.request.data.get)
14:             pub.request.tag ← RESULT
15:         else if pub.request.tag = PUSH then
16:             if popList = ∅ then
17:                 PUSH(pushList, pub)
18:             else
19:                 other ← POP(popList)
20:                 COMBINEPUSHPOP(pub, other)
21:             end if
22:         else if pub.request.tag = POP then
23:             if pushList = ∅ then
24:                 PUSH(popList, pub)
25:             else
26:                 other ← POP(pushList)
27:                 COMBINEPUSHPOP(other, pub)
28:             end if
29:         else if pub.request.tag = SWAP then
30:             PUSH(swapList, pub)
31:         end if
32:         pub ← pub.next
33:     until pub = ⊥
34:     if swapList ≠ ∅ then
35:         VECTORSWAP(swapList)
36:     end if
37:     if pushList ≠ ∅ then
38:         VECTORPUSH(pushList)
39:     end if
40:     if popList ≠ ∅ then
41:         VECTORPOP(popList)
42:     end if
43: end procedure
```

Algorithm 5.4 FC combine-push-pop procedure

1: **procedure** CombinePushPop($push$: Publication, pop : Publication)
2: $value \leftarrow push.request.data.push$
3: $push.request.data.result.index \leftarrow Vector.size$
4: $push.request.tag \leftarrow RESULT$
5: $pop.request.data.result.value \leftarrow value$
6: $pop.request.tag \leftarrow RESULT$
7: **end procedure**

more than one value to NVRAM, and therefore those operations require an additional log structure to ensure *durable linearizability*. The type of this log structure is called Op and is either empty (*NOP*), contains a list of OpSwap structures (*SWAP*), or a single OpReplace structure (*REPLACE*). The OpSwap structure contains the indices of the elements to swap and their values before the swap, while the OpReplace structure contains the new allocated memory and the new capacity of the vector. In case of a system crash, the vector can be restored by replaying the operations in the log structure.

The *PUSH* and *POP* operations are implemented in the VectorPush and VectorPop procedures, as shown in Algorithm 5.7. The VectorPush procedure first checks if there is enough capacity to store the new elements. If not, it reserves more memory by calling the Reserve procedure. It then writes the new values to the memory and ensures that the changes are persisted before updating the size of the vector by calling CLWB and SFence. After updating the size, the procedure sets the result of the operations to $\top$ and the tag to *RESULT* for each publication. The VectorPop procedure first calculates the new size of the vector and updates it. It then writes the values of the popped elements to the publications or $\perp$ if there are no more elements to pop and finally sets the tag to *RESULT*. Both procedures ensure that the size is written back to NVRAM before setting the result and tag of the publications.

In Algorithm 5.8, we present the Reserve procedure, which is responsible for allocating more memory when the vector needs to grow. The procedure first allocates new memory, copies the old data to the new memory, and writes the new memory and capacity to the log. It then ensures that the log is persisted before deallocating the old memory and confirming the allocation. After updating the data and capacity of the vector, the procedure ensures that the changes are persisted before clearing the log by setting the tag to *NOP*. Finally, the deallocation of the old memory is confirmed, before the procedure returns.

The VectorSwap procedure, shown in Algorithm 5.9, is responsible for swapping multiple pairs of elements in the vector. It first constructs a log of the swap operations and persists it. It then iterates over the publications, swapping the elements in the vector and ensuring that the changes are persisted before clearing the log by setting the tag to *NOP*. Finally, the procedure sets the result of the operations to $\top$ or $\perp$, depending on whether the indices are within the size of the vector, and sets the tag to *RESULT*. The procedure SwapPersistLog is called to construct the log and persist it. It saves the indices and values of the elements to swap in the log and ensures that the log is persisted before any changes are made to the vector.

If a system crash occurs during a *SWAP* operation or while reserving memory in the Reserve procedure, the VectorRecover procedure, shown in Algorithm 5.10, is responsible for restoring the vector to a consistent state. In case of a crash during a *SWAP* operation, the log structure contains

Algorithm 5.5 FC wait and release procedures

```
 1: procedure Wait
 2:     loop
 3:         if Futex = FREE then
 4:             locked ← CompareAndSwap(Futex, FREE, TAKEN)
 5:             if locked then
 6:                 return ⊥
 7:             end if
 8:         end if
 9:         if Futex = TAKEN then
10:             for i ← 0 to SpinCount do
11:                 if LocalPub.request.tag = RESULT then
12:                     LocalPub.request.tag ← NONE
13:                     return LocalPub.request.data.result
14:                 end if
15:                 Yield
16:             end for
17:             contended ← CompareAndSwap(Futex, TAKEN, CONTENDED)
18:             if ¬contended then
19:                 MaybeInsert
20:                 continue
21:             end if
22:         end if
23:         if Futex = CONTENDED then
24:             FutexWait(Futex, CONTENDED)
25:             if LocalPub.request.tag = RESULT then
26:                 LocalPub.request.tag ← NONE
27:                 return LocalPub.request.data.result
28:             end if
29:         end if
30:         MaybeInsert
31:     end loop
32: end procedure

33: procedure Release
34:     before ← Swap(Futex, FREE)
35:     if before = CONTENDED then
36:         FutexWake(Futex)
37:     end if
38: end procedure
```

Algorithm 5.6 NV-Vector types and member variables

```
 1: enum OpTag { NOP, SWAP, REPLACE }

 2: struct OpSwap
 3:     idx_a : INT
 4:     idx_b : INT
 5:     val_a : VALUE
 6:     val_b : VALUE
 7: end struct

 8: struct OpReplace
 9:     data : ARRAY<VALUE>
10:     cap : INT
11: end struct

12: struct Op
13:     tag : OpTag
14:     data : ARRAY<OpSwap> ∪ OpReplace
15: end struct

16: global
17:     Data : ARRAY<VALUE>
18:     Cap : INT
19:     Size : INT
20:     Log : Op
21: end global
```

a list of OpSwap entries, each consisting of the indices of the elements to swap and their values before the operation. In most cases, the values at index idx_a and idx_b can be overwritten with the val_b and val_a respectively, but because of the possibility of multiple *SWAP* operations acting on the same elements, some care must be taken. The values stored in val_a and val_b are read from memory before any changes are made and therefore, if another operation has already swapped an element at index idx_a or idx_b, the values in val_a and/or val_b are outdated. An easy solution to this problem is, to just read the values from the underlying memory, if such an overlapping *SWAP* operation occurs. To show the correctness of this approach, we can consider the first operation. Since no other operation has been applied to the vector, the values in val_a and val_b are up-to-date and after the operation is applied, all values in subsequent operations coming from idx_a and idx_b are invalidated but the underlying memory at idx_a and idx_b contains the correct values. The case for following operations is similar, we just have to make sure, we detect an overlap with an earlier operation and, in such a case, read the value from memory. If a crash occurs while reserving new memory, the log structure contains the new allocated memory in $data$ and the new capacity in cap. The recovery procedure for this case is more straightforward and only replays the possibly missed instructions in Reserve. An important aspect are the guarantees given for the function calls interacting with the allocator. It is possible for a call to Allocate to be forgotten, if a crash occurs after the memory has been allocated but before the tag of the log structure is set to *REPLACE*, such a call might never be followed by a call to ConfirmAllocation. Deallocate and ConfirmDeallocation might be

Algorithm 5.7 NV-Vector push and pop procedure

```
 1: procedure VectorPush(publications : array<Publication>)
 2:     if Size + |publications| > Cap then
 3:         Reserve(Size + |publications|)
 4:     end if
 5:     idx ← Size
 6:     for p ∈ publications do
 7:         Data[idx] ← p.request.data.push
 8:         CLWB(Data[idx])
 9:         idx ← idx + 1
10:     end for
11:     SFence
12:     SetSize(idx)
13:     for p ∈ publications do
14:         p.request.data.result.value ← ⊤
15:         p.request.tag ← RESULT
16:     end for
17: end procedure

18: procedure VectorPop(publications : array<Publication>)
19:     oldSize ← Size
20:     if oldSize ≥ |publications| then
21:         newSize ← oldSize − |publications|
22:     else
23:         newSize ← 0
24:     end if
25:     SetSize(newSize)
26:     idx ← newSize
27:     for p ∈ publications do
28:         if idx < oldSize then
29:             p.request.data.result.value ← Data[idx]
30:         else
31:             p.request.data.result.value ← ⊥
32:         end if
33:         p.request.tag ← RESULT
34:         idx ← idx + 1
35:     end for
36: end procedure

37: procedure SetSize(newSize : int)
38:     Size ← newSize
39:     CLWB(Size)
40:     SFence
41: end procedure
```

Algorithm 5.8 NV-Vector reserve procedure

```
1: procedure RESERVE(newCap : INT)
2:     oldData ← Data
3:     newData ← ALLOCATE(newCap)
4:     MEMCPY(newData, oldData, Size)
5:     CLWB(newData)
6:     Log.data ← (newData, newCap)
7:     CLWB(Log.data)
8:     SFENCE
9:     Log.tag ← REPLACE
10:    CLWB(Log.tag)
11:    SFENCE
12:    DEALLOCATE(oldData)
13:    CONFIRMALLOCATION(newData)
14:    Data ← newData, Cap ← newCap
15:    CLWB(Data, Cap)
16:    SFENCE
17:    Log.tag ← NOP
18:    CLWB(Log.tag)
19:    SFENCE
20:    CONFIRMDEALLOCATION(oldData)
21: end procedure
```

called more than once, with the same parameters, when the log is replayed. CONFIRMALLOCATION might never be called, even though the memory has been successfully reserved, because a crash can occur after $Data$ was already set to the new memory, but before the deallocation could be confirmed. For a more detailed explanation on how this protocol prevents memory leaks and ensures *detectability*, refer to Section 5.3.

5.3 Persistent Buddy Allocator

This section describes the implementation of a persistent allocator, which is used to allocate and deallocate memory for the vector. The allocator is based on the well-known buddy allocation technique, first described by Knowlton [Kno65] and was adapted to satisfy *durable linearizability* as well as *detectable recoverability*. The buddy allocator can allocate memory areas of 2^k bytes, where k is called the order of the memory area. A memory area of order k has an adjacent buddy area of the same size, which together form a memory area of order $k + 1$. This allows the allocator to split and merge memory areas efficiently, while keeping external fragmentation low. In sake of brevity, we omit some well-known details of the allocator and focus on the parts that are relevant for the persistent implementation.

In Algorithm 5.11, we present the types and variables used in the allocator implementation. The allocator is represented by the global variables $FreeArea$ and Log, which hold the free memory areas and the log of the allocator operations, respectively. The $FreeArea$ variable is an array of free memory areas (structure PAGE), indexed by the order of the memory area. Memory areas

Algorithm 5.9 NV-Vector swap procedure

```
1: procedure VectorSwap(publications : array<Publication>)
2:     SwapPersistLog(publications)
3:     for p ∈ publications do
4:         idx_a ← p.request.data.swap.idx_a, idx_b ← p.request.data.swap.idx_b
5:         if idx_a < Size ∧ idx_b < Size then
6:             t ← Data[idx_a]
7:             Data[idx_a] ← Data[idx_b]
8:             Data[idx_b] ← t
9:             CLWB(Data[idx_a], Data[idx_b])
10:        end if
11:    end for
12:    SFence
13:    Log.tag ← NOP
14:    CLWB(Log.tag)
15:    SFence
16:    for p ∈ publications do
17:        if p.request.data.swap.idx_a < Size ∧ p.request.data.swap.idx_b < Size then
18:            p.request.data.result.value ← ⊤
19:            p.request.tag ← RESULT
20:        else
21:            p.request.data.result.value ← ⊥
22:            p.request.tag ← RESULT
23:        end if
24:    end for
25: end procedure

26: procedure SwapPersistLog(publications : array<Publication>)
27:    op ← ∅
28:    for p ∈ publications do
29:        idx_a ← p.request.data.swap.idx_a, idx_b ← p.request.data.swap.idx_b
30:        if idx_a < Size ∧ idx_b < Size then
31:            val_a ← Data[idx_a], val_b ← Data[idx_b]
32:            Push(op, (idx_a, idx_b, val_a, val_b))
33:        end if
34:    end for
35:    Log.data ← op
36:    CLWB(Log.data)
37:    SFence
38:    Log.tag ← SWAP
39:    CLWB(Log.tag)
40:    SFence
41: end procedure
```

Algorithm 5.10 NV-Vector recover procedure

```
1: procedure VectorRecover
2:     if Log.tag = SWAP then
3:         for i ← 0 to |Log.data| do
4:             log ← Log.data[i]
5:             val_a ← log.val_a, val_b ← log.val_b
6:             for j ← 0 to i do
7:                 other ← Log.data[j]
8:                 if other.idx_a = log.idx_a ∨ other.idx_b = log.idx_a then
9:                     val_a ← Data[log.idx_a]
10:                end if
11:                if other.idx_a = log.idx_b ∨ other.idx_b = log.idx_b then
12:                    val_b ← Data[log.idx_b]
13:                end if
14:            end for
15:            Data[log.idx_a] ← val_b
16:            Data[log.idx_b] ← val_a
17:            CLWB(Data[log.idx_a], Data[log.idx_b])
18:        end for
19:        SFence
20:        Log.tag ← NOP
21:        CLWB(Log.tag)
22:        SFence
23:    else if Log.tag = REPLACE then
24:        p ← ⊥
25:        if Data ≠ Log.data.data then
26:            p ← Data
27:            Deallocate(Data)
28:            ConfirmAllocation(Log.data.data)
29:            Data ← Log.data.data
30:            CLWB(Data)
31:        end if
32:        Cap ← Log.data.cap
33:        CLWB(Cap)
34:        SFence
35:        Log.tag ← NOP
36:        CLWB(Log.tag)
37:        SFence
38:        if p ≠ ⊥ then
39:            ConfirmDeallocation(p)
40:        end if
41:    end if
42: end procedure
```

of the same order are stored as doubly linked lists, where each memory area holds a reference to the next and previous memory area in the list. In addition, the PAGE structure contains the order of the memory area, the payload (actual usable memory), and a header that is used to store information about at which order the memory area was split or merged. The Log variable holds the current operation of the allocator, which can be either a request to allocate or deallocate memory (*REQUEST*, *RELEASE*) or a no-operation (*NOP*). After a system crash, the allocator can be restored by replaying the operations in the log structure.

Algorithm 5.12 shows the REQUEST procedure, which is responsible for allocating memory areas of a given order. The procedure first searches for a free memory area of the requested order in the $FreeArea$ array, starting from the requested order and increasing the order until a free memory area is found, or the maximum order is reached. If an area with enough space is found, the procedure writes the operation to the log, removes the memory area from the list of free areas, and marks it as used in the current order. If the previously found memory area is bigger than the requested one, the procedure splits the memory area into smaller areas, inserts the buddies into the free list and also marks the area as used in the lower orders. After splitting the memory area, the procedure ensures that the changes are persisted before clearing the log by setting the tag to *NOP* and returning the allocated memory area. If no free memory area is found, the procedure returns $\perp$.

The deallocation of memory areas is handled by the RELEASE procedure, shown in Algorithm 5.13. The procedure first tries to merge the released memory area with its buddy, creating a new memory area of the next higher order. This process is repeated until the buddy of the current memory area is not free, or the maximum order is reached. After the loop, the procedure writes the operation to the log, marks the memory area as unused in the current order, and adds it to the free list. In the next step, All previously merged buddies are set to unused in the lower orders and removed from the free list. At the end of the procedure, it is ensured that the changes are persisted before clearing the log by setting the tag to *NOP*.

In case a system crash occurs, during the REQUEST or RELEASE procedure, the RECOVER procedure, shown in Algorithm 5.14, is responsible for restoring the allocator to the state, before the failed operation. To revert partial operations, the procedure essentially works backwards through the changes possibly made by the failed operation and undoes them. In case a *REQUEST* operation failed, the first step is to remove the split buddies from the free list and mark them as unused. After that, the memory area is set to unused and added back to the free list. If the system crash occurred during a *RELEASE* operation, the procedure first splits the previously merged buddies, inserts them into the free list and marks them as used. At the end, the procedure removes the merged memory area from the free list and sets it to used. At the end of the recovery process, the reverted changes are persisted, and the log is cleared. Note that the RECOVER procedure just rolls back the changes made by the failed operation, but since the parameters of the failed operation are stored in the log ($requestOrder, releaseOrder, releasePage$), the operation can easily be replayed to allow for *detectability*.

An important aspect of ensuring consistency in case of crashes while allocating and deallocating, is the way the allocator is called by the vector. As shown in Algorithm 5.8, the vector first allocates memory, before persisting the allocated memory in the log. If a crash occurs before the memory is persisted, the allocator might have allocated memory that is not referenced by the vector. To prevent memory leaks, the allocator temporarily saves a reference to the allocated space in persistent memory, before returning it to the caller. This reference is removed, when the allocation is later confirmed by the caller. A similar approach is used for deallocations, where instead of deallocating

Algorithm 5.11 Allocator types and member variables

```
1:  enum AreaOpTag { NOP, REQUEST, RELEASE }

2:  struct Page
3:      header : int
4:      order : int
5:      next : Page ∪ ⊥
6:      prev : Page ∪ ⊥
7:      payload : array<byte>
8:  end struct

9:  struct AreaOpRequest
10:     requestOrder : int
11:     pageOrder : int
12:     page : Page
13:     next : Page ∪ ⊥
14: end struct

15: struct AreaOpRelease
16:     releaseOrder : int
17:     pageOrder : int
18:     releasePage : Page
19:     freeFirst : Page ∪ bot
20:     page : Page
21:     next : Page ∪ ⊥
22:     prev : Page ∪ ⊥
23: end struct

24: struct AreaOp
25:     tag : AreaOpTag
26:     data : AreaOpRequest ∪ AreaOpRelease
27: end struct

28: global
29:     FreeArea : array<Page ∪⊥>
30:     Log : AreaOp
31: end global

32: constant
33:     MaxOrder : int
34: end constant
```

Algorithm 5.12 Allocator request procedure

```
 1: procedure REQUEST(order : INT)
 2:     requestOrder ← order
 3:     loop
 4:         if order > MaxOrder then
 5:             return ⊥
 6:         end if
 7:         free ← FreeArea[order]
 8:         if free ≠ ⊥ then
 9:             break
10:         end if
11:         order ← order + 1
12:     end loop
13:     WRITELOG(REQUEST, (requestOrder, order, free, free.next))
14:     if free.next ≠ ⊥ then
15:         free.next.prev ← ⊥
16:     end if
17:     FreeArea[order] ← free.next
18:     CLWB(FreeArea[order])
19:     SETUSED(free, order)
20:     while order > requestOrder do
21:         order ← order − 1
22:         buddy ← SPLITPAGE(free, order)
23:         FreeArea[order] ← buddy
24:         CLWB(FreeArea[order])
25:         SETUSED(free, order)
26:     end while
27:     CLWB(free)
28:     SFENCE
29:     WRITELOG(NOP, ⊥)
30:     return free
31: end procedure
```

right away, a reference to the memory, which is about to be deallocated, is temporarily saved in persistent memory. An explicit call to confirm the deallocation is needed to remove the reference and really deallocate the memory. The caller must guarantee, that in case of a crash, all previously allocated but unconfirmed memory it wants to use, is confirmed in the recovery phase. It is acceptable, that the caller might confirm the allocation before and after the crash, but once it is confirmed for the first time, the caller is responsible for the memory until it is deallocated. Similarly, the call to deallocate can appear before the crash and after the crash, but once it occurs for the first time, the caller must ensure that the memory is not used anymore. If a system crash occurs, after memory is deallocated but not confirmed, it is acceptable for the caller to never confirm said deallocation. If the allocator restarts after a crash, it waits for the recovery phase of all callers to finish, before deallocating all memory that was allocated before the crash, but not confirmed

Algorithm 5.13 Allocator release procedure

```
 1: procedure Release(page : Page, order : int)
 2:     releasePage ← page
 3:     releaseOrder ← order
 4:     while order < MaxOrder do
 5:         buddy ← GetBuddy(page, order)
 6:         if IsUsed(buddy, order) then
 7:             break
 8:         end if
 9:         page ← MergePage(page, buddy)
10:         order ← order + 1
11:     end while
12:     WriteLog(RELEASE, (releaseOrder, order, releasePage,
            FreeArea[order], page, page.next, page.prev))
13:     SetUnused(page, order)
14:     InsertFree(page, order)
15:     while order > releaseOrder do
16:         order ← order − 1
17:         buddy ← GetBuddy(page, order)
18:         if ¬ IsUsed(page, order) then
19:             CLWB(page)
20:             t ← page
21:             page ← buddy
22:             buddy ← t
23:         end if
24:         SetUnused(page, order)
25:         RemoveBuddy(page, order)
26:     end while
27:     CLWB(page)
28:     SFence
29:     WriteLog(NOP, ⊥)
30: end procedure
```

during the recovery. It also deallocates all memory that was deallocated before the crash, but never confirmed. This protocol ensures that no memory leaks or double deallocations occur, while still allowing for *detectability*.

5.4 Platform-Specific Optimizations

In this section, we describe platform-specific optimizations used in the implementation of our *flat combining* algorithm. These optimizations consist of using the Linux futex mechanism to reduce the load on the processor in case of high contention and the use of non-temporal stores to bypass the processor cache and write directly to NVRAM without the need to flush the cache explicitly.

Algorithm 5.14 Allocator recover procedure

```
 1: procedure RECOVER
 2:     if Log.tag = REQUEST then
 3:         for order ← Log.data.requestOrder to Log.data.pageOrder do
 4:             FreeArea[order] ← ⊥
 5:             CLWB(FreeArea[order])
 6:             SETUNUSED(Log.data.page, order)
 7:         end for
 8:         SETUNUSED(Log.data.page, Log.data.pageOrder)
 9:         CLWB(Log.data.page)
10:         if Log.data.next ≠ ⊥ then
11:             Log.data.next.prev ← Log.data.page
12:             CLWB(Log.data.next.prev)
13:         end if
14:         FreeArea[Log.data.pageOrder] ← Log.data.page
15:         CLWB(FreeArea[Log.data.pageOrder])
16:     else if Log.tag = RELEASE then
17:         page ← Log.data.page
18:         for order ← Log.data.releaseOrder to Log.data.pageOrder do
19:             buddy ← GETBUDDY(page, order)
20:             INSERTBUDDY(buddy, order)
21:             SETUSED(page, order)
22:             CLWB(page)
23:             page ← MERGEPAGE(page, buddy)
24:         end for
25:         REMOVEFREE(page)
26:         SETUSED(page, Log.data.pageOrder)
27:         CLWB(page)
28:     end if
29:     SFENCE
30:     WRITELOG(NOP, ⊥)
31: end procedure
```

The Linux futex mechanism (*fast user space mutex*) was introduced by Franke et al. [FRK02] as a way to reduce the overhead of acquiring and releasing a lock by using atomic operations in user space and only using comparatively expensive system calls in case of contention. For this to work, the futex mechanism uses a futex variable, which is a 32-bit integer, to store the state of the lock. To acquire the lock, a thread tries to atomically set the futex variable, using a CAS operation. If the lock is already taken, the thread can go to sleep using the futex system call, which takes the previously read value of the futex variable as an argument. The futex system call puts the thread to sleep and returns, if the futex variable is changed by another thread or if the futex variable does not have the expected value, given as an argument. This mechanism allows the implementation of locking procedures that stay in user space in the common case of no contention, but can handle contention efficiently by putting the thread to sleep, reducing unnecessary load on the processor.

Algorithm 5.15 Allocator auxiliary procedures (1)

```
 1: procedure WriteLog(tag : AreaOpTag, data : AreaOpRequest ∪ AreaOpRelease ∪ ⊥)
 2:     if data ≠ ⊥ then
 3:         Log.data ← data
 4:         CLWB(Log.data)
 5:         SFence
 6:     end if
 7:     Log.tag ← tag
 8:     CLWB(Log.tag)
 9:     SFence
10: end procedure

11: procedure InsertFree(page : Page, order : int)
12:     if FreeArea[order] ≠ ⊥ then
13:         FreeArea[order].prev ← page
14:         CLWB(FreeArea[order].prev)
15:     end if
16:     page.prev ← ⊥
17:     page.next ← FreeArea[order]
18:     FreeArea[order] ← page
19:     CLWB(page, page.next, page.prev)
20: end procedure

21: procedure RemoveFree(page : Page)
22:     FreeArea[Log.data.pageOrder] ← Log.data.freeFirst
23:     free ← FreeArea[Log.data.pageOrder]
24:     CLWB(FreeArea[Log.data.pageOrder])
25:     page.next ← Log.data.next
26:     page.prev ← Log.data.prev
27:     if FreeArea[Log.data.pageOrder] ≠ ⊥ then
28:         FreeArea[Log.data.pageOrder].prev ← ⊥
29:         CLWB(FreeArea[Log.data.pageOrder].prev)
30:     end if
31: end procedure
```

In our implementation (shown in Algorithm 5.1 and Algorithm 5.5), the futex variable can be in three states: *FREE*, *TAKEN*, and *CONTENDED*. To acquire the lock, a thread tries to atomically set the futex variable to *TAKEN*, using a CAS operation. If the lock is already taken, the thread spins for a certain number of iterations, waiting for the lock to be released. If the lock is not released after the spinning phase, the thread sets the futex variable to *CONTENDED* and executes the futex system call to go to sleep. An already contended lock causes the thread to go to sleep immediately, preventing unnecessary load on the processor. If the thread, currently holding the lock, is done with its operation, it sets the futex variable to *FREE*. Only if the futex variable was previously set to *CONTENDED*, the thread calls the futex system call to wake up the waiting threads, preventing an unnecessary expensive system call in the common case of no contention. The impact of this optimization is discussed in Chapter 6.

Algorithm 5.16 Allocator auxiliary procedures (2)

```
 1: procedure REMOVEBUDDY(buddy : PAGE, order : INT)
 2:     if buddy.next ≠ ⊥ then
 3:         buddy.next.prev ← buddy.prev
 4:         CLWB(buddy.next.prev)
 5:     end if
 6:     if buddy.prev ≠ ⊥ then
 7:         buddy.prev.next ← buddy.next
 8:         CLWB(buddy.prev.next)
 9:     else
10:         FreeArea[order] ← buddy.next
11:         CLWB(FreeArea[order])
12:     end if
13: end procedure

14: procedure INSERTBUDDY(buddy : PAGE, order : INT)
15:     if buddy.next ≠ ⊥ then
16:         buddy.next.prev ← buddy
17:         CLWB(buddy.next.prev)
18:     end if
19:     if buddy.prev ≠ ⊥ then
20:         buddy.prev.next ← buddy
21:         CLWB(buddy.prev.next)
22:     else
23:         FreeArea[order] ← buddy
24:         CLWB(FreeArea[order])
25:     end if
26: end procedure
```

The other platform specific optimization used in our implementation is the use of non-temporal stores to write directly to NVRAM, bypassing the processor cache [Int24]. The x86 architecture provides processor instruction with so-called non-temporal hints. These hints are used to tell the processor that the data being written is not going to be used again soon and should not be stored in the cache hierarchy. The main use case in classical programming (without the use of NVRAM) is to avoid cache pollution, when writing large amounts of data that are not going to be read again soon. When using NVRAM, non-temporal stores have the additional advantage of operating directly in the persistent domain, preventing the need to flush the cache explicitly with CLWB to ensure that the data is written to NVRAM. Additionally, because non-temporal store instructions are not ordered with respect to other memory operations, the CPU is able to better handle the relatively high latency of NVRAM (compared to RAM) by continuing to execute load/store instructions while the data is being written. Our implementation uses non-temporal stores when subsequent loads are not expected, e.g. when writing the current operation to the log structure or when writing the value of a *PUSH* operation into memory.

6 Evaluation

In this chapter, we evaluate the performance of our *flat combining* algorithm in different usage scenarios. We first compare the performance of our implementation to the DFC algorithm presented by Rusanovsky et al. [RAB+21] and the *PBstack* implementation by Fatourou et al. [FKK22b]. The code for these implementations is available under [RAB+20] and [FKK22a], respectively. Both algorithms are similar to our implementation, in that they use software combining to reduce cache coherence traffic and the number of persistent memory operations. The main difference between our algorithm and the ones compared is the guarantee of *detectable recoverability*, in that all three algorithms satisfy *durable linearizability*, but our implementation does not provide *detectability*.

Our experiments were conducted on a machine with an Intel Xeon Gold 6338 processor, running at 2.0 GHz with 32 physical and 64 logical cores. The machine has 64 GiB of RAM and 512 GiB of NVRAM, using the Intel Optane persistent Memory technology. The operating system used was Ubuntu 24.04.2 LTS with Linux kernel 6.8.0-50. The implementation was written in C++ and compiled with g++ 14.2.0 using the `-O2` optimization flag.

The benchmark used to compare our algorithm (PFC) to DFC and *PBstack* is a routine, where threads concurrently push and pop elements to and from the underlying data structure. Each thread executes 10M operations (5M push and 5M pop operations) in 10 consecutive runs and the median time of the runs is used as the result. We compare the performance with varying parallelism, starting from 1 thread up to 128 threads.

As PFC tries to be usable in real-world systems, as a drop-in replacement for `std::vector`, some compromises had to be made, to allow for multiple instances of the data structure to exist in the same process, as well as to allow for publications to be removed from the list, if its thread has not accessed the data structure for a certain amount of time. The first requirement precludes the use of a simple thread-local variable to store the publication, because the C++ standard does not provide a way to declare instance-scoped thread-local variables. Although the Linux pthreads API provides a way to assign a thread-local memory region to an arbitrary key, with the use of the `pthread_key_create` function, the number of keys is limited by the constant `PTHREAD_KEYS_MAX`, which is only guaranteed to be at least 128, which would limit the number of instances usable in one process to `PTHREAD_KEYS_MAX` [Lin24]. The solution, used in our implementation, is to use a thread-local hash map, where the key is an identifier of the instance and the value is the thread's publication associated with that instance. To satisfy the second requirement, the publications are stored in a linked list, to allow for easy removal of publications from the list, while still allowing for fast access to the local publication of a thread. DFC and *PBstack* do not adhere to these requirements, allowing for a more efficient implementation. For comparison purposes, we also evaluate a version of PFC, where those requirements are not enforced, allowing for a simple thread-local variable to store the publication and a contiguous array to store the list of publications. We call this version FPFC.

Additionally, we evaluate the impact of the futex lock mechanism and non-temporal stores, as described in Section 5.4, by comparing the performance to a version of our algorithm, where the waiting threads spin until the lock is released, instead of going to sleep under high contention ($PFC_{no\text{-}yield}$) and a version of our algorithm, where non-temporal stores are replaced with normal stores, followed by PWB instructions ($PFC_{no\text{-}nt}$). The omission of both optimizations is evaluated as well and presented as $PFC_{no\text{-}nt\text{-}yield}$.

Since the *PBstack* and DFC implementations do not provide a swap or get operation, we compare the performance of PFC and FPFC to a naive implementation using a `std::mutex` to synchronize access to our non-volatile vector implementation. This naive implementation is labeled *Mutex* in the following results. The benchmarks for get as well as swap operations were once conducted in a mixed workload scenario, where each thread executes the same number of push, pop and get/swap operations and once in a workload, where each thread executes only get/swap operations.

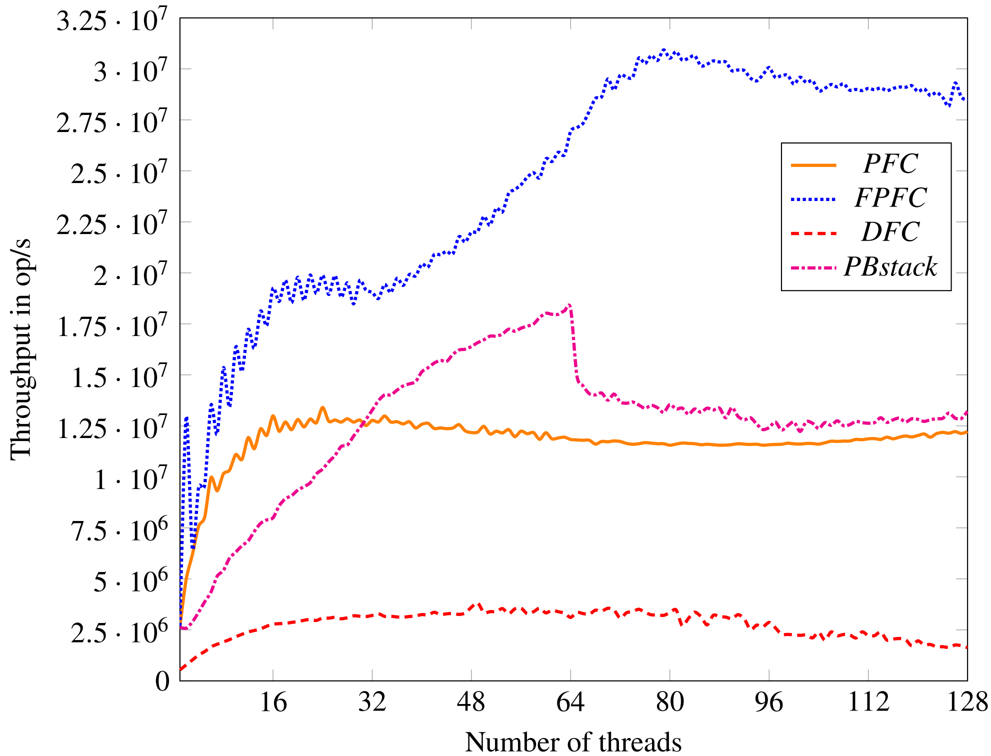

Figure 6.1: *Push-pop* benchmark results for *PFC*, *FPFC*, *DFC*, and *PBstack*

Figure 6.1 shows the results of the benchmark, comparing the performance of PFC and FPFC to DFC and *PBstack*. The X-axis shows the number of threads used in the benchmark, while the Y-axis shows the number of operations per second. The results show that both our algorithms perform better than DFC in all cases, with the performance difference increasing with the number of threads. At 8 threads, DFC achieves a throughput of 1.9M operations per second, while PFC and FPFC achieve 10.1M and 15.3M operations per second respectively. At 32 threads (the number of physical cores of the machine), DFC achieves a throughput of 3.2M operations per second, while PFC reaches 12.8M and FPFC 19.0M operations per second. When the number of threads is increased over the number of logical cores of the machine (64), the performance of DFC decreases. At 128 threads, the maximum number tested, DFC achieves a throughput of 1.6M operations per

second, while PFC and FPFC reach 12.2M and 28.5M operations per second respectively. *PBstack* also performs better than DFC in all tested scenarios, but up to 30 threads, it performs worse than PFC and FPFC. At 8 threads, *PBstack* can execute 5.4M operations per second and at 16 threads, it reaches 8.0M operations per second, compared to 13.0M and 19.2M operations per second for PFC and FPFC. The throughput of *PBstack* continues to increase with the number of threads, up to the number of logical cores of the machine, where it reaches 18.2M operations per second. After that point, the performance decreases abruptly and stabilizes around 13M operations per second. PFC reaches its highest throughput quite early at 24 threads, with 13.4 operations per second, but maintains this throughput up to 128 thread. FPFC scales well, even over the number of logical cores of the machine, where the highest number of operations per second is reached at 79 threads, with 30.9 operations per second. This number decreases only slightly to the aforementioned 28.5M operations per second at 128 threads.

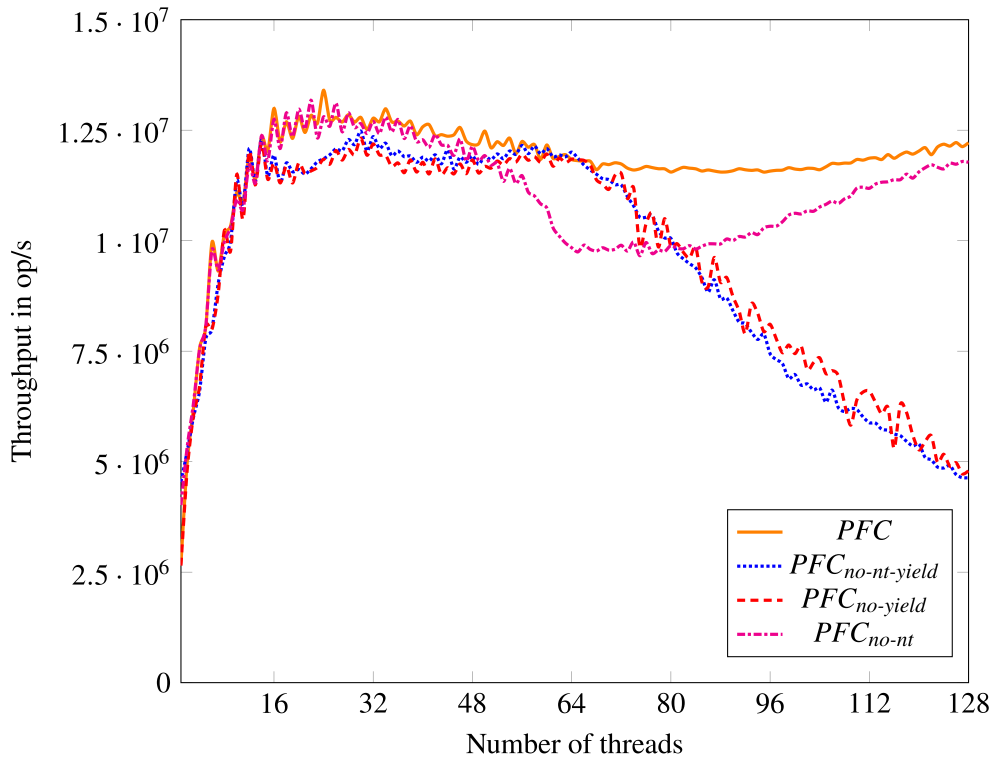

Figure 6.2: *Push-pop* benchmark results for *PFC* with and without futex lock mechanism and non-temporal stores

Figure 6.2 shows the results of the benchmark, comparing the performance of PFC with and without the futex lock mechanism and non-temporal stores. Figure 6.3 shows the same comparison for FPFC. As can be seen in both figures, the performance impact of non-temporal stores is quite small, nevertheless the versions of our algorithm without non-temporal stores perform measurably worse than the optimized versions when the number of threads is higher than the number of logical cores. In both figures, a clear diversion of the curves can be seen, starting around 64 threads. Interestingly, when comparing the performance of the versions with and without the futex lock mechanism, the throughput of PFC is in both cases similar up to 64 threads, but the throughput of FPFC shows a significant difference, from the beginning. Even at 16 threads, the throughput of FPFC without

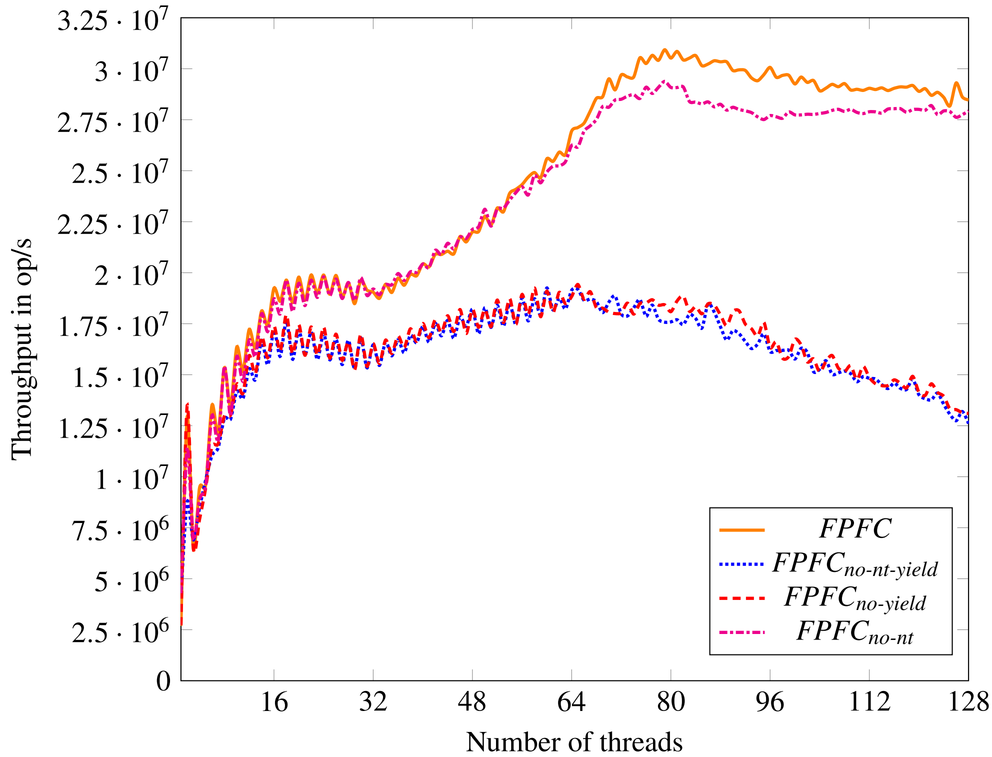

Figure 6.3: *Push-pop* benchmark results for *FPFC* with and without futex lock mechanism and non-temporal stores

the futex lock is, at 17.2M operations per second, about 12% lower than the throughput of the optimized version. At 64 threads, this difference increases to about 43%, where the unoptimized version reaches only 18.8M operations per second, compared to 26.9M in the optimized version. At 128 threads, the throughput of PFC and FPFC without the futex lock mechanism is 4.8M and 13.1M operations per second respectively, making the optimized PFC version about 2.5 times faster and the optimized FPFC version about 2.2 times faster than their counterparts without the futex lock.

The comparison of the performance of swap operations can be seen in Figure 6.4 and Figure 6.5. The results show that swap operations drastically degrade the throughput of our algorithm, compared to only push and pop operations. This behavior is expected, because of at least two reasons. First, no *elimination* is possible on swap operations and second, the execution of multiple combined swap operations can not be atomically persisted, requiring writes to the log structure and additional persist barriers. The benchmarks for get operations are shown in Figure 6.6 and Figure 6.7. The results are very similar to the ones of the *push-pop* benchmarks, shown in Figure 6.1. Interestingly, the highest throughput measured in all benchmarks, is achieved by the naive implementation with a single thread, at 56.9M get operations per second. But even at low parallelism, the naive implementation is outperformed by both PFC and FPFC in all swap and get benchmarks. The results of the swap and get benchmarks show, that the performance of the *flat combining* algorithm is very dependent on the kind of operations performed on the underlying data structure. While read-only operations like get, or operations, where *elimination* is possible, like push and pop, perform very well, but even in scenarios with a lot of such operations, the performance can be seriously degraded by the presence of expensive operations like the implemented swap procedure.

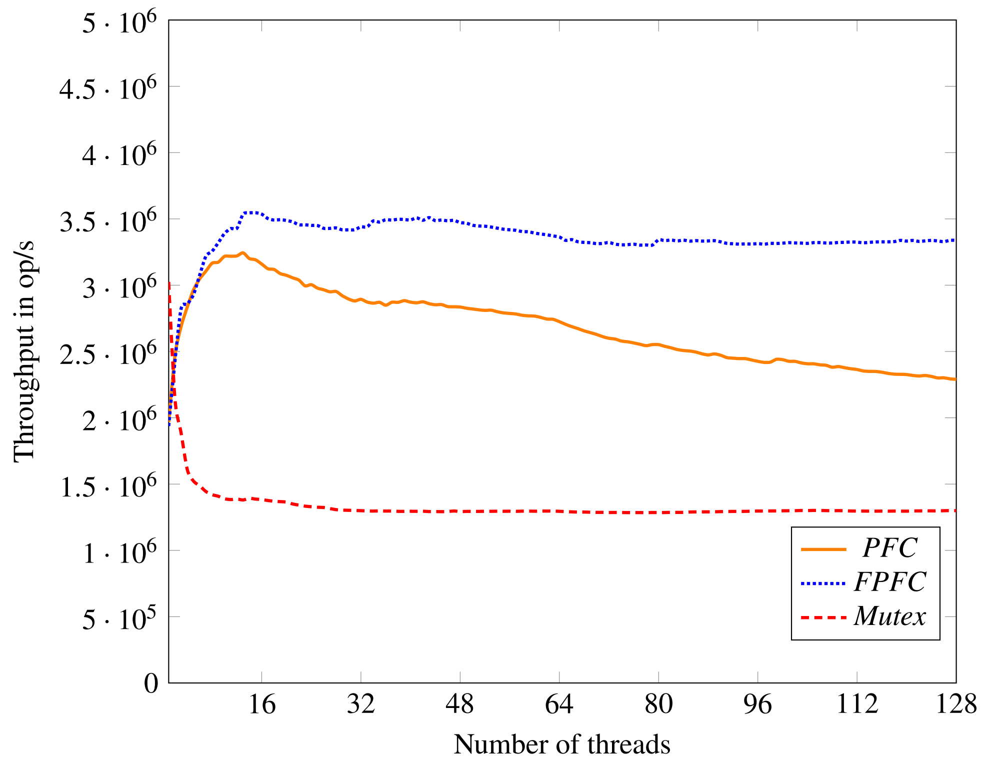

Figure 6.4: Mixed *swap* and *push-pop* benchmark results for *PFC*, *FPFC*, and *Mutex*

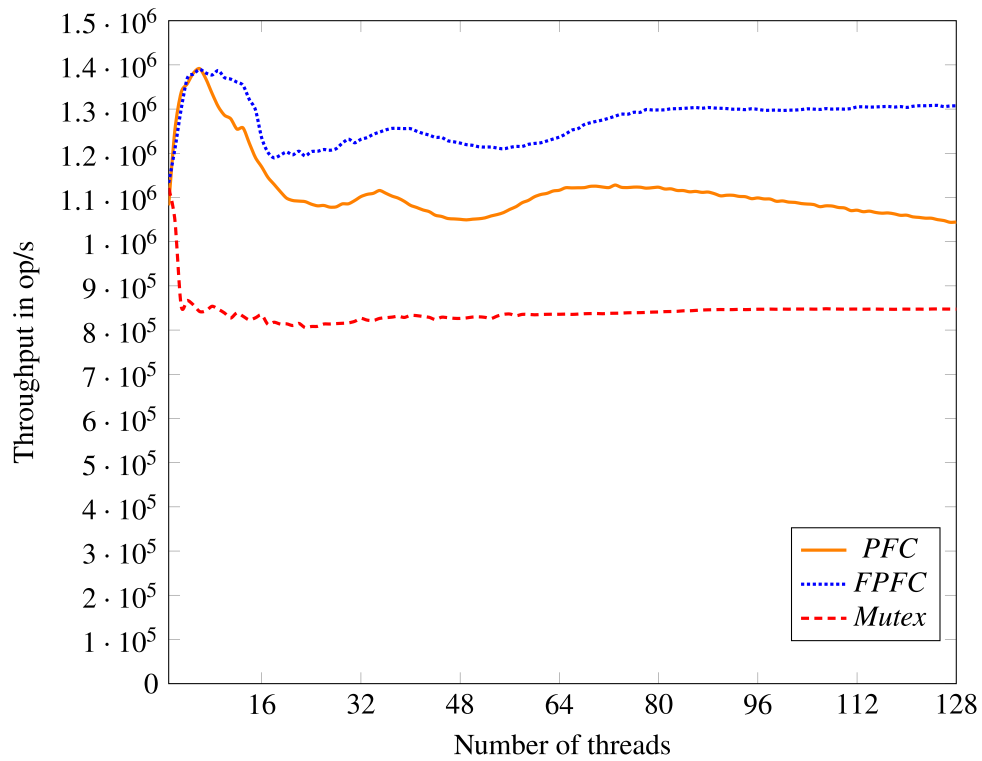

Figure 6.5: Pure *swap* benchmark results for *PFC*, *FPFC*, and *Mutex*

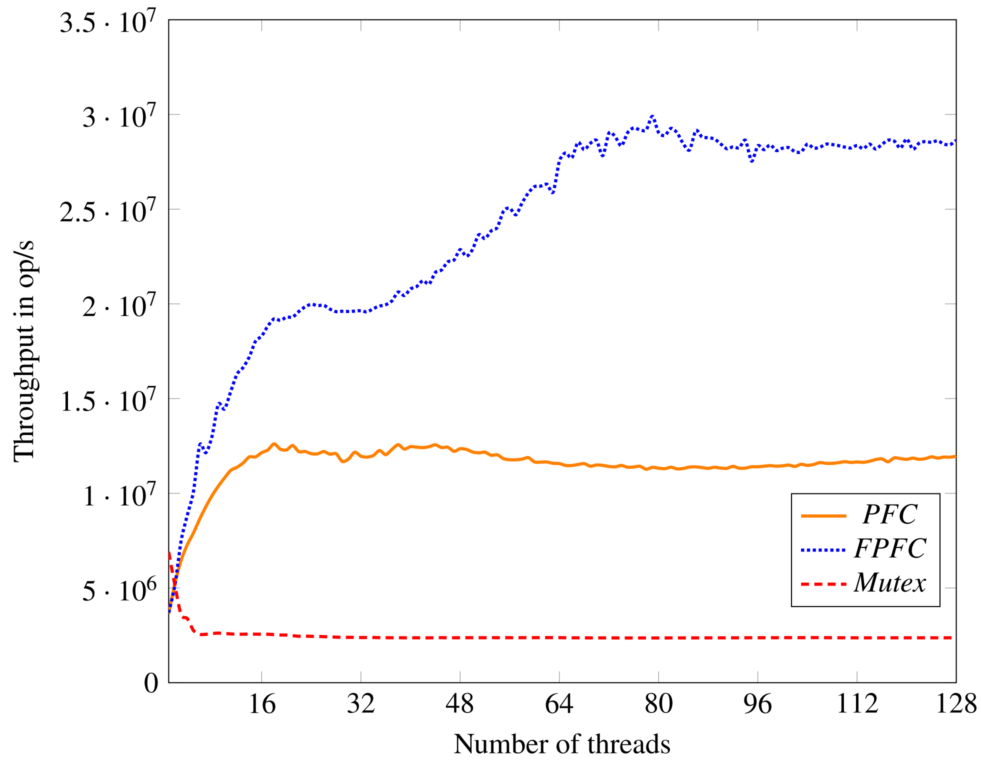

Figure 6.6: Mixed *get* and *push-pop* benchmark results for *PFC*, *FPFC*, and *Mutex*

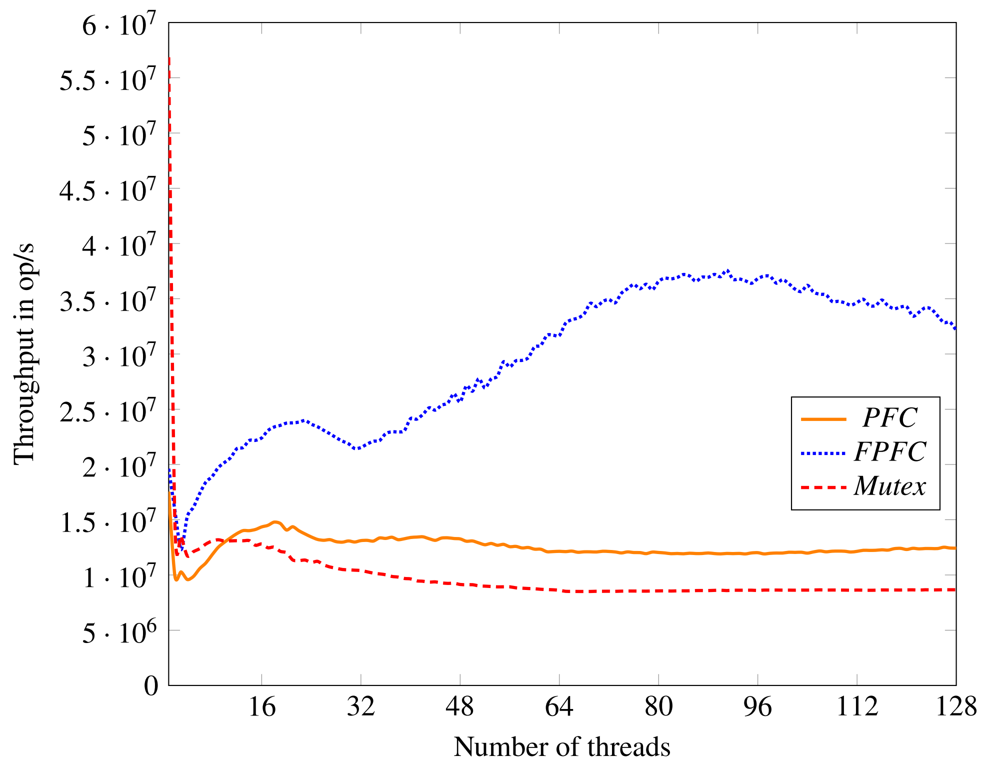

Figure 6.7: Pure *get* benchmark results for *PFC*, *FPFC*, and *Mutex*

7 Conclusion and Outlook

In this work, we present an implementation of a *durable linearizable* resizable array that leverages the *flat combining* algorithm. Our approach, called PFC, addresses the persistent memory challenges by integrating durability into a high-performance concurrent data structure. Through extensive experimentation, we demonstrate that our implementation not only outperforms the naive strategy of using a single lock to protect the entire data structure, but it also shows substantial performance improvements over state-of-the-art software combining methods such as DFC [RAB+21] and *PBstack* [FKK22b]. The experimental evidence indicates that our design scales well under increasing thread counts and effectively exploits the advantages of non-volatile memory systems to ensure both performance and data persistence.

DFC was the first algorithm to provide a non-transactional persistent stack implementation [RAB+21]. Although it achieves good performance compared to other durable algorithms [FKK22b; RAB+21], it is not even close to the performance of the other algorithms, compared in this work. The maximum throughput of DFC is about 3.8 million operations per second, while all other implementations achieve well over 10 million operations per second. *PBstack* is a more recent implementation, that achieves better scalability than DFC, but still falls short of the performance of our implementation, when some requirements are relaxed which abandon the possibility for drop-in replacement of `std::vector` (see Chapter 6).

Although PFC guarantees *durable linearizability*, it does not provide *detectability*, in contrast to DFC and *PBstack*, which provide both guarantees. We conjecture that the absence of *detectability* is not the only reason for the increased performance of our implementation, but also the narrow tailoring of the algorithm to the underlying data structure and therefore possible exploitation of the specific characteristics, that the resizable array provides. Under the model of *explicit epoch persistency* [IMS16], multiple push or pop operations can be executed in one epoch, only requiring a single explicit persist barrier instruction. We also show that some implemented platform-specific optimizations, can have a significant impact on the performance of the algorithm, especially under high parallelism. The proposed futex lock mechanism increases throughput at 128 threads by more than 100% compared to a simple spin lock.

Since the presented *flat combining* algorithm is a general-purpose algorithm, not limited to resizable arrays, it can be used to implement other data structures, and we believe that the performance improvements shown in this work can be (partially) transferred to other data structures as well. Especially data structures providing the possibility for *elimination* [RAB+21] of operations, and/or an efficient way to execute multiple operations with reduced persistence overhead, could massively benefit from the proposed *flat combining* algorithm.

Looking ahead, there are several exciting avenues for future work. One promising direction is the mentioned transfer of the PFC algorithm to other data structures. An experimental performance evaluation of such implementations could (in-)validate our conjecture above. Another interesting path is the exploration of possibilities to adapt our algorithm to guarantee *detectability*, which

would bring our implementation in line with DFC and *PBstack*, in terms of durability guarantees. Examining the performance of such an implementation could check our other presumption, that the absence of *detectability* is not the only reason for the superior performance of our algorithm, especially compared to DFC.

Bibliography

[ABF+22] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, E. Kosmas. “Detectable recovery of lock-free data structures”. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 262–277. ISBN: 9781450392044. DOI: 10.1145/3503221.3508444 (cit. on p. 19).

[Aue52] I. L. Auerbach. “A static magnetic memory system for the ENIAC”. In: *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*. ACM ’52. Pittsburgh, Pennsylvania: Association for Computing Machinery, 1952, pp. 213–222. ISBN: 9781450373623. DOI: 10.1145/609784.609813 (cit. on p. 17).

[BGT16] R. Berryhill, W. Golab, M. Tripunitara. “Robust Shared Objects for Non-Volatile Main Memory”. In: *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Ed. by E. Anceaume, C. Cachin, M. Potop-Butucaru. Vol. 46. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 20:1–20:17. ISBN: 978-3-939897-98-9. DOI: 10.4230/LIPIcs.OPODIS.2015.20 (cit. on p. 18).

[BHR20] O. Ben-Baruch, D. Hendler, M. Rusanovsky. “Upper and Lower Bounds on the Space Complexity of Detectable Objects”. In: *Proceedings of the 39th Symposium on Principles of Distributed Computing*. PODC ’20. Virtual Event, Italy: Association for Computing Machinery, 2020, pp. 11–20. ISBN: 9781450375825. DOI: 10.1145/3382734.3405725 (cit. on pp. 19, 25).

[CCA+11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, S. Swanson. “NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 105–118. ISBN: 9781450302661. DOI: 10.1145/1950365.1950380 (cit. on p. 23).

[CJRK23] K. Cho, S. Jeon, A. Raad, J. Kang. “Memento: A Framework for Detectable Recoverability in Persistent Memory”. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591232 (cit. on p. 19).

[CNF+09] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, D. Coetzee. “Better I/O through byte-addressable, persistent memory”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 133–146. ISBN: 9781605587523. DOI: 10.1145/1629575.1629589 (cit. on p. 18).

[DMS11] D. Dice, V. J. Marathe, N. Shavit. “Flat-combining NUMA locks”. In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 65–74. ISBN: 9781450307437. DOI: 10.1145/1989493.1989502 (cit. on pp. 21, 22).

[FHMP18] M. Friedman, M. Herlihy, V. Marathe, E. Petrank. “A persistent lock-free queue for non-volatile memory”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. Vienna, Austria: Association for Computing Machinery, 2018, pp. 28–40. ISBN: 9781450349826. DOI: 10.1145/3178487.3178490 (cit. on pp. 19, 23).

[FK11] P. Fatourou, N. D. Kallimanis. “A highly-efficient wait-free universal construction”. In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 325–334. ISBN: 9781450307437. DOI: 10.1145/1989493.1989549 (cit. on pp. 22, 23).

[FK12] P. Fatourou, N. D. Kallimanis. “Revisiting the combining synchronization technique”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’12. New Orleans, Louisiana, USA: Association for Computing Machinery, 2012, pp. 257–266. ISBN: 9781450311601. DOI: 10.1145/2145816.2145849 (cit. on pp. 22, 23).

[FKK22a] P. Fatourou, N. D. Kallimanis, E. Kosmas. *Reference Code for PBstack*. 2022. URL: https://github.com/ConcurrentDistributedLab/PersistentCombining/tree/eaba017fc655ca24d040a54a56ce1e2bcf428497 (cit. on p. 47).

[FKK22b] P. Fatourou, N. D. Kallimanis, E. Kosmas. “The performance power of software combining in persistence”. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 337–352. ISBN: 9781450392044. DOI: 10.1145/3503221.3508426 (cit. on pp. 16, 19, 20, 23, 25–27, 47, 53).

[FRK02] H. Franke, R. Russell, M. Kirkwood. “Fuss, futexes and furwocks: Fast userlevel locking in linux”. In: *AUUG Conference Proceedings*. Vol. 85. AUUG, Inc. 2002, pp. 479–495 (cit. on p. 43).

[GH13] M. Gorelik, D. Hendler. “Brief announcement: an asymmetric flat-combining based queue algorithm”. In: *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*. PODC ’13. Montréal, Québec, Canada: Association for Computing Machinery, 2013, pp. 319–321. ISBN: 9781450320658. DOI: 10.1145/2484239.2484279 (cit. on p. 22).

[GKR16] M. Galimullin, E. Kalishenko, N. Rapotkin. “Performance analysis of thread synchronization strategies in concurrent data structures based on flat-combining”. In: *2016 18th Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT)*. Apr. 2016, pp. 54–59. DOI: 10.1109/FRUCT-ISPIT.2016.7561508 (cit. on p. 21).

[GL04] R. Guerraoui, R. Levy. “Robust emulations of shared memory in a crash-recovery model”. In: *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* 2004, pp. 400–407. DOI: 10.1109/ICDCS.2004.1281605 (cit. on p. 18).

[HIST10a] D. Hendler, I. Incze, N. Shavit, M. Tzafrir. "Flat combining and the synchronization-parallelism tradeoff". In: *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '10. Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 355–364. ISBN: 9781450300797. DOI: 10.1145/1810479.1810540 (cit. on pp. 15, 19–22, 27).

[HIST10b] D. Hendler, I. Incze, N. Shavit, M. Tzafrir. "Scalable Flat-Combining Based Synchronous Queues". In: *Distributed Computing*. Ed. by N. A. Lynch, A. A. Shvartsman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 79–93. ISBN: 978-3-642-15763-9. DOI: 10.1007/978-3-642-15763-9_8 (cit. on p. 21).

[HSY04] D. Hendler, N. Shavit, L. Yerushalmi. "A scalable lock-free stack algorithm". In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '04. Barcelona, Spain: Association for Computing Machinery, 2004, pp. 206–215. ISBN: 1581138407. DOI: 10.1145/1007912.1007944 (cit. on p. 27).

[HW87] M. P. Herlihy, J. M. Wing. "Axioms for concurrent objects". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 13–26. ISBN: 0897912152. DOI: 10.1145/41625.41627 (cit. on p. 18).

[IMS16] J. Izraelevitz, H. Mendes, M. L. Scott. "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model". In: *Distributed Computing*. Ed. by C. Gavoille, D. Ilcinkas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327. ISBN: 978-3-662-53426-7. DOI: 10.1007/978-3-662-53426-7_23 (cit. on pp. 18, 53).

[Int24] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2024. URL: https://www.intel.com/content/www/us/en/content-details/843820 (cit. on pp. 18, 45).

[Kno65] K. C. Knowlton. "A fast storage allocator". In: *Commun. ACM* 8.10 (Oct. 1965), pp. 623–624. ISSN: 0001-0782. DOI: 10.1145/365628.365655 (cit. on pp. 25, 36).

[Lin24] Linux Kernel Organization. *pthread_key_create(3) - Linux man page*. 2024. URL: https://linux.die.net/man/3/pthread_key_create (cit. on pp. 25, 47).

[MI85] F. Masuoka, H. Iizuka. "Semiconductor memory device and method for manufacturing the same". US4531203A. Toshiba Corp. 1985 (cit. on p. 17).

[MLH94] P. Magnusson, A. Landin, E. Hagersten. "Queue locks on cache coherent multiprocessors". In: *Proceedings of 8th International Parallel Processing Symposium*. 1994, pp. 165–171. DOI: 10.1109/IPPS.1994.288305 (cit. on p. 22).

[MS91] J. M. Mellor-Crummey, M. L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65. ISSN: 0734-2071. DOI: 10.1145/103727.103729 (cit. on pp. 22, 23).

[OTY99] Y. Oyama, K. Taura, A. Yonezawa. "Executing parallel programs with synchronization bottlenecks efficiently". In: *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*. Vol. 16. Citeseer. 1999, p. 95 (cit. on pp. 19–21).

[PCW14] S. Pelley, P. M. Chen, T. F. Wenisch. “Memory persistency”. In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 265–276. ISSN: 0163-5964. DOI: 10.1145/2678373.2665712 (cit. on p. 18).

[RAB+20] M. Rusanovsky, H. Attiya, O. Ben-Baruch, T. Gerby, D. Hendler, P. Ramalhete. *Reference Code for DFC*. 2020. URL: https://github.com/matanr/detectable_flat_combining/tree/c71c724374b5ebf6d56f343e789dc3afa5ade3de (cit. on p. 47).

[RAB+21] M. Rusanovsky, H. Attiya, O. Ben-Baruch, T. Gerby, D. Hendler, P. Ramalhete. “Flat-Combining-Based Persistent Data Structures for Non-volatile Memory”. In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by C. Johnen, E. M. Schiller, S. Schmid. Cham: Springer International Publishing, 2021, pp. 505–509. ISBN: 978-3-030-91081-5. DOI: 10.1007/978-3-030-91081-5_38 (cit. on pp. 16, 20, 23, 25–27, 47, 53).

[VTS11] H. Volos, A. J. Tack, M. M. Swift. “Mnemosyne: lightweight persistent memory”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 91–104. ISBN: 9781450302661. DOI: 10.1145/1950365.1950379 (cit. on p. 23).

[YT+87] P.-C. Yew, N.-F. Tzeng, et al. “Distributing hot-spot addressing in large-scale multiprocessors”. In: *IEEE transactions on Computers* 100.4 (1987), pp. 388–395. DOI: https://doi.org/10.1109/TC.1987.1676921 (cit. on pp. 19, 21).

All links were last followed on March 11, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature