PREP-UC: A Practical Replicated Persistent Universal Construction

Gaetano C. Coccimiglio
gccoccim@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Trevor A. Brown
trevor.brown@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Srivatsan Ravi
srivatsr@usc.edu
University of Southern California
Los Angeles, California, USA

ABSTRACT

The process of designing and implementing correct concurrent data structures is non-trivial and often error prone. The recent commercial availability of non-volatile memory has prompted many researchers to also consider designing concurrent data structures that persist shared state allowing the data structure to be recovered following a power failure. These so called persistent concurrent data structures further complicate the process of achieving correct and efficient implementations. Universal constructions (UCs) which produce a concurrent object given a sequential object, have been studied extensively in the space of volatile shared memory as a means of more easily implementing correct concurrent data structures. In contrast, there are only a handful of persistent universal constructions (PUCs) which beyond producing a concurrent object from a sequential object, guarantees that the object can be recovered following a crash. Existing PUCs satisfy the correctness condition of durable linearizability which requires that operations are persisted before they complete. Satisfying the weaker correctness condition of buffered durable linearizability allows for improved performance at the cost of failing to recover some completed operations following a crash. In this work we design and implement both a buffered durable linearizable and a durable linearizable PUC based on the node replication UC. We demonstrate that we can achieve significantly better performance satisfying buffered durable linearizability while also restricting the maximum number of operations that can be lost after a crash.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent algorithms**.

KEYWORDS

Universal construction, Durable linearizability, Buffered durable linearizability, Non-volatile Memory

ACM Reference Format:
Gaetano C. Coccimiglio, Trevor A. Brown, and Srivatsan Ravi. 2022. PREP-UC: A Practical Replicated Persistent Universal Construction. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490148.3538568

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07…$15.00
https://doi.org/10.1145/3490148.3538568

1 INTRODUCTION

Recently, byte-addressable Non-Volatile Memory (NVM) has become commercially available. This has prompted many researchers to explore designing persistent concurrent data structures for use with NVM [14, 18–20, 34, 36]. Unfortunately, designing efficient persistent concurrent data structures has proven to be a challenge. The main reason for this difficulty is the fact that, currently, NVM exists only as a portion of main memory while processor caches, registers and DRAM remain volatile. This work considers system crashes caused by power failures. If a system crash occurs, only data in NVM can be recovered. To guarantee that data is written back to NVM the programmer must explicitly force cached data to be written back to main memory at key times. This requires the use of *flush* (also known as *write-back*) instructions as well as expensive *persistent fence* instructions.

Much of the recent literature has focused on adding persistence to volatile data structures [7, 11, 12, 14, 18–20, 34, 36]. These works have proposed hand-crafted approaches and general transformations, both of which are concerned with precisely where flush and fence instructions should be used to achieve persistence. Starting from volatile data structures is logical given that volatile concurrent data structures have been studied extensively [23]. However, designing volatile concurrent data structures is itself a difficult endeavour, and adding persistence only increases complexity. By comparison, sequential data structures are relatively straightforward.

A universal construction (UC) is a generic mechanism which takes a sequential object as input and produces a concurrent object. A persistent universal construction (PUC) is a universal construction which also ensures that the object can be recovered following a system crash. Using a PUC we can easily obtain persistent concurrent data structures by starting from a volatile *sequential* data structure.

Crucially, UCs do not modify the underlying sequential implementation and instead view it as a black box. In the context of PUCs this is important because it means that we cannot add flush or fence instructions between the reads and writes performed by the sequential implementation. There is a wealth of prior research regarding UCs [4, 13, 16, 17, 22, 24]. By comparison there is relatively little prior work studying PUCs [7, 12]. Existing PUCs have focused on achieving correctness but lack performance and scalability [12].

Volatile concurrent data structures typically satisfy the correctness condition of *linearizability* [25]. However, linearizability has no notion of crashes or recovery so it cannot be directly applied to persistent concurrent data structures. Izraelevitz et al. proposed

durable linearizability and *buffered durable linearizability* for defining correctness of persistent concurrent objects for failure models in which all threads fail simultaneously (typically referred to as a full-system crash) [26]. This work considers the same failure model which reflects the reality of crashes caused by power failures. Informally, an object satisfies durable linearizability if following a crash, the object state reflects a consistent operation subhistory that includes all operations that completed prior to the crash. In other words, durable linearizability allows at most n operations to be *lost* when a crash occurs for a system with n threads (specifically, a subset of the operations in progress when the crash occurs).

An object that satisfies durable linearizability also satisfies the weaker buffered durable linearizability. Intuitively, buffered durable linearizability requires that following a crash, the object state reflects a *prefix* of the operations that completed before the crash. This means that a buffered durable implementation does not necessarily bound the number of operations that can be lost as a result of a crash. Recent work has favored durable linearizability as the correctness condition for persistent concurrent data structures [7, 11, 12, 14, 18–20, 34, 36]. Relatively few papers have explored the question of whether the relaxed constraints of buffered durable linearizability can be exploited to improve performance (e.g., [30, 35]).

Contributions. In this work we present PREP-UC, a PUC based on the Node Replication algorithm of Calciu et al. [4]. PREP-UC utilizes logging along with two persistence-only replicas of the underlying sequential object which are periodically persisted to ensure that the data structure can be recovered following a system crash. We provide both durable linearizable and buffered durable linearizable implementations of PREP-UC. The node replication UC is specifically designed for non-uniform memory access (NUMA) systems. PREP-UC preserves this NUMA-awareness while guaranteeing persistence. We perform an empirical evaluation of our implementations, comparing them to CX-PUC of Correia et al. [12]. We demonstrate that a PUC can achieve significantly better performance with a buffered durable linearizable implementation while still guaranteeing an upper bound on the number of operations that can be lost in the event of a crash.

2 BACKGROUND

First, we review the definitions and properties of durable linearizability and buffered durable linearizability. We then describe some practical details relating to the current implementation of NVM in real hardware. We also present an overview of related work.

2.1 Durable vs. Buffered Durable Linearizability

Durable linearizability essentially defines how to apply linearizability for models where full system crashes are possible. Intuitively it requires that any operation that completed before a crash will be reflected in the state of the object after recovery. This also requires that operations are made persistent at some point before the operation completes. Durable linearizability is a local property [26]. One consequence of this is that durable linearizable objects can be composed.

Buffered durable linearizability provides weaker guarantees. Following a system crash, it requires that the state of the object reflects some prefix of the operations that completed prior to the crash. Buffered durable linearizability does not require operations to be persisted before they complete and is not a local property as a result.

2.2 Persistence in Practice

Currently NVM exists as part of main memory while the processor cache, registers and DRAM are still volatile. Whenever we perform a write, it first takes effect in the volatile cache. Flush and fence instructions are required to force data to be written back from the cache to NVM. The processor may arbitrarily flush data to NVM (without the programmer's knowledge) as a result of the cache coherence protocol. When this occurs we refer to it as a *background flush*. Programmers are forced to utilize explicit flush and fence instructions because they cannot rely on background flushes to ensure that data is written back to NVM. On Intel platforms fences correspond to the SFENCE instruction and there are four instructions that flush data from the cache; The CLFLUSH, CLFLUSHOPT, and CLWB instructions each force a single cache line to be written back to NVM. The CLFLUSH instruction blocks until the flush is completed. CLFLUSHOPT and CLWB are asynchronous and must be followed by an SFENCE to block until the flush is completed. The WBINVD instruction is a privileged instruction that invalidates and writes back the *entire cache* (at all levels) of the processor that executes it. WBVIND can be executed in user space with the help of a kernel module and system call.

2.3 Related Work

PUCs. Cohen et al. presented the Order Now Linearize Later (ONLL) PUC [7]. ONLL is lock-free. It relies on a shared global lock-free volatile queue and per thread persistent logs. The global queue represents the state of the underlying object in the form of the linearization order of all update operations that have ever been applied on the object. Each entry in one of the persistent logs stores up to n operations (including their arguments) and a unique index representing the linearization order of the first operation in the entry. A notable quality of ONLL is that it produces an implementation in which read-only operations do not flush or fence. An update operation is added to the global queue before it is added to the thread's persistent log along with any previous operations that are not yet guaranteed to be persistent.

Correia et al. presented CX-PUC [12] which is based on CX-UC by the same authors [13]. CX-UC utilizes $2n$ replicas of the underlying sequential data structure and maintains a pointer to the most up-to-date replica. Update operations are added to a shared global queue which establishes the linearization order of update operations. Once an update is added to the queue the thread that invoked the update will lock one of the $2n$ replicas. The thread then brings the replica up-to-date by applying all update operations up to and including the operation that it invoked. After applying the update the thread will attempt to declare the current replica as the most up-to-date replica via a CAS. At any point in time, only one thread can have write access to a replica but many threads can have read access. Concurrent access to a replica is managed by a strong try reader-writer lock. To guarantee durable linearizability CX-PUC must persist all $2n$ replicas since it must recover the most up-to-date replica which can be any of the $2n$ replicas. Moreover, the entire

replica must be persisted after applying a single update operation which is very expensive. CX-PUC requires that the underlying sequential data structure utilizes a persistent allocator provided by the PUC which often requires modifications to the sequential implementation.

Transforms. Some researchers have proposed transforms or recipes which provide rules and mechanisms for more adding persistence to existing objects. The most well known of these is the transform described by Izraelevitz et al. [26]. This transform was designed for a release consistency memory model and requires adding a significant number of fence instructions; one fence per store-release and load-acquire and two fences per CAS. This amount of fencing is known to be excessive. Similarly to ONLL, PRONTO from Memaripour et al. utilizes a form of operation logging for update operations to achieve durable linearizable objects [29]. Unlike ONLL, PRONTO uses a background thread to asynchronously log an update operation and its arguments which doubles the maximum number of concurrent threads. The Mirror transform from Friedman et al. keeps a volatile replica of every persistent variable and uses specialized implementations for reads, writes, CASs and fetch-and-adds [20]. The FliT transform from Wei et al. relies on a custom *persist* and associates a counter with every persist variable which is used to decide when to flush and fence [34]. The custom type is used to instrument reads and writes. FliT provides durable linearizability. Montange relies a custom persistent memory allocator and an epoch based approach to achieve buffered durable linearizability [35]. Wang et al. presented NAP, an approach that converts an existing concurrent persistent memory index into a NUMA aware persistent memory index [32]. NAP primarily focuses on providing efficient accesses to items when the access pattern is Zipfian.

Transactional Memory. Unlike PUCs persistent transactional memory (PTM) approaches must interpose reads and writes. There are many existing PTMs: NV-heaps [5], RedoPTM, CX-PTM [12], Mnemosyne [31], Romulus [11], DudeTM [28], DHTM [27], PHyTM [2] and PMDK [10]. All of these works are software TMs except for PHyTM which is a hybrid TM. Similar to the PUCs that we previously described, these PTMs rely on techniques such as logging (both undo and redo logging), data replication and custom memory management mechanisms/allocators. In general, these PTMs typically perform better compared to PUCs however TMs require compiler support or require the programmer to instrument reads and writes, which is a significant amount of work.

3 NODE REPLICATION

PREP-UC is based on the node replication universal construction (NR-UC) of Calciu et al. [4]. In this section we provide an overview of node replication. NR-UC minimizes cross socket memory accesses and communication for NUMA systems by replicating the underlying sequential data structure on each NUMA node. As a result, a system with N NUMA nodes requires N replicas. Each replica is protected by a trylock and a reader-writer lock.

Across NUMA nodes, threads communicate via a shared global log. The log is implemented as a circular buffer. An index called *logTail* tracks the next available empty entry in the log. The log stores update operations and their respective arguments which must be applied to every replica. Each log entry also stores a flag that we refer to as the emptyBit which denotes whether the entry is full. The emptyBit of a log entry is flipped after a new operation and its arguments are written to the log entry. In an infinite log an empty log entry always has an emptyBit equal to zero. For a finite log, the first time that the log wraps around the meaning of the emptyBit changes so that 1 means empty and 0 means full. Each time the log wraps around the parity of the emptyBit's meaning flips. The emptyBit allows log entries to be reused and ensures that a thread will never execute an update operation with incomplete or stale arguments.

Index Name	Scope	Meaning
localTail	Per Replica	Last update applied to the local replica
completedTail	Global	Last update applied to any replica
logTail	Global	Last log entry

Table 1: Summary of the different indexes used in NR-UC

Each replica has a *localTail* index which points to the log entry up to which the replica has been updated. A thread can reserve log entries by atomically updating the logTail index via a compare-and-swap (CAS). At any point during an execution at most one thread per replica will contend on updating the logTail (not necessarily the same thread each time). The node replication UC also keeps track of the *completedTail* index which indicates the log entry after which the entries have not yet been applied to any replica. Note that while the log itself is finite, the various index variables (logTail, localTail, and completedTail) are all monotonically increasing. When the log wraps we get the log entry corresponding to the particular index, by computing the value index mod the log size.

Within a single NUMA node threads coordinate using flat combining, a technique that was originally presented in [21]. Flat combining relies on batching update operations. A batch consists of update operations invoked by threads on the NUMA node corresponding to the local replica. Each thread on the NUMA node has its own slot in the batch. The replica also contains an array of responses for each operation in the batch. These responses are initially empty. Each thread on the node can add an update operation to a local batch. A single thread known as the *combiner* is responsible for adding the operations in the local batch to the global log and applying the operations in the batch to the local replica. A thread becomes the *combiner* by claiming the trylock of the local replica. For this reason, the trylock is referred to as the *combiner lock*.

When a thread wants to perform an update operation it will first add its operation to the batch local to the replica corresponding to the NUMA node on which the thread is running. Next the thread will attempt to claim the combiner lock. If the thread successfully installs itself as the combiner it will then reserve enough log entries to append all operations in the batch to the log by modifying the logTail. After updating the logTail, the combiner will write its batch into the log. Next the combiner will claim the reader-writer lock in write mode. The combiner will then bring the local replica up-to-date by performing any pending operations up to the previous logTail, and update the localTail of the replica to the new logTail.

Next the combiner will attempt to update the completedTail index to the localTail of the replica via CAS. The combiner then applies all of the updates in its batch to the local replica and updates the responses for the completed operations in the batch. Finally the combiner will release the reader-writer lock and the combiner lock. If a thread failed to claim the combiner lock it will be blocked until the combiner lock is released or the response of the update operation invoked by the thread is updated to a non-empty value. Threads executing read-only operations claim the reader-writer lock in read mode and are blocked until the localTail of the replica is greater than or equal to the completedTail. This is necessary to ensure that the response of the read-only operation reflects the effects of all completed update operations.

A user interacts with NR-UC via a procedure called *ExecuteConcurrent* which takes, as its arguments, the name of an operation and arguments to be passed to that operation. *ExecuteConcurrent* performs the operation on the underlying sequential data structure and returns the result of completing that operation. The user must also provide a mechanism for *ExecuteConcurrent* to determine if an operation is read-only.

In practice it is typical for threads to be bound to unique processors such that there is a fixed mapping of threads to replicas. One could also imagine a mechanism wherein threads could detect the NUMA node that the thread is currently running on rather than pinning threads to specific processors. Calciu et al. utilized the former approach of binding threads to specific processors.

4 PREP-UC: DESIGN DECISIONS

In this section we provide an overview of PREP-UC, which is based on the node replication UC (NR-UC) of Calciu et al. [4].

4.1 Extending Node Replication for Persistence

NR-UC has two main design features that make it a good candidate for extension into a PUC. First, NR-UC already maintains a *log* of all update operations and their arguments. Moreover, the use of batching and flat combining reduces overhead and contention on the log which is beneficial if we want to persist the log. Second, we know how up-to-date each of the N replicas are based on their localTails which allows us to think of any single replica as a snapshot of the data structure.

Storing the log in persistent memory is a natural first step towards achieving persistence, however, unless we allow for an infinite log (and, correspondingly, accept that we will need to invoke unboundedly many operations to recover after a crash), it is not sufficient to persist *only* the log. To limit the size of the persistent log, one might persist a replica, which could then serve as a sort of checkpoint from which the log can begin. Of course, extending NR-UC into a correct and efficient PUC is not as simple as just storing the log and a replica in persistent memory. We will now discuss three key ideas of PREP-UC. In particular we will describe how we go about persisting the operation log, which replicas we persist, and how we persist consistent replicas *without instrumenting* reads, writes and memory (de)allocations.

Operation Log. With regards to persisting the log, we cannot assume that a log entry fits into a single cache line. Moreover, we might require multiple writes to store the operation and its arguments into a log entry. For this reason we must ensure that we cannot recover an empty or partially full log entry following a crash.

Recall that NR-UC utilizes a Boolean flag (which we call the emptyBit) to indicate whether a log entry is full or not. This flag is updated after the operation and its arguments are written into the log entry. When adding new entries to the log we first write the arguments of the operation followed by the operation in the form of a function pointer. To minimize the number of fences required to persist this data, when a combiner thread appends a batch of updates to the log, it will first write the arguments of every update in the batch followed by the operations, asynchronously flushing each of the modified cache lines. After all operations and their arguments are written into the log a single fence is executed. Next, for each entry that the combiner wrote, it will set the emptyBit to the appropriate value indicating that the log entry is full.

Since we also plan to persist one or more replicas, following a crash, we need to know which log entries must be applied to bring the recovered replicas back up-to-date. For a particular replica the relevant entries are the ones between the localTail of the replica and the completedTail. For this reason we also persist the completedTail index.

Persistent Replicas. In order to ensure that the shared operation log remains finite, we must persist at least one of the N replicas. At any point in time, any one of the N replicas could be the most up-to-date replica. One could imagine a design that persists all N replicas, but doing so would incur significant overhead. For this reason we would like to persist the fewest number of replicas required to guarantee that we can recover the data structure following a crash.

Persisting only one of the N replicas is not sufficient to guarantee correctness. This is because during an update operation, a replica could enter an inconsistent state which could then be written back to NVM as a result of a background flush. For example, suppose that the underlying sequential object is a set implemented as a search tree where the first step of an update operation is to save a copy of the root pointer then set the root pointer to null. Now suppose that a background flush occurs writing the null root pointer to NVM. If a crash occurs at this point the entire data structure is lost since we would recover only the null root pointer. This is unavoidable since a PUC utilizes the underlying sequential object like black box which means that a PUC cannot flush and fence during an update because it does not know how the sequential object performs updates. To prevent this problem we introduce two dedicated *persistent replicas* and we ensure that only one of the two replicas is being updated at any time. We keep track of which of the two replicas was being modified such that following a crash we know which of the persistent replicas was definitely not being modified.

The reader might wonder why we introduce dedicated persistence replicas rather than simply persisting two of the existing replicas. In order to ensure that only one of the persistent replicas is updated at a time, whenever a thread tries to update one of the persistent replicas we would need to block updates to the other persistent replica, potentially preventing *all threads* on the corresponding NUMA node from making progress. Moreover, since read-only operations require a replica to be up to date, read-only operations could also be blocked.

To maintain the dedicated persistent replicas, we spawn an additional *persistence thread*. Similarly to threads in NR-UC, we bind persistence thread to a unique processor. The use of a dedicated persistence thread ensures that we have complete control over when the persistent replicas are updated *without* blocking operations on an entire NUMA node. The persistence thread updates the two persistent replicas in cycles such that only one of the two replicas receives updates during a single update cycle. The persistent replica that receives updates during the current cycle is designated the *active persistent replica* and the other is designated the *stable persistent replica*. The stable persistent replica remains in a quiescent and consistent state in NVM while the active persistent replica is being updated.

We periodically persist the active persistent replica in order to persist a new *checkpoint* and swap the active and stable replicas. This occurs when the log is *close* to being full. In the (hopefully rare) case that the log becomes completely full before the persistence thread can bring the active replica up-to-date, the volatile replicas are prevented from adding new log entries until the active replica is brought up-to-date. Otherwise we do not block updates to the volatile replicas.

As in NR-UC, each worker thread (any thread other than the persistence thread) only accesses one of the N volatile replicas. The worker threads operating on the volatile replicas do not need to access the persistent replicas, but they do need to know about the localTails of the two persistent replicas in order to correctly reuse log entries when the log (a circular buffer) wraps around.

Flushing Persistent Replicas. At the end of an update cycle the persistence thread must write the active persistent replica back to NVM. The main challenge arises from the fact that a PUC cannot interpose reads and writes (because it cannot modify the code for the sequential object). Existing research has addressed the problem of persisting data without the ability to interpose reads and writes through the use of custom memory management mechanisms. Crucially, these approaches rely on isolating the replicas in specific regions of memory of some predetermined size. CX-PUC utilized a custom allocator so they could track the locations and sizes of memory regions containing the persistent replicas. They flush an entire replica each time it changes. PRONTO used custom page-fault handlers to allow flushing individual pages instead of an entire replica. When we want to persist a replica we must flush all of its addresses that have changed since they were last flushed. If we could instrument writes then we could track what addresses were modified and only flush those. Since we cannot instrument writes, we can flush all addresses like CX-PUC, flush pages like PRONTO or flush the cache. We choose to flush the cache since it contains all of the persistent replicas' modified addresses. There is a trade off, if the data structure is very small and fits in the cache then flushing the cache is wasteful but when the data structure is large it is better to flush the cache. To flush the cache we utilize the same approach as Cohen et al. [6] which relies on the WBINVD instruction. The WBINVD instruction invalidates the entire cache hierarchy of the executing processor and writes modified data back to main memory. Processors on the same node as the executing processor are blocked until the write-back and invalidate operation is completed. We refer the reader to 8.7.13.1 of [8] for further details regarding the WBINVD instruction. Since only one thread accesses the persistent replicas we only need to execute the WBINVD instruction on one processor.

PREP-UC Interface. Users interact with PREP-UC utilizing the same interface as NR-UC, specifically via a procedure called *ExecuteConcurrent*. As in NR-UC, the ExecuteConcurrent procedure utilized by PREP-UC requires that the user provides the operation to be executed on the underlying sequential data structure, the arguments to that operation and a means to determine if the operation is read-only. For the latter, we simply utilize an optional Boolean argument which if true, designates that the operation is read-only. There are some implementation related details regarding how the user can provide an operation to PREP-UC which we discuss in detail in § 5.

4.2 Choice of Correctness Condition

We have already mentioned the different requirements of buffered durable linearizability compared to durable linearizability. Most importantly, durable linearizability requires all operations to be persisted before they are completed whereas buffered durable linearizability allows operations to be persisted at some point after the operation is completed.

PREP-UC does not change the point at which an operation is completed compared to NR-UC. A thread can only return from an update operation after the operation is applied to the local replica and the response of the operation is stored. This implies the following three facts regarding completed operations: 1) any update operation that has been appended to the log but not applied to the local replica of the thread that invoked the update is incomplete (even if it has been applied to other replicas), 2) all completed update operations correspond to log entries that precede the completedTail index and 3) the response of a read-only operation will always reflect a snapshot of the data structure that is up-to-date with the completedTail. These facts motivated our decision to persist the operation log, two persistent replicas and the completedTail all of which contribute to the satisfaction of durable linearizability without requiring an infinite log.

Durable linearizability guarantees that in the worst case, at most one operation per thread can be lost as a result of a crash. For systems with a large number of concurrent threads the number of operations that can be lost due to a crash is already relatively high. Moreover, we pay a significant performance cost to guarantee durable linearizability in terms of extra flushes and fences as well extra synchronization between threads. In our case, durable linearizability forces us to persist the operation log and completedTail. This is a performance concern because the operation log is a global object shared between all NUMA nodes. Due to the fact that the read and write latency of NVM is higher than DRAM, placing the log in NVM increases the latency of update operations and placing the completedTail in NVM increases the latency of both update and read-only operations.

A buffered durable linearizable implementation could avoid persisting both the operation log and the completedTail index which would lower the overhead related to persistence. Of course, with a buffered durable linearizable implementation we could lose many operations as a result of a crash. Our buffered durable linearizable implementation of PREP-UC still utilizes the persistence thread and

two persistent replicas. The number of operations that can be lost after a crash in the worst case depends on how frequently we persist one of the persistent replicas. If we persist the active persistent replica after applying ϵ operations then the number of completed operations that can be lost after a single crash is bounded in terms of ϵ. Specifically, a single crash would result in at most $\epsilon + \beta - 1$ operations being lost where β is the batch size which is the same as the number of threads per NUMA node. We provide more detail on this bound in § 5.1.

We argue that for some applications a buffered durable linearizable PUC can be favourable compared to a durable linearizable PUC. In § 6 we demonstrate that the buffered durable linearizable implementation of PREP-UC significantly outperforms the durable linearizable version even when the value of ϵ is fairly small.

Correctness. The shared log stores update operations. The order that operations were added to the log is precisely the linearization order of the operations. The durability order, i.e., the order in which operations are persisted, will be a prefix of the linearization order since we can only apply an update to a persistent replica once it has been added to the log, at which point its linearization order is fixed. Durable linearizability requires that all completed operations are recovered after a crash. After a crash, the durable linearizable implementation of PREP-UC recovers the entire contents of the log. An update operation is completed only after it is applied to the local replica corresponding to the NUMA node of the process that invoked the update. Since updates can only be applied to a replica after they are added to the log, recovering the entire contents of the log guarantees that all completed operations are recovered following a crash. Buffered durable linearizability requires that a prefix of the completed operations are recovered after a crash. After a crash, the buffered durable linearizable implementation of PREP-UC recovers a prefix of the log which is a prefix of completed operations. In § 5 we discuss worst case executions in terms of the number of operations that may be lost after a crash for both implementations of PREP-UC.

Liveness. PREP-UC, like NR-UC, relies on blocking through the use of locks, specifically trylocks and reader-writer locks. Our implementation of PREP-UC is deadlock-free however, there are two simple changes that one could make in order to achieve starvation-freedom. To understand the liveness guarantees of PREP-UC we can examine the points at which worker threads contend on resources.

During update operations in PREP-UC, worker threads add their operation to the batch associated with the local replica. Each worker thread has its own slot in the batch thus adding an operation to the batch does not require any synchronization. Worker threads within each node contend on the trylock to determine which thread on the node will become the combiner. Multiple combiners contend on the shared log, specifically, they must each execute a CAS to reserve log entries. An adversarial scheduler could schedule threads such that one thread never completes this CAS. Replacing the CAS with a fair lock would allow for starvation-free update operations. In practice we utilize a finite log. A log entry can only be reused once it has been applied to every replicas. Consequently, slow replicas can become a progress problem. Our implementation utilizes a simple helping mechanism which ensures that slow replicas do not cause deadlock. We discuss this in greater detail in § 5.

```
1  struct Operation:
2      void*           op = null
3      void*           args[MAX_ARGS] = null
4      uint            argCount = 0
5
6  struct LogEntry :
7      bool            emptyBit = 0
8      Operation       op
9
10 class Replica :
11     DataStructure*  ds
12     Operation       batch[β]
13     void*           responses[β]
14     uint            localTail = 0
15     TryLock         combinerLock
16     RWLock          rwLock
17
18 class PReplica :
19     DataStructure*  ds
20     uint            localTail = 0
21
22 class PREP-UC :
23     LogEntry        d_sharedLog[LOG_SIZE]
24     Replica         replias[N]
25     PReplica        p_replicas[2]
26     uint            logMin = LOG_SIZE - 1
27     uint            logTail = 0
28     uint            d_completedTail = 0
29     uint            p_activePReplica = 0
30     uint            flushBoundary = ϵ
```

Algorithm 1: PREP-UC Data Types. Variables that are always allocated from NVM are prefixed with p_. Variables that are allocated from NVM only for the durable linearizable implementation are prefixed d_. N is the number of NUMA nodes. β is batch size which is the same as the number of threads per NUMA node

During read-only operations, worker threads contend on the reader-writer lock of the local replica. The worker thread must claim the lock in read mode. An adversarial scheduler could schedule threads such that update operations are able to infinitely hold the reader-writer lock in write mode starving a thread attempting to claim it in read mode. To allow for starvation-free read-only operations we would simply use a starvation-free reader-write lock.

5 PREP-UC: IMPLEMENTATION DETAILS

In this section we will present the implementation details of PREP-UC. We will discuss both the buffered durable linearizable implementation of PREP-UC and the durable linearizable implementation. To better differentiate between the two implementations we will refer to the durable linearizable implementation of our PUC as PREP-Durable and the buffered durable linearizable implementation as PREP-Buffered. We will continue to use PREP-UC to refer to both implementations. Since the buffered durable linearizable implementation is the same as the durable linearizable implementation with some mechanisms removed we will begin by presenting the former.

5.1 Buffered Durable Linearizable PREP-Buffered

Algorithm 1 shows the data types of variables utilized by PREP-UC. The two variables prefixed with p_, namely the p_replicas

array and p_activePRepica are allocated from NVM. The p_replica array corresponds to the two persistent replicas. The PReplica class has fewer data members compared to the Replica class. Since the persistent replicas are only accessed by the persistence thread there is no need for the persistence thread to utilize flat combining. More specifically, the persistent replicas do not need to be locked and they have no use for the batch and response arrays. As with the volatile replicas, each persistent replica has a pointer to its copy of the sequential data structure and a localTail.

Each persistent replica, including the replica's copy of the sequential data structure, must be allocated from NVM. Allocating NVM requires the use of a *persistent allocator* [1, 15]. A persistent allocator services dynamic memory allocations via memory mapped files backed by NVM (commonly referred to as persistent memory files). Minimally, a PUC requires a persistent allocator that guarantees the following: 1) the allocator's operations do not corrupt allocated objects in the event of a crash and 2) after a crash allocated objects remain at the same virtual address. The second requirement ensures that pointers utilized by the sequential data structure are not invalidated after a crash. In practice this requirement would be fulfilled by requiring that the persistent memory file used by the PUC is always mapped to the same virtual address.

The underlying sequential implementation may perform dynamic memory allocations. In the case of the persistent replicas, we must ensure that the allocations performed by the sequential implementation utilize a persistent allocator. Unfortunately, we cannot globally override the system allocator (for example using LD_PRELOAD) with a persistent allocator because this would cause all objects to be allocated from NVM. We also cannot provide a persistent allocator to the sequential implementation since this would require modifying the sequential code.

To avoid modifying the sequential implementation we instead encapsulate the standard memory allocation functions in a wrapper that allows a thread to swap between the system allocator and a persistent allocator. We utilize a thread local variable which acts as a flag to enable/disable the persistent memory allocator. A thread can locally enable the persistent allloctor by setting the flag. Doing so ensures that all allocations by the thread will utilize the persistent allocator. When the thread wants to disable the persistent allocator and return to using the system allocator it simply reverts the flag back to its initial state. With this mechanism in place, the persistence thread can set the flag to enable the persistent alloctor before calling any methods on the sequential object. Once control returns to PREP-UC the persistence thread swaps back to the volatile allocator. Worker threads operating on the volatile replicas never enable the persistent allocator. The mechanism for swapping the allocator is built into PREP-UC and does not require modifications to the sequential implementation or intervention from the user. The sequential implementation cannot dynamically load its own allocator, however, if the sequential implementation wants to use a custom allocator it can do so by statically linking the allocator.

Algorithm 2 shows the function executed by the persistence thread. The function utilizes a single infinite loop. In each iteration of the loop the persistence thread will attempt to update the active persistent replica and persist it if necessary. The persistence thread first utilizes the integer variable p_activePReplica to identify the active persistent replica. The p_activePReplica variable is allocated

```
1  def UpdatePersistentReplicas() :
2      while true :
3          auto rep = p_replicas[p_activePReplica]
4          auto tail = d_completedTail
5          if tail <= rep.localTail :
6              continue
7          UpdateFromLog(tail, rep)
8          rep.localTail = tail
9          if flushBoundary <= tail :
10             WBINVD()
11             SFENCE()
12             flushBoundary += ϵ
13             p_activePReplica = !p_activePReplica
14             CLFLUSH(p_activePReplica)
```

Algorithm 2: UpdatePersistentReplicas function executed by the persistence thread.

from NVM and is explicitly persisted to ensure that it can be recovered and used to identify the active persistent replica following a crash. Initially the p_activePReplica variable is set to 0 which corresponds to the first of the two persistent replicas.

Next, the persistence thread reads the completedTail index and brings the active persistent replica up-to-date with the completedTail via the UpdateFromLog function. The *flushBoundary* index tracks the next log entry after which the active persistent replica must be persisted. Initially the flushBoundary is ϵ where ϵ is provided as input at compile time. Generally, we want to wait until the log is close to being full to flush the active persistent replica. A smaller ϵ means that we will swap between the active and stable persistent replicas more frequently which requires more writes to NVM. In § 6 we discuss some reasonable values for ϵ.

When the persistence thread finds that the completedTail is greater than or equal to the flushBoundary it will stop updating the active persistent replica then persist it. To persist the active persistent replica we execute a WBINVD instruction via a system call then execute an SFENCE instruction. This will write the active persistent replica out of the volatile cache and back to NVM. Next we advance the flushBoundary index, increasing it by ϵ. Finally the persistence thread will swap the active and stable persistent replicas by modifying and flushing the p_activePReplica variable. In the next iteration of the loop a new update cycle will begin and the persistence thread will begin updating the new active persistent replica.

Algorithm 4 shows the function utilized by worker threads to reserve entries in the shared log. A thread is blocked from reserving new log entries if the flushBoundary is less than the logTail. When the flushBoundary is less than the logTail this means that the active persistent replica must be written back to persistent memory. In this case we cannot allow new operations to be added to the log otherwise, the number of operations that can be lost after a single crash occurs would vary depending on the speed of the persistence thread. Note that log entries contain the emptyBit and an operation. In the pseudocode we represent the operation as a function pointer, the number of arguments accepted by the function and an array of arguments. We require access to this information primarily for the durable linearizable implementation. The buffered durable linearizable implementation can (and does) make use of high level language features such the std::function in C++ to encapsulate this information.

Reusing log entries and logMin. Algorithm 3 shows the UpdateOrWaitOnLogMin function. This function is necessary to allow for a finite log since it ensures that a log entry is never overwritten until all replicas have applied the entry. The bulk of the UpdateOrWaitOnLogMin function implements the algorithm described in section 5.6 of [4] which primarily involves updating the *logMin* index. The logMin index points to the log entry prior to which all other log entries have been applied to every replica meaning they can be safely reused. The logMin index is only updated once a thread reserves the log entry at the *lowMark* index which is equivalent to the $logMin - beta$. The lowMark is defined in terms of *beta* because *beta* is the maximum number of log entries that can be reserved with a single update to the logTail.

When a combiner thread reserves the log entry at the lowMark it must scan the localTails of every replica and updates the logMin index to point to the same log entry as the lowest localTail of all replicas. In our implementation, this is accomplished by setting the logMin equal to the lowest localTail plus the log size minus 1. This approach requires that ϵ must not be greater than the $LOG\_SIZE - \beta - 1$.

The addition of the two persistent replicas does not add significant complexity to this function. If a combiner needs to update the logMin then it must check the localTails of both persistent replicas. We take some liberties when portraying this in the pesudocode by abusing the plus operator on arrays. In practice, all replicas can be stored in a single array or a helper function can be used to iterate over all replicas. The former is convenient for simplicity but adds some minor overhead.

Due to the fact that only one of the persistent replicas is updated at a time, it is likely that the stable persistent replica will be far behind the logTail. This is especially true if ϵ is large since a large ϵ means that the stable and active persistent replicas will swap infrequently. This can be problematic if the lowMark happens to occur before the next flush boundary since this would mean that the combiner attempting to update the logMin will be blocked and the persistence thread will also eventually be blocked due to the fact that it must apply the operations that the blocked combiner has not yet written to the log. Note that while this is more likely to occur with the persistent replicas since they will be updated less frequently and less quickly compared to the volatile replicas, the same problem can occur in NR-UC with a slow volatile replica.

Unfortunately, Calciu et al were somewhat vague when describing the way that they reuse log entries with regards to ensuring that a slow thread does not cause deadlock. To avoid this problem we simply implement a mechanism that allows a combiner thread that is blocked from updating the logMin to signal that the replica causing the block needs to be updated. For the persistent replicas this is simply done by reducing the flushBoundary. For the volatile replicas we use an array of booleans where each replica has a dedicated slot in the array. While a combiner thread is waiting for the logMin to be updated it will check this array to see if its local replica needs to be updated. If the slot corresponding to the combiner's local replica is true then the combiner will bring its local replica up to date with the completedTail index. With this mechanism in place deadlock caused by a slow replica is avoided.

Recovery Procedure. Following a crash, we must recover the stable persistent replica. At any point in time NVM contains the p_replicas array and the p_activePReplica integer. During recovery we utilize p_activePReplica to identify which replica was the stable persistent replica prior to the crash. In order to safely resume execution, instead of creating a new copy of the sequential object for each volatile replica, we instantiate all N volatile replicas as copies of the stable persistent replica. We do the same for the other persistent replica. All other variables will also be set to their initial values. The localTails of all replicas will be 0, the log will be empty, both the logTail and completedTail will be 0 and the flushBoundary will be ϵ. At this point we can spawn a new persistence thread then worker threads can begin to execute operations on the object.

Worst Case Execution. For any single crash event PREP-Buffered guarantees that at most $\epsilon + \beta - 1$ completed update operations will be lost. It is easy to see that ϵ operations can be lost after a crash since we only persist a replica after applying at least ϵ update

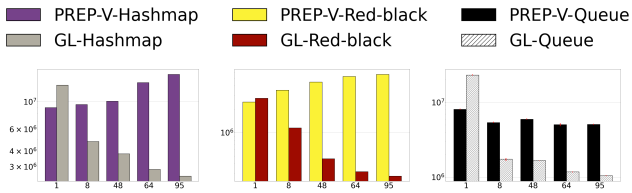
```
def UpdateOrWaitOnLogMin(newTail) :
    uint lowMark = logMin - β
    while lowMark < newTail :
        uint lowest = ∞
        uint repID = 0
        foreach rep in replicas + p_replicas :
            uint lt = rep.localTail
            repID++
            if lt < lowest :
                lowest = lt
        if lowest + LOG_SIZE - 1 == logMin :
            if repID >= N :
                if p_activePReplica != repID
                 and flushBoundary >= lowMark :
                    flushBoundary = lowMark - 1
            else :
                updateReplicaNow[repID] = true
            while replicas[repID].localTail == lowest :
                # Wait
            updateReplicaNow[repID] = false
            continue
        logMin = lowest + LOG_SIZE - 1
        SFENCE()
        lowMark = logMin - β
    else :
        while lowMark < newTail:
            if updateReplicaNow[repID]:
                uint repID = getLocalReplicaID
                rep = replicas[repID]
                rep.rwLock.writeLock()
                UpdateFromLog(completedTail, rep)
                rep.rwLock.writeUnlock()
                updateReplicaNow[repID] = false
            lowMark = logMin - β
```

Algorithm 3: UpdateOrWaitOnLogMin function responsible for allowing log entries to be reused.

```
def ReserveLogEntries(replica, numEnts) :
    while true :
        uint tail = logTail
        uint newTail = tail + numEnts
        while flushBoundary < tail :
            # Block until stable persistent
            # replica is up-to-date with tail
        if CAS(logTail, tail, newTail) :
            UpdateOrWaitOnLogMin(newTail)
            return tail
```

Algorithm 4: ResLogEntries function incorporating blocking on the flushBoundary.

operations. β represents the maximum batch size which is the same as the number of threads per NUMA node. This also means that β is the maximum number of log entries that a combiner thread can reserve. If the logTail is $\epsilon - 1$ then neither persistent replica has been persisted. The active persistent replica may or may not be up-to-date with the logTail. A thread operating on one of the volatile replicas can add at most β new update operations to the log. All β update operations could complete before the active persistent replica is updated and persisted. If a crash occurs at this point then the stable persistent replica that we recover will be empty since it was never updated. In this case the original $\epsilon - 1$ operations are lost along with the β operations that were last added to the log. Since the log returns to empty after a crash, for c crash events at most $c(\epsilon + \beta - 1)$ completed update operations will be lost.



(a) Hashmap **(b) Red-black tree** **(c) Priority queue**

Figure 1: Throughput for volatile UCs: PREP-V (Volatile PREP) and Global Lock (GL). Y-axis is ops/sec. X-axis is number of threads. (a) and (b) 90% read-only workload. 1 million keys. (c) 100% update workload where workers execute pairs of enqueue and dequeue operations.

5.2 Durable Linearizable PREP-Durable

The durable linearizable implementation must ensure that all operations are persisted before they complete. To achieve this we build on top of the buffered durable linearizable implementation by persisting the log and the completedTail index along with the two persistent replicas. In practice there is some nuance related to persisting the log. A log entry contains a pointer to the function that carries out the update operation along with its arguments. High level language features, such as std::function in C++, allow for more easily storing and executing functions with different signatures. Unfortunately, these features were not designed to be used with NVM. An instance of the std::function in C++ that we persisted would no longer usable following a crash. For this reason we instead persist the raw function pointers which are valid following a crash assuming the functions were not dynamically loaded at runtime. This makes calling the function more difficult since we cannot store information about the function signature. In this case, we rely on an *Execute* method provided by the underlying sequential implementation which accepts a function pointer and its arguments and performs the appropriate function call. This was the same approach utilized Calicu et al. for NR-UC, although they could have used a wrapper since NR-UC is not persistent. The Execute function is essentially a switch statement where we have one case per public method. When there are few public methods this is actually faster than wrappers like std::function. While slightly less convenient, this requires no more work compared to wrappers like std::function which still require at least one line of code to wrap the function call. In practice the Execute function could be generated via preprocessor macros or written by the programmer.

Persisting the completedTail is fairly straightforward. The compeltedTail is updated via a CAS. If a thread successfully updates the completedTail it must then persist it. Since the completedTail is monotonically increasing it is sufficient to perform a CLFLUSH instruction on the address of d_completedTail after the CAS. We can reduce the number of CLFLUSH instruction performed by marking the completedTail to indicate whether or not it has been persisted, a technique that has been utilized in existing hand-crafted persistent data structures [14, 33].

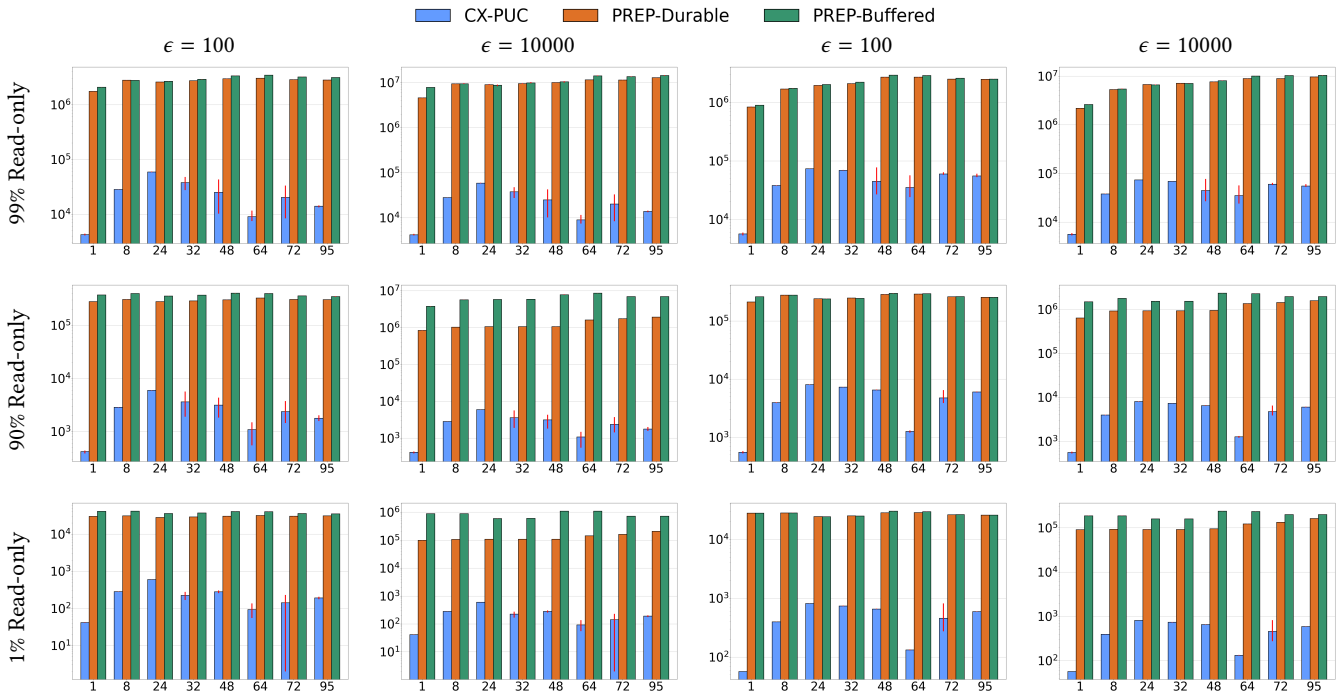
Recovery Procedure. As with the buffered durable linearizable implementation, the recovery procedure of PREP-Durable must identify the stable persistent replica using the p_activePReplica variable. The next step of the recovery procedure is to instantiate the active persistent replica as a copy of the stable persistent replica. Once we have both persistent replicas, we update the active persistent replica by applying all operations in the log corresponding to non-empty log entries starting from the localTail of the stable persistent replica up to the completedTail. After bringing the active persistent replica up-to-date we persist it and swap the value of p_activePReplica. At this point we can create the N volatile replicas as copies of the stable persistent replica. Finally, we empty the log and set the localTails of every replica, the logTail and the completedTail to 0.

Worst Case Execution. PREP-Durable ensures that all operations are persisted before they complete. In the worst case at most n pending update operations can be lost as a result of a crash where n is the maximum number of worker threads.

6 EVALUATION

We implemented both PREP-Buffered and PREP-Durable in C++ and compared them against CX-PUC of Correia et al [12]. We test the performance in terms of throughput, in operations per second, of these PUCs across various workloads using several different sequential data structures. We utilize the same benchmark as [3] for conducting the tests. We ran the experiments on a NUMA system with 2 Intel Xeon Gold 5220R 2.20GHz processors, each of which has 24 cores and 48 hardware threads. The system has a 36608K L3 cache, 1024K L2 cache, 64K L1 cache and 1.5TB of NVRAM. The NVRAM modules installed on the system are Intel Optane DCPMMs [9]. The code was compiled with GCC 9.3.0 with the highest optimization level of -O3. In each test we prefill the data structure to 50% capacity and begin performance measurements after prefilling.

All experiments comparing CX-PUC and PREP-UC utilize the same allocators, specifically jemalloc 5.0.1-25 for the volatile allocator and the same simple free-list based allocator used by Correia et al. for the persistent memory allocator. All experiments start with a freshly created 64GB persistent memory file (much larger than necessary for these micro-benchmarks). For PREP-UC we utilize a log size of 1 million for all experiments. We utilize at most 95 of the 96 available hardware threads as worker threads which ensures that there is an available hardware thread for the persistence thread utilized by PREP-UC. Every thread, including the persistence thread,



(a) Resizable Hashmap **(b) Red-black tree**

Figure 2: Throughput for sets with 1 million keys. Y-axis is ops/sec. X-axis is number of threads.

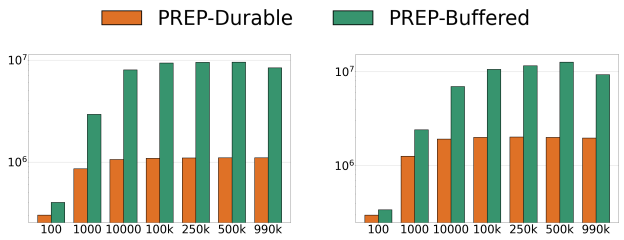
is bound to a unique processor such that all available processors on a NUMA node are utilized before utilizing processors on other nodes. More specifically, experiments for up to 24 threads utilize the available processors on a single node and 24 to 48 threads utilize all available processors and hyper-threads on a single node; 49 to 72 threads and 72 to 96 threads do the same on the second node. The results shown here are the average of 3 trials where each trial lasts for 10 seconds. To compare with a familiar baseline, in Figure 1 we show the results of using some volatile UCs, specifically PREP-V, our implementation of PREP-UC with all persistence related code removed and a simple UC that protects a single copy of the sequential data structure with a global lock.

Hashmap. Figure 2a shows the throughput for a resizable linked list based hashmap with 1 million keys implemented via each of the PUCs. In all workloads keys were accessed according to a uniform distribution. As expected CX-PUC performs quite poorly for all workloads. This is primarily due to the fact that CX-PUC must flush the entire replica after each update operation and reads are also executed on persistent replicas. By comparison both PREP-Durable and PREP-Buffered perform significantly better. For the extremely read-dominant workload PREP-Durable has similar performance to PREP-Buffered and both scale well. For the other workloads with more update operations we can see that the performance difference between the two implementations is larger due to the added overhead on update operations that results from persisting the log. The first column of Figure 2a PREP-Buffered and PREP-Durable used an ϵ of 100 which is extremely small and limits scaling for update-heavy workloads. When ϵ is larger we scale well for all workloads.

Red-black Tree. Figure 2b shows the throughput for a red-black tree with 1 million keys implemented via each of the PUCs where keys are accessed according to a uniform distribution. The experiments with the red-black tree produce similar results to the hashmap. CX-PUC performs poorly while PREP-Durable and PREP-Buffered have similar performance due to the low value used for ϵ.

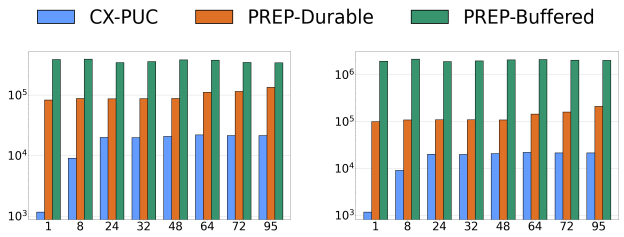
Effects of ϵ. Columns one and three of Figure 2 PREP-UC used an ϵ of 100 which is extremely small. We can see that with this low value for ϵ the performance of PREP-Durable is very close to the performance of PREP-Buffered. This is expected since the WBINVD instruction used to persist a replica is very expensive. As seen in columns two and four of Figure 2, when ϵ is larger, for example 10000 which is only 1% of the log size, there is a significant increase in throughput. Moreover, ϵ is large, the difference between PREP-Durable and PREP-Buffered is much larger especially for workloads with more update operations. Figure 3 shows how different values of ϵ effect the throughput of PREP-UC for a hashmap.

Priority Queue. Figure 4 shows the throughput for a persistent concurrent priority queue implemented via each of the PUCs. The sequential implementation for this priority queue is the C++ standard library priority_queue. We prefill the priority queue with 50000 items. Our experiments with the priority queue consisted of only 100% update workloads. To prevent major changes in the size of the priority queue, every worker thread performs an enqueue followed



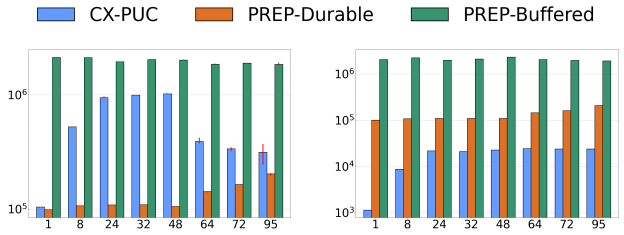
(a) 48 threads (b) 95 threads

Figure 3: PREP-UC throughput (ops/sec) for hashmap for different values of ϵ. 90% read-only workload 1 million keys.



(a) **Average 50k items.** $\epsilon = 1000$ (b) **Average 500k items.** $\epsilon = 10000$

Figure 4: Throughput for priority queue. Y-axis is ops/sec. X-axis is number of threads. 100% update workload where workers execute pairs of enqueue and dequeue operations.

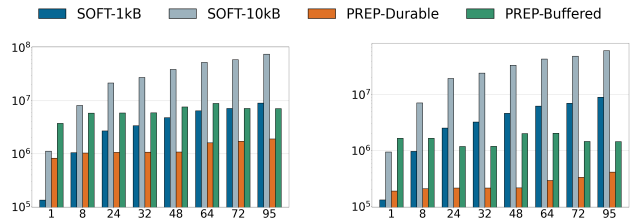


(a) **Average 500 items.** $\epsilon = 10000$ (b) **Average 50k items.** $\epsilon = 10000$

Figure 5: Throughput for stack. Y-axis is ops/sec. X-axis is number of threads. 100% update workload where workers execute pairs of push and pop operations.

by a dequeue. In this case the size of the data structure is relatively small. When ϵ is also small the difference in performance between PREP-UC and CX-PUC is not as large compared to the hashmap or red-black tree. When ϵ is larger PREP-Buffered significantly out performs the other PUCs while PREP-Durable still significantly outperforms CX-PUC. In practice if the data structure is very small could flush the entire address space of a replica rather than using WBINVD.

Stack. Figure 5 shows the throughput for a persistent concurrent stack implemented via each of the PUCs. We prefill the stack with 500 items and as with the priority queue we utilize only 100% workloads where worker threads perform a push followed by a pop. When ϵ is large PREP-Buffered performs much better while PREP-Durable still performs poorly. CX-PUC performs well in this case and when ϵ is small it outperforms PREP-UC. The main reason that PREP-UC does not perform as well for this case is the fact



(a) **90% Read-only 1 million keys** (b) **50% Read-only 1 million keys**

Figure 6: Throughput of PREP-UC Resizable Hashmap compared to SOFT hashtable with 1k buckets (SOFT-1kB) and SOFT hashtable with 10k buckets (SOFT-10kB) for different thread counts. Y-axis is ops/sec. X-axis is number of threads. $\epsilon = 10000$ (1% of the key range).

that the size of the data structure is extremely small. This allows for better performance using the persistence strategy of CX-PUC, which write-backs a specific address range. PREP-UC suffers from the large overhead of the WBINVD instruction which must be executed frequently when ϵ is small since the log fills quickly for update only workloads.

PREP-UC versus Hand-Crafted Hashtable. To frame the performance of PREP-UC relative to hand-crafted data structures we compare the realizable hashmap implemented via PREP-UC against the hashtable of Zuriel et al. Zuriel et al. implemented a hand-crafted persistent concurrent hashtable utilizing their *sets with an optimal flushing technique (SOFT)* [36]. The SOFT algorithm maintains two copies of the data structures keys where each key is stored in both persistent memory and volatile memory. SOFT persists only one copy of the data structure keys along with some meta data which is required to determine if the key was still in the data structure prior to a crash. Read-only operations in SOFT do not perform any flushes or fences. The SOFT hashtable has a fixed number of buckets each of which is a persistent linked-list. The number of buckets can obviously impact the performance of a non-resizable hashtable. We compare PREP-UC against the SOFT hashtable using one thousand buckets (SOFT-1kB in Figure 6) and the SOFT hashtable using ten thousand buckets (SOFT-10kB in Figure 6).

For this experiment we utilized the same persistent memory allocator that was used by SOFT for PREP-UC, specifically libvmmalloc which is part of Intel's Persistent Memory Development Kit (PMDK) [10]. In Figure 6 we see that SOFT outperforms PREP-UC especially for update-heavy workloads.

In general it is expected that a hand-crafted persistent concurrent data structure should perform better compared to a PUC. There are two important factors that contribute to the better performance of SOFT over PREP-UC. Firstly and most importantly, an update operation in SOFT (and any hand-crafted data structure) can easily determine the exact memory words modified by the update which must be written back to persistent memory. SOFT persists only those modified memory words which are required to recover the data structure following a crash. This is in contrast to PREP-UC which relies on periodically executing the expensive WBINVD instruction to write back modifications made to the active persistent replica. PREP-UC could not utilize the same persistence mechanism as SOFT because PREP-UC views the sequential data structure as

a black box and therefore it cannot determine the exact memory words modified by an update operation. Secondly, since SOFT does not persist data structure links, it avoids any overhead that would be caused by traversing and updating a data structure stored entirely in persistent memory. The persistent replicas utilized by PREP-UC are stored entirely in persistent memory.

7 CONCLUSION

In this work we presented PREP-UC, a PUC based on the node replication UC [4]. We implemented two versions of PREP-UC guaranteeing buffered durable linearizability, and durable linearizability, respectively. PREP-Buffered, bounds the number of operations that can be lost after a crash based on the input parameter ϵ. We demonstrate that even when ϵ is small, our buffered durable linearizable PUC out performs durable linearizable equivalents for several different data structures and workloads.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments and suggestions. We also thank William Sigouin for his contributions to early implementations of PREP-UC and we thank Pedro Ramalhete for his insights that helped inspire this work.

This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, Waterloo Huawei Joint Innovation Lab project "Scalable Infrastructure for Next Generation Data Management Systems", NSERC Discovery Launch Supplement: DGECR-2019-00048, NSERC Discovery Program grant: RGPIN-2019-04227, and the University of Waterloo.

REFERENCES

[1] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2016. Makalu: Fast recoverable allocation of non-volatile memory. *ACM SIGPLAN Notices* 51, 10 (2016), 677–694.

[2] Trevor Brown and Hillel Avni. 2016. PHyTM: Persistent Hybrid Transactional Memory. *Proc. VLDB Endow.* 10, 4 (2016), 409–420.

[3] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. *Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time.* Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/3332466.3374542

[4] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. *ACM SIGPLAN Notices* 52, 4 (2017), 207–221.

[5] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[6] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[7] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3210377.3210400

[8] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Available at: intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html. , 286 pages.

[9] Intel Corporation. 2017. Intel Optane Persistent Memory. Available at: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.

[10] Intel Corporation. 2018. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 271–282. https://doi.org/10.1145/3210377.3210392

[12] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent Memory and the Rise of Universal Constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/3342195.3387515

[13] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2020. *A Wait-Free Universal Construction for Large Objects.* Association for Computing Machinery, New York, NY, USA, 102–116. https://doi.org/10.1145/3332466.3374523

[14] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 373–385.

[15] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/3423211.3425671

[16] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

[17] Panagiota Fatourou and Nikolaos D. Kallimanis. 2020. The RedBlue Family of Universal Constructions. *Distrib. Comput.* 33, 6 (dec 2020), 485–513. https://doi.org/10.1007/s00446-020-00370-7

[18] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[19] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

[20] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent.* Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

[21] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[22] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[23] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. I–XX, 1–508 pages.

[24] Maurice P. Herlihy. 1988. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. Association for Computing Machinery, New York, NY, USA, 276–290. https://doi.org/10.1145/62546.62593

[25] M. P. Herlihy and J. M. Wing. 1987. Axioms for Concurrent Objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 13–26. https://doi.org/10.1145/41625.41627

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[27] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 452–465. https://doi.org/10.1109/ISCA.2018.00045

[28] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices* 52, 4 (2017), 329–343.

[29] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. *Pronto: Easy and Fast Persistence for Volatile Data Structures.* Association for Computing Machinery, New York, NY, USA, 789–806. https://doi.org/10.1145/3373376.3378456

[30] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 91)*, Andréa W. Richa (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 37:1–37:16. https://doi.org/10.4230/LIPIcs.DISC.2017.37

[31] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[32] Qing Wang, Youyou Lu, Junru Li, Minhui Xie, and Jiwu Shu. 2022. Nap: Persistent Memory Indexes for NUMA Architectures. *ACM Trans. Storage* 18, 1, Article 2 (jan 2022), 35 pages. https://doi.org/10.1145/3507922

[33] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[34] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3503221.3508436

[35] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L Scott. 2020. Montage: A general system for buffered durably linearizable data structures. *arXiv preprint arXiv:2009.13701* (2020).

[36] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.