The Fence Complexity of Persistent Sets

by

Gaetano Coccimiglio

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

Abstract

This thesis studies fence complexity of concurrent sets in a non-volatile shared memory model. I consider the case where CPU registers and cache memory remain volatile while main memory is non-volatile. Flush instructions are required to force shared state to be written back to non-volatile memory. These flush instructions must be accompanied by the use of expensive fence instructions to enforce ordering among such flushes. Collectively I refer to a flush and a fence as a psync. In this model the system can crash at any time. When the system crashes the contents of volatile memory are lost. I consider lock-free implementations of the set abstract data type and the safety properties of strict linearizability and durable linearizability. Strict linearizability forces crashed operations to take effect before the crash or not take effect at all; the weaker property of durable linearizability enforces this requirement only for operations that have completed prior to the crash event. In this thesis, I consider classes of strict linearizable implementations that guarantee operations take effect at or before the point when the operation is persisted. I prove two lower bounds for lock-free implementations of the set abstract data type. First, I prove that it is impossible to implement strict linearizable lock-free sets in which read-only (or search) operations do not flush or fence. Second, I prove that for any durable-linearizable lock-free set there must exist an execution in which some process must perform at least one redundant psync as part of an update operation. I also present several implementations of persistent concurrent lock-free sets. I evaluate these implementations against existing persistent sets. This evaluation exposes the impact of algorithmic design and safety properties on psync complexity in practice as well as the cost of recovering the data structure following a system crash.

Acknowledgements

I would like to thank my supervisor Dr. Trevor Brown for his guidance and assistance throughout my studies at the University of Waterloo. He was always available whenever I needed feedback and was always very supportive and encouraging. He has made my studies in the Master of Mathematics program very enjoyable and I look forward to continued collaboration with him.

I would also like to thank my thesis committee readers, Dr. Wojciech Golab and Dr. Ian Munro for their comments and feedback.

I would like to thank Dr. Srivatsan Ravi of the University of Southern California for his contributions and assistance for a paper related to this thesis submitted to DISC 2021 and his guidance throughout writing this thesis.

I would like to thank the Graduate Studies administrators at the University of Waterloo for their assistance in quickly processing my transfer to the University.

Finally I would like to thank my parents and my brother for their support and assistance throughout my graduate studies. They were all very encouraging of my choice to pursue higher education and without them I doubt that I would have succeeded.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Within recent years, byte-addressable non-volatile memory (NVM) has become commercially available. In particular, I focus on Intel's Optane persistent memory [20]. This byte-addressable NVM can be leveraged to allow applications to persist even if the system loses power. This offers a promising avenue for increasing application uptime by giving developers a clear path to adding an increased level of fault tolerance to their data structures.

In the past, only data stored in block addressable *persistent* storage devices would persist through power failures or system reboots. These persistent storage devices include magnetic disk drives or flash solid-state drives (SSDs). While they offer very large capacities, they suffer from the fact that they have latency that is orders of magnitude higher than DRAM and they are not byte-addressable.

The recently available NVM hardware offers a faster alternative to compared to magnetic disk drives or SSDs. NVM has characteristics that place it between DRAM and SSDs. The persistent memory offers latency that is comparable to, but still higher, than DRAM and much lower than SSDs. The capacity of NVM is much larger than DRAM and the bandwidth is lower than DRAM. NVM also has asymmetric read and write latency and asymmetric read and write bandwidth [42]. NVM has been also been shown to be more sensitive to data locality compared to DRAM [42, 51].

This thesis considers *persistent data structures*. Note that in this thesis *persistent data structures* refers to data structures that *persist* through system crashes (power failures). This differs from the data structure community where *persistent data structures* refer to data structures that support multiple versions [27, 49].

The characteristics of NVM make it a compelling option for achieving efficient implementations of persistent data structures compared to SSDs, however, it is clear that achieving performance comparable to transient data structures stored in DRAM will be a challenge. If it were possible to replace all volatile memory with persistent memory then the task of implementing persistent data structures would be trivial. Most systems feature a small number of CPU registers, a limited amount of cache memory and a larger capacity of DRAM. Registers, cache memory and DRAM are all volatile. This means that the contents these forms of memory are lost if the system loses power. Unfortunately, the newly available persistent memory will not entirely replace these volatile memory technologies. Intel Optane persistent memory exists alongside these other forms of volatile memory. Optane persistent memory utilizes Intel's 3D XPoint memory technology [21]. It is expected that CPU registers and cache memory will remain volatile and NVM will eventually replace DRAM [44, 58]. Currently, NVM coexists with DRAM where both are directly attached to the memory bus.[1]

This exposes the key challenge in designing efficient persistent data structures. Since the CPU registers and cache memory remain volatile, writes first occur in volatile memory. Without extra intervention by the programmer there is no guarantee that data in the volatile cache will be written to persistent memory before a crash occurs. Moreover, cache coherency protocols might flush data from the cache to persistent memory in a different order than the writes occurred. Forcing shared state to be written back to NVM sometimes requires the programmer to explicitly *flush* shared objects to NVM by using *explicit flush* and *persistence fence* primitives, the combination of which is referred to as a *psync* [41, 58]. Chapter 2 discusses these primitives and other aspects of computation model in greater detail. Informally, an *explicit flush* is used to write data to persistent memory but it is asynchronous. A *persistence fence* is used to block until all flushes preceding the persistence fence have completed.

Concurrent data structures are at the heart of scalable concurrent software applications. There has been a vast amount of research regarding volatile concurrent data structures e.g. [3, 4, 10, 11, 12, 35, 38, 40, 47]. Concurrent key-value store are one of the most commonly used types of concurrent data structures. Key-value stores are often the basis for implementing database indexes as well as components of other applications e.g. [26, 52, 48]. This thesis focuses specifically on persistent concurrent sets. In particular, this thesis presents a detailed study of the psync complexity of *concurrent sets* in theory and practice. Concurrent sets are implementations of the set abstract data type that allow multiple

[1]When used in Memory Mode, Intel Optane NVM requires a minimum of a 1:4 ratio of DRAM to NVM. In any other configuration mode the NVM is exposed through a filesystem - see Chapter 7 for further details.

concurrent processes to execute data structure operations. A more formal definition of sets is described in Chapter 2. I choose to study sets as a proxy for key-value stores. While sets export only a limited number of operations, they can be easily extended to create more robust key-value stores. The simplicity of sets allows us to focus specifically on issues related to achieving persistence.

While concurrent sets have been studied extensively for volatile shared memory [39], they are still relatively nascent in non-volatile shared memory. Concurrent data structures in volatile shared memory typically satisfy the *linearizability* safety property, data structures that utilize for non-volatile shared memory must consider the state of the persistent object following a full system crash. The safety property of *durable linearizability* satisfies linearizability and following a crash, requires that the object state reflect a consistent operation subhistory that includes all completed operations before the crash [41]. The stronger *strict linearizability* forces crashed operations to take effect before the crash or not take effect at all unlike durable linearizability which enforces this requirement only for operations that have completed prior to the crash event [1]. I describe these safety properties in more detail in Chapter 2.

I choose to study durable linearizability and strict linearizability because both of these safety proprieties provide *locality* and preserve *program order* which are known to be desirable qualities [39]. Informally, if a safety property provides locality (or compositionality) then whenever each object in some system satisfies the safety property, the system as a whole also satisfies the safety property [55]. Informally program order refers to the order in which any single process completes data structure operations. Another reason that I choose to focus on durable linearizability and strict linearizability is that both of these safety properties are defined using models in which processes can crash which makes them applicable to defining correctness of persistent sets.

Following a system crash a recovery procedure is used to return the object back to a consistent state. New data structure operations cannot begin until the recovery procedure completes. Strict linearizability was originally defined without the concept of recovery. I observe that there is a problem in the way that strict linearizability has been generalized for the case when the system can recover from crashes. In order to understand the intentions of Aguilera and Frølund consider this motivating example quoted from section 1 of [1] which describes why it is important to restrict when an operation can take effect: “suppose that a military officer presses a button to launch a missile during war, but the missile does not come out. It might be catastrophic if the missile is suddenly launched years later after the war is over.” If the system can launch the missile during the execution of the recovery procedure, from the perspective of the system the launch operation might still appear to complete before the crash, however, this behaviour is clearly problematic and appears to contradict the intentions of Aguilera and Frølund. In practice if a system crash occurs due to a power failure it is likely that the system will only reboot and recover at some later time. Moreover, it is unrealistic to bound the time between a crash and the following recovery. In chapter 2 I describe how strict linearizability does not exactly capture the intuition provided by the missile launch example. In this thesis I consider only classes of strict linearizable implementations that guarantee operations take effect at or before the point when the operation is persisted. This is discussed in greater detail in § 2.2.5. These restricted classes of strict linearizable implementations successfully capture the intuition of strict linearizability provided by Aguilera and Frølund.

Existing literature related to persistent sets has trended towards persisting less data structure state to minimize the cost of writing to NVM. For example, the Link-Free and SOFT [58] persistent list-based sets do not persist any *pointers* in the data structure. Instead they persist the *keys* along with some other metadata used after a crash to determine if the key is in the data structure. This approach requires at most a single psync for update operations; however, not persisting the structure results in a more complicated recovery procedure with slower worst case performance.

A manuscript by Israelevitz and nine other authors presented a seminal in depth study of the performance characteristics of real NVM hardware [42]. Their results may have played a role in motivating the trend to persist as little as possible and reduce the number of fences. In particular, they found (Figure 8 of [42]) that the latency to write 256 bytes and then perform a psync is at least 3.5x the latency to write 256 bytes and perform a flush but no persistence fence. Moreover, they found that the write bandwidth of NVM could be a severe bottleneck, as a write-only benchmark (Figure 9 of [42]) showed that NVM write bandwidth *scaled negatively* as the number of threads increased past four, and was approximately *9x lower* than volatile write bandwidth with 24 threads.

Although these results are compelling, it is unclear whether these latencies and bandwidth limitations are a problem for concurrent sets in practice. As it turns out, the push for synchronization mechanisms that minimize the amount of data persisted, and/or the number of *psyncs*, has many consequences, and the balance may favour incurring increased psyncs in some cases.

This thesis is structured as follows: In Chapter 2 I describe volatile memory and non-volatile memory models of computation. Next, in Chapter 3 I provide some preliminary information regarding metrics to evaluate persistent sets and I review existing persistent sets.

In Chapter 4 I prove the following lower bound: it is impossible to implement strict linearizable lock-free sets in which read-only operations do not flush and perform a persistence fence for the classes of strict linearizable implementations considered in this thesis. This establishes a clear theoretical separation between strict linearizable sets and durable linearizable sets. Based on current trends this lower bound would suggest that there would be practical performance issues for any strict linearizable set.

I provide a different theoretical bound in Chapter 5 where I prove that for any implementation of a durable linearizable concurrent lock-free set there must exist an execution in which some process performs a *redundant* psync as part of an update operation. Informally, a redundant psync is one that does not change the contents of persistent memory. In practice redundant psyncs could have a significant impact on performance since persistence fences introduce latency and flushes consume write bandwidth. This lower bound holds when the recovery procedure is constrained and when the recovery procedure is unconstrained.

In order to analyze the actual practical implications of the lower bounds I present a family of persistent concurrent set algorithms in Chapter 6. To implement these sets I extended a technique used by David et al [24]. I implement both strict linearizable and durable linearizable variants where the durable linearizable versions offer read-only operations that never perform a psync.

The culmination of these insights is described in Chapter 7 where I undertake a systematic empirical study of persistent sets. Specifically, I evaluate my persistent concurrent set algorithms against existing persistent sets to expose the impact of algorithmic design and safety properties on persistence fence complexity in practice as well as the cost of recovering the data structure following a system crash.

I find that, perhaps surprisingly, strict linearizable sets that do not offer persistence-free read-only operations sometimes perform as well as the durable linearizable sets that do. I note that the strict linearizable sets have significantly stronger bounds on the cost of recovering the data structure after a system crash (and that faster crash recovery is a critical use case for NVM hardware). The results of this thesis suggests that psync complexity is not a good predictor of performance in practice, thus motivating need for better metrics to compare persistent objects. These results suggest that, in contrast to existing work, minimizing psync complexity might not be the best approach for designing persistent sets. I recommend that researchers should not immediately sacrifice strict linearizability by prioritizing persistence-free searches and instead should begin with strict linearizable implementations.

Chapter 2

Model

In this chapter I introduce the shared memory model used in this thesis. I describe a typical volatile shared memory model and the non-volatile shared memory model.

2.1 Volatile Memory Computational Model

We present preliminaries for the standard *volatile* shared memory model and then explain how we extend the model to *non-volatile* (or *persistent*) shared memory.

Processes and shared memory. We consider an asynchronous shared memory system in which a set of $n \in \mathbb{N}$ processes communicate by applying *operations* on shared *objects*. Each process $p_i; i \in n$ has an unique identifier and an initial state. Each process can run at arbitrarily different speeds and the speed of any process can change at any arbitrary time. Processes can experience crashes. A crashed process is indistinguishable from a process that is extremely slow.

Objects and Implementations. An object is an instance of an *abstract data type* which specifies a set of operations that provide the only means to manipulate the object. An *abstract data type* defines a set of operations, a set of responses, a set of states, an initial state and a transition relation δ that determines, for each state and each operation, the set of possible resulting states and produced responses [2]. Here, $(q, \pi, q', r) \in \delta$ implies that when an operation π is applied on an object of type τ in state q, the object may move to state q' and return a response r.

An *implementation* of an object type τ (sometimes we just say object) provides a specific data-representation of τ by applying *primitives* on a set of shared *base objects* each of which is assigned an initial value. Throughout this thesis b is typically used to denote a base object.

Primitives. I assume that the primitives applied on base objects are *deterministic*. The main primitive that this thesis relies on is a generic *read-modify-write* (*RMW*) procedure applied to a base object [28, 37]. A read-modify-write primitive applied by a process to a base object b atomically updates the value of the b with a new value. The new value is a function $g(v, w)$ of the old value v and the input parameters w. The read-modify-write primitive returns a response to the process. The response is based on the function $h(v, w)$. The functions g and h are defined differently for specific read-modify-write primitives. This thesis relies on two examples of the RMW primitive, specifically *Compare-and-swap* (*CAS*) and *double-wide-compare-and-swap* (*DWCAS*).

Compare-and-swap (*CAS*) is an example of a RMW primitive. The CAS primitive accepts two input parameters, the desired new value of the base object *new* and the expected old value of the base object *exp*. The CAS primitive will atomically update the value of the base object b to the new value *new* if and only if the old value of b is equal to *exp*.[1] The CAS primitive always returns the old value of the object v. If the value of the object was updated to *new* meaning the response returned by the CAS primitive is equal to *exp* then we say that the CAS succeeded (otherwise we say that the CAS failed).

There is also the *double-wide-compare-and-swap* (*DWCAS*) primitive. DWCAS is essentially an extension of CAS allowing two objects to be updated simultaneously. DWCAS accepts four input parameters the expected and new values for the first object, exp_1 and new_1 and the expected and new values for the second object exp_2 and new_2. DWCAS will atomically update the value of to the first obejct to new_1 and the value of the second object to new_2 if and only if the value of the first object is equal to exp_1 and the value of the second object is equal to exp_2.[2]

In order to more clearly distinguish between CAS and DWCAS, throughout the remainder of the thesis I refer to CAS as SWCAS, (single-word compare-and-swap).

Set Abstract Data Type. This work focuses specifically on the *set* type. Without loss of generality I focus on sets that store integer values. I often refer to the values stored in the set as *keys*. Throughout the thesis k is used to denote some key. A set is initially empty, meaning it contains no keys. The set type exports the operations `insert`(k), `remove`(k), `contains`(k) where $k \in \mathbb{Z}$. Each of these operations returns a boolean response. `insert`(k)

[1] In the AMD x86-64 ISA, the CAS primitive is equivalent to the cmpxchg8b instruction and applies to single word (8-byte) registers.

[2] In the AMD x86-64 ISA, DWCAS is equivalent to the cmpxchg16b. The cmpxchg16b instruction atomically updates two adjacent 8-byte words.

returns `true` if and only if k was not in the set and returns `false` otherwise. `remove`(k) returns `true` if and only if k was in the set and returns `false` otherwise. After `insert`(k) is complete, k is present in the set, and after `remove`(k) is complete, k is absent from the set. The `contains`(k) operation returns `true` if and only if k is present in the set and `false` otherwise. The `contains`(k) operation does not change the state of the set thus throughout the thesis I often refer to `contains`(k) operations as searches.

Executions and configurations. An *event* of a process p_i in the volatile shared memory model is an invocation or response of an operation performed by p_i or a primitive applied by p_i to a base object along with its response. An event of a process p_i is sometimes referred to as an *admissible step* of p_i. A *configuration* specifies the value of each base object and the state of each process. The *initial configuration* is the configuration in which all base objects have their initial values and all processes are in their initial states.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* of an implementation I is an execution fragment where, starting from the initial configuration, each event is issued according to I and each response of a rmw event on the base object b matches the state of b resulting from all preceding events on b.

An execution $E \cdot E'$, denoting the concatenation of E and E', is an *extension* of E and we say that E' *extends* E. Let E be an execution fragment. For every process identifier k, $E|k$ denotes the subsequence of E restricted to events of process p_k. If $E|k$ is non-empty we say that p_k *participates* in E, otherwise we say E is p_k*-free.*

An operation π *precedes* another operation π' in an execution E, denoted $\pi \rightarrow_E \pi'$, if the response of π occurs before the invocation of π' in E. Two operations are *concurrent* if neither precedes the other. An execution is *sequential* if it has no concurrent operations. Two executions E and E' are *indistinguishable* to a set $\mathcal{P}$ of processes, if for each process $p_k \in \mathcal{P}$, $E|k = E'|k$. An operation $\pi_k \in ops(E)$ is *complete in* E if it returns a matching response in E. Otherwise we say that it is *incomplete* or *pending* in E. We say that an execution E is *complete* if every invoked operation is complete in E.

Well-formed executions. In the volatile shared memory model, I assume that executions are *well-formed.* An execution is well formed if an only if no process invokes a new operation before the previously operation returns. Specifically, I assume that for all p_k that participate in E, $E|k$ begins with the invocation of an operation, is sequential and there is no event between a matching response event and the subsequent following invocation. Intuitively an execution is well-formed if there is no concurrency within any single process.

Histories. A *history* H of an execution E is the subsequence of E consisting of all invocations and responses of operations. Histories H and H' are *equivalent* if for every process

p_i, $H|i = H'|i$. All of the other terminology defined for executions applies analogously to histories as well.

Lock-free Progress. We say that an implementation I is *lock-free* if it guarantees that in every execution E of I some process will always make progress by completing its operation within a finite number of its own steps. Note that lock-free progress is sometimes referred to as *non-blocking* progress.

Definition 1 (Linearizability)**.** *A complete history H is* linearizable *with respect to an object type τ if there exists a sequential history S equivalent to H such that (1) $\rightarrow_H \subseteq \rightarrow_S$ and (2) S* is consistent with the sequential specification of type τ. *A history H is linearizable if it can be* completed *(by adding matching responses to a subset of incomplete operations in H and removing the rest) to a linearizable history [40, 5].*

2.2 Non-volatile Memory Computation Model

Non-volatile memory introduces various complexities that are not defined (or are ignored for the purpose of simplicity) in the volatile memory computation model. In this section I review the non-volatile memory computation model. The model that I review here is a slightly simplified version of the *explicit epoch persistency* model from Izraelivitz et al. [41] which itself is adapted from the *epoch persistency* model presented by Pelly et al and Condit et al. [50, 18].

Memory Hierarchy. In practice system memory is organized hierarchically. In the volatile memory computation model there is no need to distinguish between any of the different levels of the memory hierarchy. For the non-volatile computation model we need to distinguish between which levels of the hierarchy are volatile and which are persistent.

At the top of the memory hierarchy we have CPU registers followed by cache memory. Most systems have several levels of cache memory (usually between two and four). For the purpose of the computation model we are not concerned with practical differences (speed and capacity) between registers and cache memory. What is relevant is the fact that both registers and all levels of cache memory are volatile. In practice some systems might define a *persistence domain* or *power-fail protected domains* that includes some or all of cache memory (such as the idea of *whole-system persistence* presented by Narayanan and Hodson [46]). In the event of a power failure, the persistence domain represents the regions of memory that can be safely persisted using residual power in the power supply or other capacitors [23]. Following cache memory, we have main memory. This thesis assumes that the main memory of the system is exclusively non-volatile memory (while this is not accurate in practice, it is trivial to map the entire address space of an application such that it is exclusively in persistent memory).

2.2.1 Persistence Model

I consider a shared memory system that can crash at any. Events in the non-volatile computation model include system crash events (sometimes we just say crash event or crash). Histories in the non-volatile computation model can include crash events along with invocations and responses. In practice a system crash can occur as a result of a power failure.

In the event of a system crash all processes crash simultaneously. If this occurs then the contents of the volatile shared memory and all individual process states are returned to their initial values but the contents of non-volatile shared memory remains persistent.

A history H containing c crash events can be partitioned into c subhistories and crashes such that $H = X_0 \cdot \perp_0 \cdot X_1 \cdot \perp_1 \cdot ... \cdot X_c \cdot \perp_c$ where $\perp_i$ is the i-th crash event and X_i is the i-th crash free subhistory of H (we slightly abuse the notation here by extending the history with an event, however since histories contain crash events we can think of $\perp_i$ as a subhistory that contains only the single crash event). As in [41] we call a crash free subhistory of H an *era* of H.

We say that an operation π was concurrent with a crash c_i if the operation was invoked in the era X_i but has no matching response in X_i. Informally we would call π a crashed operation. Note that crash events do not break the well-formedness of an execution. The events of any single process are restricted to a single era. A crash effectively destroys the processes in the era that preceded it. When a system reboots new processes are spawned. I assume that at most n processes participate in any single era of a history H. In practice the unique identifier of the processes could be the same as a processes from the previous era. If this is undesirable one might consider including the era in the unique identifier of the process. This thesis does not require the extra detail that would be provided by including the era in the process identifier so I choose not to change the process identifiers.

After a crash a recovery procedure is invoked in order to return the objects in non-volatile memory back to a consistent state. I extend the definition of well-formed executions to include the required execution of the recovery procedure.

Well-formed executions extended. In the non-volatile shared memory model, an execution E is well formed if an only if 1) for all p_k that participate in E, $E|k$ begins with the invocation of an operation, is sequential and there is no event between a matching response event and the subsequent following invocation and 2) after a system crash some process invokes the recovery procedure and no other operation is invoked until the recovery procedure completes.

Note that the recovery procedure does not return a value. In some cases the recovery procedure does not need to take any steps. In this case we say that the recovery procedure is empty. For most objects, including sets, the semantics of recovery are not well defined and it would be difficult to specify a general definition. I assume that the details of the recovery procedure are specified by the implementation of the object. I refer to all operations other than the recovery procedure as *regular* data structure operations (in the case of sets these are `insert`, `remove` and `contains`).

Modifications to base objects first take effect in the volatile shared memory. These modifications become persistent only once they are written to non-volatile memory. We can think about every base object having a volatile value and a non-volatile value (which can differ). In the initial configuration the volatile value of every base object is the same as the non-volatile value.

Base objects in volatile memory are flushed asynchronously by the process (without the programmer's knowledge) to non-volatile memory arbitrarily. I refer to this as a *background flush*. The programmer can also *explicitly* flush base objects to non-volatile memory by invoking *flush* primitives, typically accompanied by *persistence fence* primitives. An *explicit flush* is a primitive that is applied on a base object. The *explicit flush* is non-blocking meaning it can return *before* the base object has been written to persistent memory. An *explicit flush* by process p is guaranteed to written back to persistent memory only after a subsequent *persistence fence* by p. An explicit flush combined with a persistence fence is referred to as a *psync*. It is convenient for a *psync* by the processor to be a single event. For this reason I assume that *background flushes* are atomic. This assumption captures the idea that the processor can arbitrarily write any object to non-volatile memory at any arbitrary time without adding the need for the processor to perform persistence fences. [3]

Definition 2 (Persistence Event)**.** *Let E be an execution of a persistent set, we refer to any background flush, explicit flush or persistence fence as a persistence event in E.*

Persistence events are the only events that change the configuration of non-volatile shared memory. Writes and rmw events only change the value of some object in volatile memory. They do not directly modify non-volatile memory. After a crash, the first read of

[3]In Intel's 3DXPoint implementation of non-volatile memory, the *flush* primitive corresponds to the *clflushopt* instruction or the *clwb* instruction, and the *persistence fence* primitive corresponds to the *sfence* instruction. There is also a *clflush* instruction which performs a *psync*.

a shared base object will return the value of the object when it was last persisted. Thus the result of the first read on a base object b performed by some operation will always return either 1) the value of b that was last persisted by a psync before the previous crash event (this corresponds to the value of b immediately prior to the latest of either an explicit flush on b and persistence fence or a background flush on b) or 2) the initial value of b if there was never an explicit flush or background flush on b prior to the previous crash. If the recovery procedure always reads the values of all shared objects then the recovery procedure will be the only operation that directly observes the contents persistent memory. As a result, operations cannot force a read to view persistent memory which means that if an operation needs to ensure that some base object has been written to persistent memory then the operation would need to explicitly flush the base object and perform persistence fence to guarantee that the object is persistent.[4] In practice implementations of an object might store information in volatile memory to track the state of persistent memory in order to avoid the need to perform a psync. After a crash it is the responsibility of the recovery procedure restore the contents of persistent memory back to a consistent state which will involve loading data in persistent memory back into volatile memory.

I assume that psync events happen independently of RMW events and that psyncs do not change the contents of volatile shared memory (other than updating the program counter of a process). Since we cannot force an arbitrary read to return the value of the object when it was last persisted, this means that a psync has no visible side effects. These assumptions are consistent with the implementation of current hardware.

2.2.2 Durable Linearizability

Definition 3 (Durable Linearizability)**.** *A history H is durable linearizable, if it is well-formed and if ops(H) is linearizable where ops(H) is the subhistory of H containing no crash events [41].*

Intuitively durable linearizability requires that any operation that completed before a crash event will be reflected in the state of the object after recovery. Durable linearizability makes no guarantees on operations that were concurrent with a crash event. In some cases, it might be possible to complete one or more of the operations that were concurrent with

[4]In practice, performing a read on an object that is not present in the volatile cache memory would load the object from main memory (persistent memory) and one could force this effect typically with some kind of contrived behaviour. This would also involve disabling features such as prefetching. Intel's Memory Latency Checker [19] provides an example of this type of behaviour. It is unlikely that this would be desirable or common for data structures in general.

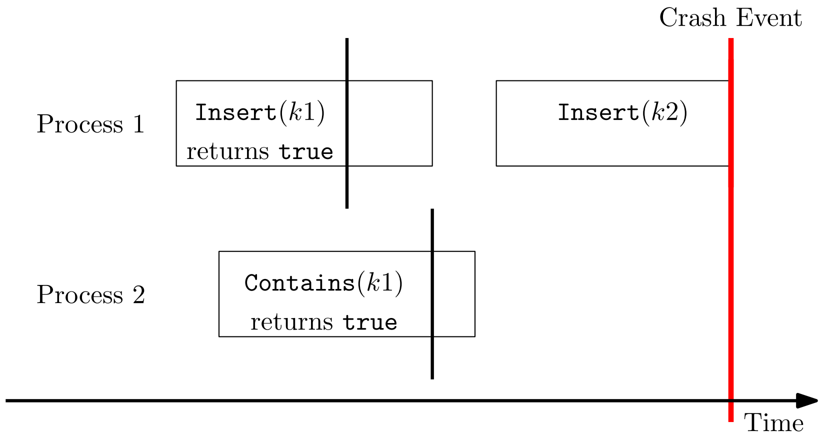


Figure 2.1: Durable Linearizability Example Execution 1. The vertical lines on the completed operations indicate the linearization point of the operation. In this example the operation `Insert`$(k2)$ by process 1 is concurrent with the crash event. The other two operations complete before the crash thus after recovery, the set must contain at least $k1$. It might also contain $k2$ if the second insert progressed far enough before the crash and the recovery procedure does not undo the insert.

a crash during or after recovery. Moreover, if an operation progressed far enough but did not complete it could still be reflected in the state of the object after recovery. There can also exist executions of durable linearizable objects in which operations that are concurrent with a crash event must be completed. An example of this for the set type is shown in Figure 2.2. This intuitive definition is more easily derived from the rephrased definition of durable linearizability from [32].

Definition 4 (Happens Before Order)**.** *Consider a history H containing the events e_1 and e_2. We say that e_1 happens before e_2 which is denoted as $e_1 \prec e_2$ if e_1 precedes e_2 in H and one of the following 4 conditions is true: (1) e_1 is a crash, (2) e_2 is a crash, (3) e_1 is a response and e_2 is an invocation, or (4) there exists an event e_3 such that $e_1 \prec e_3 \prec e_2$. This definition is the same as in [41].*

Definition 5 ($\prec$-Consistent Cut)**.** *A $\prec$-consistent cut of a history H is a subhistory W of H where if $e \in W$ and $e' \prec e \in H$ then $e' \in W$ and $e' \prec e \in W$.*

Definition 6 (Buffered Durable Linearizability)**.** *A history H containing c crash events is buffered durable linearizable if it is well formed and there exist subhistories $H_1, H_2, ..., H_{c-1}$ such that for all $0 \leq i \leq c$, (1) H_i is a $\prec$-consistent cut of X_i (the ith era of H), and (2) the history $H_0 \cdot H_1 \cdot ... \cdot H_{i-1} \cdot X_i$ is linearizable.*

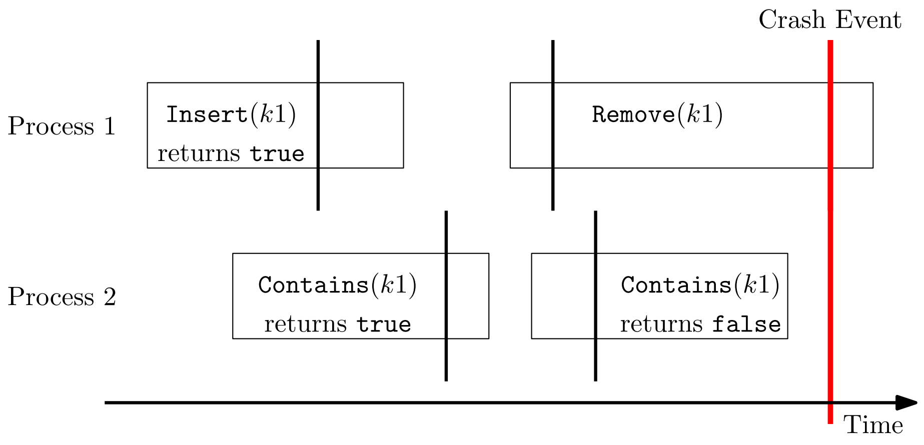


Figure 2.2: Durable Linearizability Example Execution 2. The vertical lines on the completed operations indicate the linearization point of the operation. In this example the operation `remove`($k1$) by process 1 is concurrent with the crash event. The other three operations complete before the crash. In this case, the response of the `contains`($k1$) by process 2 reflects the fact that the remove concurrent with the crash has been linearized. This means that the effects of the remove must be reflected in the state of the set after the crash. Thus, after recovery the set must not contain $k1$.

Buffered durable linearizability is a weaker correctness condition compared to durable linearizability. Intuitively, buffered durable linearizability allows for any number of operations to be lost after crash. This work does not focus on buffered durable linearizability.

2.2.3 Recovering After a Crash

It is important to understand what the expected state of an object should be after recovering from a system crash. From the previous section, we know that an operation must be persisted before it returns. However, not all operations need to be persisted. Operations that cannot affect the response of other operations do not need to be persisted. For this reason it is reasonable to focus on operations that do affect the responses of other operations. I refer to such operations as *update* operations.

Definition 7 (Update Operation)**.** *Consider an execution E of a durable linearizable object wherein the operation π was invoked. I refer to π as an update operation if and only if there exists some extension E' of E such that removing the invocation of π (and response if π completed in E) from the history H' of E' to construct the history H'' results in ops(H'') being non-linearizable.*

The definition of an *update* operation describes that π is an *update* because there exists an extension of the execution where another operation that depends on π is invoked and the response of that operation reflects the fact that π takes effect first.

Since the recovery procedure is always invoked following a crash event it is helpful to define the extension of an execution with a crash event and recovery procedure. I refer to this as the crash-recovery extension.

Definition 8 (Crash-Recovery Extension)**.** *Consider an execution E of a durable linearizable object. Let $\perp$ denote a system crash event. Let E' be $E \cdot \perp \cdot E_R$ where E_R is the sequential execution of the recovery procedure. We refer to E' as a* ***crash-recovery extension*** *of E.*

The crash-recovery extension is used to define the state of the object after recovery. It is important that the crash-recovery extension is defined in terms of a recovery procedure. There are a variety of ways that one could implement a recovery procedure for a persistent set. The implementation of the recovery procedure defines if and how operations would be undone or completed after a crash. In terms of the set abstract data type, the recovery procedure determines which keys are still in the set. To discuss how operations in a previous era can affect operations invoked in future eras it is helpful to discuss the values returned by completed operations. The return value of an operation is contained in the response of the operation.

Using a similar convention as in [1] we represent the invocation of an operation π as $inv(\pi, \mathrm{w})_p$ where p is the process that invoked the *op* and w represents the parameters of the operation. Similarly, the response of an operation is represented as $ret(\pi, \mathrm{r})_p$ where r is the value contained in the response. Without loss of generality we say that the response of π is r.

Definition 9 (Critical Persistence Event (CPE))**.** *Consider an execution E of a durable linearizable object and an update operation π that was invoked in E. Let E' be the crash-recovery extension of E. Consider a solo extension of E' wherein some process invokes and completes a new operation π' where the response of π' will be r if π was linearized before π' or r' if π was not linearized before π' and $r \neq r'$.*

For an update operation π, the persistence event f in E is the critical persistence event (CPE) of π if immediately before applying f, the response of π' is r' and immediately after applying f the response of π' is r.

Intuitively, the CPE represents the first event after which it is guaranteed that the effects of the update operation will be recovered. Figure 2.3 depicts an example execution

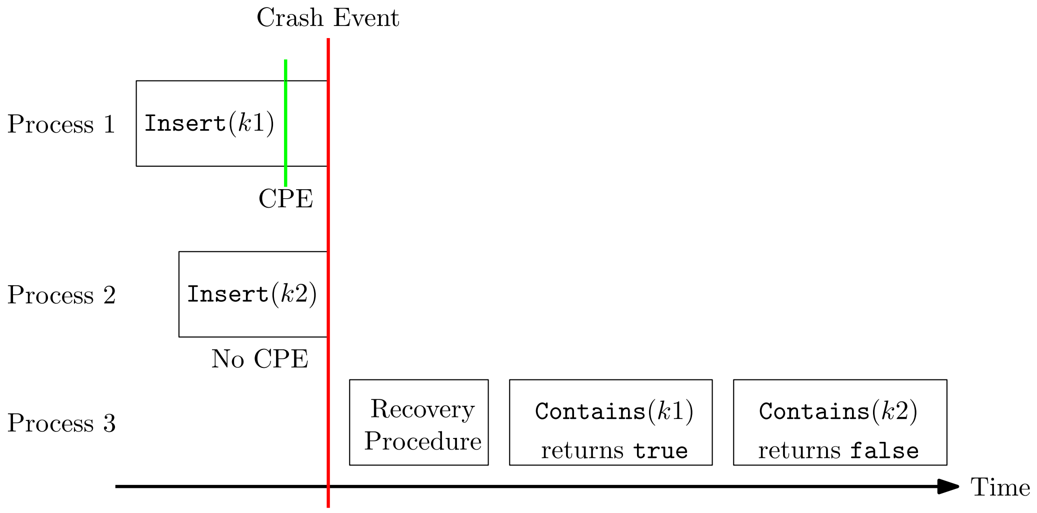


Figure 2.3: Critical Persistence Event Example. This diagram shows an example execution of a set. In this execution processes 1 and 2 both invoke insert operations with the parameters of $k1$ and $k2$ respectively. A crash event occurs before either of the inserts completes. The CPE of insert invoked by process 1 occurred at the point indicated by the green line. The insert invoked by process 2 has no CPE in this execution (alternatively we might say that the CPE of the insert invoked by process 2 never occurs before the crash event). This is reflected after the crash by the responses of the two contains operations.

of a set where two insert operations are invoked before a crash event occurs but only one of the inserts completes its critical persistence event. As a result the effects of the other insert invoked by process 2 are not recovered.

Note that the CPE of an operation cannot be an explicit flush on a base object b since an explicit flush requires a persistence fence to guarantee that that b is written to persistent memory. This means the CPE of an operation is either a background flush or a persistence fence. If the CPE of operation π is a persistence fence by process p, then p must have previously performed an explicit flush in π. We say that the base object b is *involved in the CPE* of π if the CPE is a background flush on b or if the CPE is a persistence fence where the corresponding previous explicit flush is on b.

The CPE represents the point at which the operation becomes persistent. Izraelivitz et al. briefly discuss the idea of what they refer to as a persist point [41]. A persist point represents the point at which an operation becomes persistent. The key difference is that Izraelevitz et al. defines the persist point to be some point after the linearization point of the operation. The CPE is not defined in terms of a linearization point. An operation might be linearized at some point before the CPE for example at some point which corresponds to the point at which the operation would be linearized in a volatile implementation of the same object type. Operations could also be linearized at or after the CPE of the operation. The choice of linearization point relative to the CPE can directly effect whether the implementation guarantees strict linearizability or only durable linearizability.

The CPE is also similar to the notion of a durability point defined by Friedman et al [32]. Friedman et al define durability the point of an operation as the first point in the execution when the operation becomes durable. The definition of a durability point is not specific to sets and lacks detail. With the definition of the CPE I attempted to be more explicit about some of the subtleties related to what it means for an operation to be *durable*. Specifically, the CPE is defined in terms of a recovery procedure and is explicit about the fact that the CPE always belongs to a successful update operation.

2.2.4 Strict Linearizability

Definition 10 (Strict Linearizability)**.** *A history H is* strict linearizable[5] *with respect to an object type τ if there exists a sequential history S such that $H \rightarrow S$ where $\rightarrow$ is defined by the 12 rules presented by Aguilera and Frølund in section 3.5 of [1].*

Note that the 12 rules were described for a model in which individual processes can crash. Models in which individual processes can crash subsume models in which all processes crash simultaneously (when individual processes can crash a system crash is equivalent to an individual crash event for every process where once one process crashes no other events follow except crash events by other processes). Informally, the 12 rules specify how to transform some history H into a history H' that has fewer concurrent operations and fewer crash events. I direct the reader to [1] for the formal definition of the 12 rules. Of particular interest in this thesis are the rules 6-8 which specify how to deal with crash events. Roughly speaking when we only have system crashes, given a subhistory H ending with a crash event we can construct H' such that $H \rightarrow H'$ following rules 6-8. Rule 6 states that H' can be constructed by removing the crash event from H if every operation in H completed before the crash. Rule 7 and 8 describe the fact that an operation that was concurrent with a crash may or may not take effect. Rule 7 states that H' can be constructed by removing both the crash event and the invocation of an operation that was

[5]Note that Wen and Song later presented a different correctness condition that they also called strict linearizability [56]. In this work when I refer to strict linearizability I am referring to the definition from Aguilera and Frølund [1].

concurrent with a crash event from H. Rule 8 states that H' can be constructed by inserting a matching response for an operation that was concurrent with a crash event at the crash event and removing the crash event from H. Rules 1-2 describe that the $\rightarrow$ relation is reflexive and transitive. Rules 3-5 describe how operations can be reordered. Rules 9-10 describe how to deal with operations that execute forever. Finally rules 11-12 describe how to deal with operations that abort. Informally an operation that aborts returns a special response. This thesis does not utilize the concept of aborting.

Strict linearizability is a stronger or more restrictive correctness condition compared to durable linearizability. Strict linearizability was defined for a model in which individual processes can crash. For models that only allow system crashes (as is the case in this thesis) any strict linearizable implementation is also durable linearizable.

Intuitively, strict linearizability requires that any operation that is concurrent with a crash event either takes effect before the crash event or does not take effect at all. This means that unlike durable linearizable objects, an operation that is concurrent with a crash should not be completed after the crash.

Figure 2.4 shows an execution of a set that violates strict linearizability. In this example the `insert` by process 1 takes effect after the crash. To show that this execution is not strict linearizable I will describe how we can use the rules defined in [1]. Call the `insert` by process 1 π_1, the *contains* by process 2 π_2, and the contains by process 3 π_3 and π_4. Assuming individual processes can crash, the event of process p crashing is represented as $crash_p$. The history H of this execution is as follows:

$$inv(\pi_2, k1)_2 \cdot inv(\pi_1, k1)_2 \cdot ret(\pi_2, \texttt{false})_2 \cdot crash_1 \cdot inv(\pi_3, k1)_3 \cdot ret(\pi_3, \texttt{false})_2 \cdot inv(\pi_4, k1)_3 \cdot ret(\pi_4, \texttt{true})_3.$$

To construct the sequential history such that $H \rightarrow S$ we begin by applying rule 8 and insert a response for π_1 this results in the following history:

$$inv(\pi_2, k1)_2 \cdot inv(\pi_1, k1)_2 \cdot ret(\pi_2, \texttt{false})_2 \cdot ret(\pi_1, \texttt{true})_2 \cdot inv(\pi_3, k1)_3 \cdot ret(\pi_3, \texttt{false})_2 \cdot inv(\pi_4, k1)_3 \cdot ret(\pi_4, \texttt{true})_3.$$

Next we apply rule 4 to reorder the invocation of π_2 and π_1 which results in the following history:

$$inv(\pi_2, k1)_2 \cdot ret(\pi_2, \texttt{false})_2 \cdot inv(\pi_1, k1)_2 \cdot ret(\pi_1, \texttt{true})_2 \cdot inv(\pi_3, k1)_3 \cdot ret(\pi_3, \texttt{false})_2 \cdot inv(\pi_4, k1)_3 \cdot ret(\pi_4, \texttt{true})_3.$$

At this point we cannot apply any other rules and we have a sequential history S however S is not consistent with the sequential specification of a set since the contains operations π_3 and π_4 have different responses but the only update operation in the history completes before either is invoked.

2.2.5 Strict Linearizability Intuition Versus Definition

In chapter 1 I mentioned a motivating example for strict linearizability utilized by Aguilera and Frølund. The example described a scenario in which a missile launch is initiated but the missile fails to launch and it is emphasized that the missile should not launch at some later time. The missile launch represents some critical, irrevocable event that we want to guarantee happens before the launching system crashes or never happens at all. However, it is not clear that this intuition is actually captured by the definition of strict linearizability. Strict linearizability is defined in terms of histories. The notion of an irrevocable event like the missile launch cannot be captured by histories. Continuing with the missile launch example, consider the case where the process that invoked the missile launch crashes but progressed far enough to allow other processes to help complete the launch. If some other process does help complete the launch then the missile would be launched after the crash. When this is expressed as a history where the crash is removed via the $\rightarrow$ relation, it would appear that the missile launched prior to the crash. The second process that helps launch the missile after the first process crashed could be invoked at a much later time. Since histories do not capture the concept of time, the definition of strict linearizability cannot restrict implementations such that behaviour like launching the missile *years later* is forbidden.

This issue is especially important for the model used in this thesis where the system invokes a recovery procedure following a system crash and the recovery procedure must complete before other operations can be invoked. The recovery procedure does not return a value. Thus, the response of the recovery procedure does not reflect whether or not some operation was completed prior to the crash. If the recovery procedure helps linearize an operation that did not complete before the crash then that operation will still appear to take effect effect prior to the crash in an equivalent history where the crash event is removed (via the $\rightarrow$ relation). This allows for the same problem of launching the missile after the crash which could lead to the “catastrophic” behaviour that Aguilera and Frølund wanted to avoid with strict linearizability. Moreover, the recovery procedure could be invoked at a much later time compared to when the system crash occurred. Thus, if the recovery procedure helps linearize operations then operations are not really limited to take effect by the time of the crash (if the operation does crash).

In the model used in this thesis a critical, irrevocable event is represented by the CPE of an operation. The CPE represents the point after which the effect of the operation will be recovered. The behaviour where the missile can be launched after the crash is only possible if an implementation allows operations that have a CPE to be linearized after their CPE. As a result, if we want to satisfy the intuition behind strict linearizability, we need to refer to the CPE of an operation which is not captured in a history. Modifying the definition of strict linearizability to include the notion of a CPE would require persistence events to be part of histories. This would introduce unwanted complexity. Rather than modifying the definition of strict linearizability, I instead choose to restrict the classes of strict linearizable implementations that I consider in this thesis as follows: for all classes of strict linearizable implementations considered in this thesis, operations that have a CPE will always take effect at or before their CPE.

2.2.6 Durable versus Strict Linearizabile Sets

When the system recovers from crash events by invoking a recovery procedure that must complete before other operations can be invoked, there is no meaningful difference between strict linearizability and durable linearizability when the classes of strict linearizable implementations allow operations that have a CPE to take effect after their CPE. Durable linearizability captures the idea that the effects of an operation can be visible after a crash even when the operation has no response if the operation progressed far enough. With durable linearizability processes in later eras could help complete crashed operations from a previous era. This conflicts with strict linearizability. If the effects of an operation are visible after a crash then by definition the effects of the operation are also visible during the execution of the recovery procedure. Neither strict linearizability nor durable linearizability prevents the recovery procedure from helping to complete crashed operations in a previous era. As mentioned in the previous section, this is behaviour does not satisfy the intuition provided by Aguilera and Frølund and provides further justification for the choice of restricting the classes of strict linearizable implementations considered in this thesis.

2.2.7 Other Related Correctness Conditions

There are some related correctness conditions for persistent data structures that I do not consider in this thesis. Guerraoui and Levy presented the safety condition of *persistent atomicity* which ensures that the state of an object will be consistent after a crash [34]. Persistent atomicity does not guarantee locality. Berryhill et al. presented an alterna-

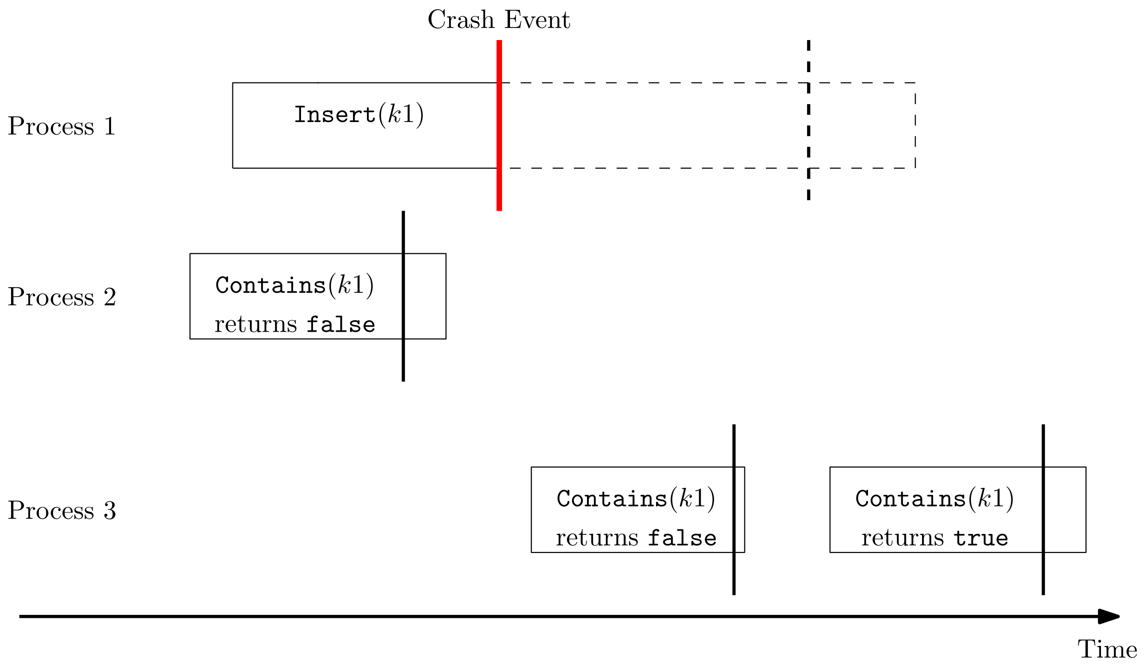


Figure 2.4: Strict Linearizability Illegal Execution Example. The vertical lines on the completed operations indicate the linearization point of the operation. In this example I assume the model used in [1] where individual processes can crash. Process 1 crashes before completing the operation Insert($k2$) but the insert it is linearized after the crash meaning this execution does not satisfy strict linearizability.

tive to persistent atomicity which does guarantee locality which they called *recoverable-linearizability* [8]. Recoverable-linearizability does not always preserve the program order after a crash. Both durable linearizability and strict linearizability guarantee locality and preserve program order after a crash. Izraelevitz et al. noted that the issues of persistent atomicity and recoverable-linearizability are related to the fact that the models used by Guerraoui and Levy as well as Berrhill et al. allow individual processes to crash, recover and continue execution. This differs from the model used in this thesis in which all processes crash together.

2.2.8 Relation to Existing Models

The non-volatile shared memory model used in this thesis is very similar to the model used by Izrealevitz et al. in [41] with some minor changes that make discussing persistence more convenient. Models in which individual processes can crash also subsume models in which all processes crash simultaneously. Models in which individual processes are allowed to crash and independently recover introduce added complexity that I do not explore in this thesis.

Chapter 3

Background

In this chapter I will provide some necessary background information regarding metrics used to compare persistent objects. We will also present some relevant existing work. Specifically I discuss existing persistent sets as well as universal constructions for creating durable linearizable implementations of volatile objects.

3.1 Complexity Measures

We would like a way to compare different implementations of persistent concurrent objects. Traditional metrics used to compare implementations of volatile concurrent objects are not very useful we want to examine the overhead related to utilizing persistent memory. This is because an implementation of a persistent concurrent object is likely to share many similarities to an implementation of a volatile object of the same abstract data type. In particular, it is important to note that an implementation of a persistent concurrent object will still require a mechanism for synchronization in volatile memory.

The need for performing psyncs is unique to persistent objects. A psync is required to ensure that data is written to persistent memory. Psyncs represent the usage of expensive fence operations. Recent work has referred to the required number of psyncs per operation when comparing persistent objects [58].

Another unique aspect of persistent objects is the need for a recovery procedure. The recovery procedure represents a period of downtime for the persistent object. If we consider the fact that a system can crash multiple times, it is desirable to have an efficient recovery procedure that can complete quickly. Along with psync complexity, I also consider the recovery complexity as measures for comparing persistent objects.

Psync Complexity. Programmers write data to persistent memory through the use of psyncs. A psync is an expensive operation because it requires the use of fences. Cohen et al. [17] prove that update operations in a durable linearizable lock-free algorithm must perform at least 1 psync. This means that update operations in durable linearizable algorithms cannot avoid performing a psync without sacrificing lock-free progress. In some implementations of persistent objects, read-only operations might perform psyncs. There is a clear focus in existing literature on minimizing the number of pysncs per data structure operation [24, 58]. These factors suggest that psync complexity is a useful metric for comparing implementations of persistent objects.

Recovery Complexity. After a crash, a recovery procedure is invoked to return the objects in persistent memory back to a consistent state. No new data structure operations can be invoked until the recovery procedure has completed. Ideally I would like to minimize this period of downtime represented by the execution of recovery procedure. Moreover, the recovery procedure is usually assumed to be performed by only a single process [24, 58]. A recovery procedure could spawn new processes, however, I am not aware of any existing persistent objects that feature a concurrent recovery procedure. The system can crash during the execution of the recovery procedure. We use the asymptotic time complexity of the recovery procedure as another metric for comparing durable linearizable algorithms. The recovery procedure can be empty, meaning it performs no instructions.

3.2 Related Persistent Sets

In this section I will describe some existing hand-crafted implementations of persistent sets. Specifically, I examine the Link-and-Persist technique of David et al. [24] and the Link-Free and SOFT algorithms from Zuriel et al. [58]. Table 3.1 summarizes some important aspects of these techniques.

3.2.1 Link-and-Persist Technique

David et al. describe a technique for implementing durably-linearizable link-based data structures called the Link-and-Persist technique [24]. Using the Link-and-Persist technique, whenever a link in the data structure is updated, a single bit mark is applied to the link which denotes that it has not been written to persistent memory. The mark is removed after the link is written to persistent memory. We refer to this mark as the *persistence mark* or *persistence bit*. Updating the persistence mark requires the use of atomic synchronization primitives. Both DWCAS and CAS can be used to update the persistence mark. The

Technique	**Correctness Condition**	**Maximum Psyncs Per Update**	**Maximum Psyncs Per Contains**	**Recovery Step Complexity**
Link & Persist	Strict	Unbounded	2[†]	Bounded by size of data structure
Link-Free	Strict	1*	1	Allocator Dependant
SOFT	Durable	1*	0	Allocator Dependant

Table 3.1: Existing Techniques for Implementing Persistent Sets. *Each process will perform at most one psync per update however every process can perform a psync to persist the same update. [†]This assumes that the successor links in a node can be persisted with a single psync.

Link-and-Persist technique requires a helping mechanism to ensure persistence. If the result of an operation depends on a marked link then that operation must persist the link and remove the mark. Data structure links are never intentionally explicitly flushed once the persistence mark is removed. However, since a link cannot be flushed and unmarked simultaneously, lock-free implementations allow for the possibility of explicitly flushing an unmarked link. Unmarked links can also be written to persistent memory as a result of background flushes. This means that persistent memory can contain both (persistence) marked and unmarked data structure links. Figure 3.1 shows an example of an `insert` operation on a persistent linked list using the Link-and-Persist technique. The Link-and-Persist technique requires a psync after any data structure link is updated. If multiple links need to be updated as part of a data structure operation then each one would require a psync.

Note that the Link-and-Persist technique is specifically designed for lock-free implementations of persistent objects. While there is nothing preventing the usage of the Link-and-Persist technique in a lock based implementation, there would be no need for a persistence mark since operations can simply hold a lock until the updated links are persisted.

The Link-and-Persist technique has two main benefits. First, the algorithm is quite simple and can be easily integrated into many implementations of existing volatile objects. Persistent objects implemented using the Link-and-Persist technique have very simple recovery procedures and allow for empty recovery procedures. David et al. also presents the link-cache algorithm which I do not consider due to the fact that it produces buffered durable linearizable implementations without adding an uncommon constraint on clients of the data structure (specifically the constraint requires that clients of the data structure cannot consider an update as completed when it returns, which is not very intuitive). Note that this technique of marking to indicate whether some data has been written to persistent

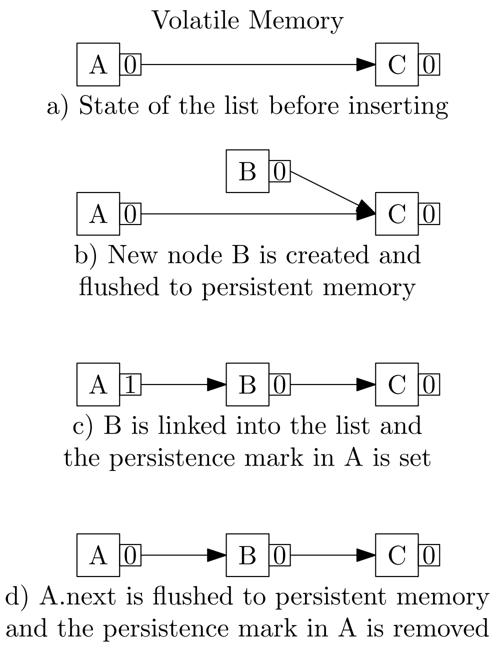

Volatile Memory
Persistent Memory
A 0
C 0
a) State of the list before inserting
B 0
b) New node B is created and flushed to persistent memory
A 1
c) B is linked into the list and the persistence mark in A is set
d) A.next is flushed to persistent memory and the persistence mark in A is removed

Figure 3.1: Sequential update of a linked list using the Link-and-Persist technique. The left column shows the state of the list in volatile memory. The right column shows the state of the list in persistent memory.

memory was also used by Wang et al. to implement a persistent multi-word CAS [54].

3.2.2 Link-Free Sets

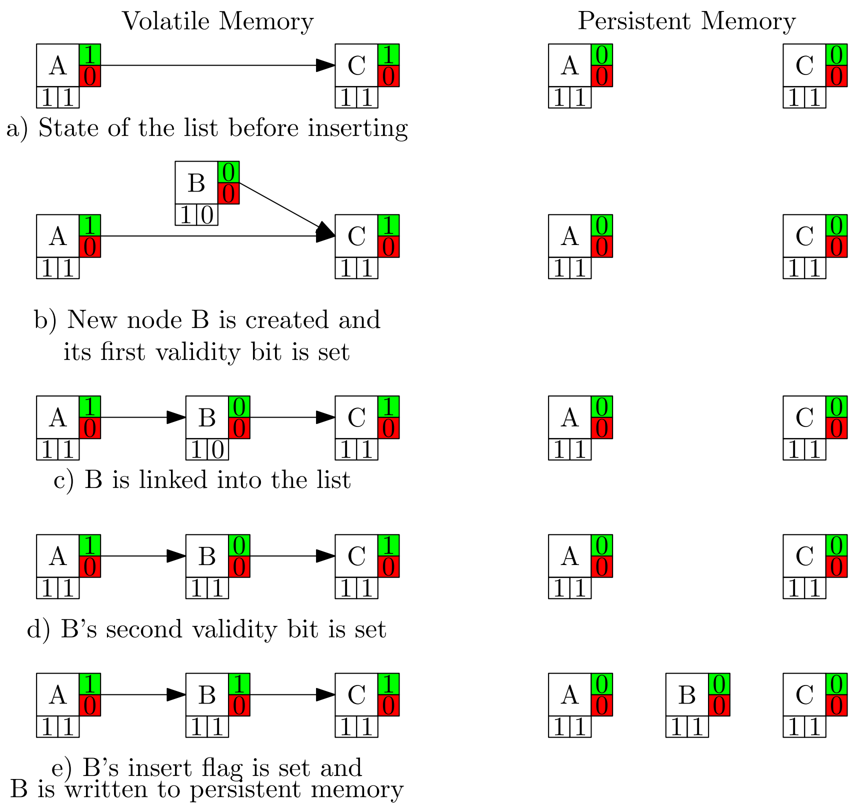
Zuriel et al. take a very different approach in their Link-Free sets algorithm [58]. The Link-Free algorithm is a durably-linearizable implementation of a persistent set. More specifically, it is an implementation of a persistent linked list. As the name suggests, the Link-Free algorithm does not persist data structure links. Instead, the Link-Free algorithm persists metadata added to every node. Every node contains extra metadata in the form of two validity bits. In practice, these validity bits are stored in a byte sized field. A node is considered in the list if and only if both validity bits match and the node is not marked as logically deleted. During an `insert`, when a node is first created one of the validity bits is flipped to invalidate the node while it is in a transient state before being inserted into the list. After linking the new node into the list, the other validity bit is flipped indicating that the node is valid. `Remove` operations will only modify the validity bits when helping concurrent `insert` operations. Zuriel et al. employ a technique similar to the persistence bit in the Link-and-Persist technique. The Link-Free algorithm uses two separate flags to indicate when a psync was performed after inserting a node and after logically deleting a node. In practice these flags are stored in separate byte sized fields. During an `insert`, the Link-Free algorithm requires a psync after the second validity bit is flipped marking the node as valid. During a `remove` a psync is required after the node is marked as logically deleted.

This algorithm is effective in reducing the number of required psyncs per operation. However, recovery procedure required by the Link-Free algorithm is non-trivial. Since the Link-Free algorithm does not explicitly persist data structure links, the list must be completely rebuilt during recovery. Moreover, the recovery procedure must traverse non-empty memory pages in order to find all of the valid nodes that belong in the list. If the data structure is very large or crashes are frequent this could be problematic.

3.2.3 Sets with an Optimal Flushing Technique

Along with the Link-Free algorithm, [58] also presents the Sets with an Optimal Flushing Technique (SOFT). This algorithm is very similar to the Link-Free algorithm with the added benefit of supporting persistence-free searches.

As in the Link-Free algorithm, the SOFT algorithm does not persist data structure links and instead persists metadata added to each node. The primary difference between



Volatile Memory
Persistent Memory
A
B
C
1
0
a) State of the list before inserting
b) New node B is created and its first validity bit is set
c) B is linked into the list
d) B's second validity bit is set
e) B's insert flag is set and B is written to persistent memory

Figure 3.2: Sequential Update of a Linked List Using the Link-Free algorithm. The left column shows the state of the list in volatile memory. The right column shows the state of the list in persistent memory. Green cells represent the insert flag and red cells represent the delete flag. The two validity bits are shown under every node.

the Link-Free algorithm and SOFT is that SOFT uses two different representations for every key in the data structure. Each key is stored in a volatile node and in a persistent node. Persistent nodes are explicitly flushed to persistent memory whereas volatile nodes are not. Every volatile node contains a pointer to the corresponding persistent node. The volatile node also contains pointers to its (volatile node) descendants. Two bits are stolen from these volatile descendant pointers in order to mark the volatile node in one of four states. The state of a volatile node is either inserted, intend to insert, deleted or intend to delete. Operations can read the state of the node in order to minimize the number of psyncs required. The persistent nodes also contain three validity bits. Two of these validity bits function similarly to the validity bits of the Link-Free algorithm. If these two bits are set then the persistent node is in a consistent state, otherwise the node is in a transient state while it is being inserted. The other validity bit is used to mark the persistent node as deleted. Initially all three validity bits are zero. The first validity bit is set after the persistent node is created by before the key is stored in the persistent node. After the key is stored in the persistent node the second validity bit is set. Figure 3.3 shows an example of the SOFT algorithm used to update a linked list.

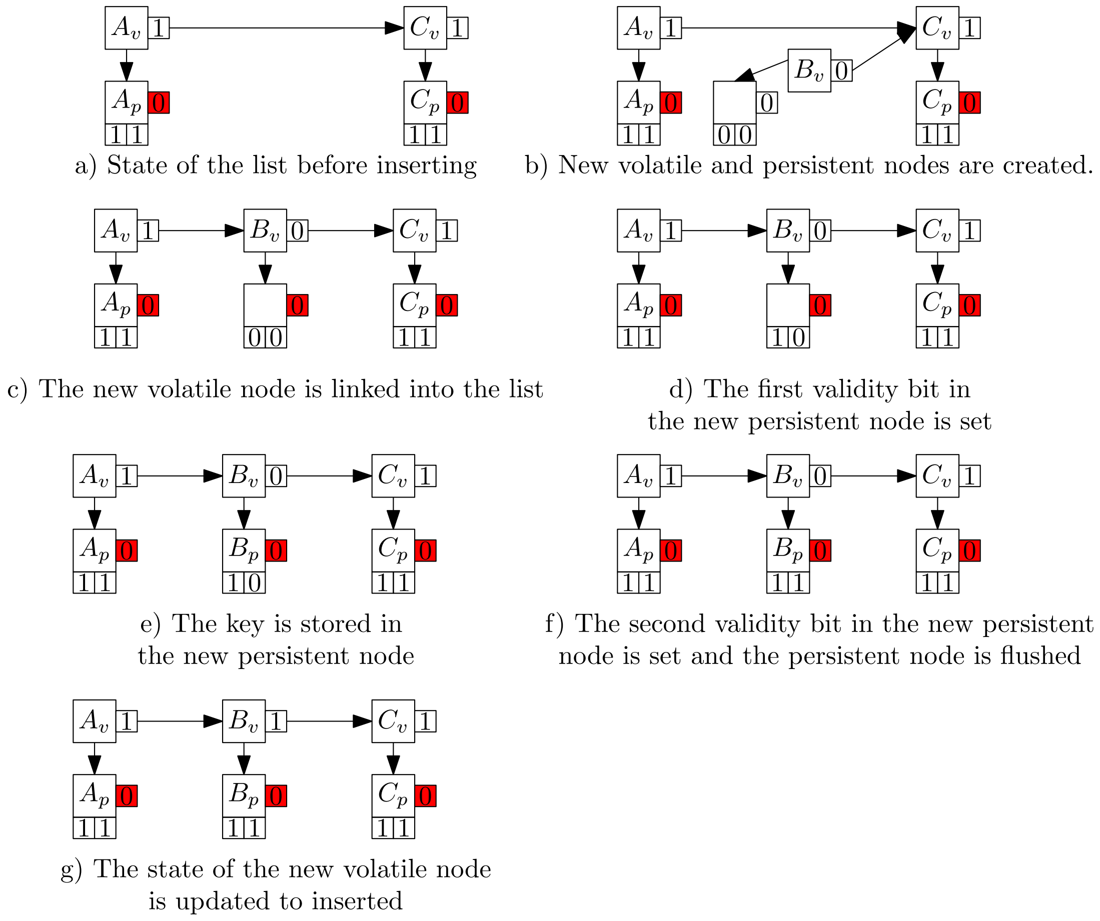
Compared to the other related algorithms, the SOFT algorithm achieves the best bounds on the number of required psyncs per operation. It is also the only algorithm that requires zero psyncs for searches. Unfortunately, the recovery procedure required by SOFT has the same issues as the Link-Free algorithm.

3.3 Related Transforms and Universal Constructions

Algorithms that automatically convert linearizable objects into durably-linearizable objects generally perform worse in practice compared to hand-crafted alternatives. This thesis focuses on hand-crafted persistent sets. I will also describe some examples of algorithms capable of producing durable linearizable objects from (transient) linearizable objects.

Transform from Izraelevitz et al. [41]

Izraelevitz et al. presented a universal construction to transform a concurrent multi-object program written for release consistency and transient memory, into an equivalent program for explicit epoch persistency. This universal construction can produce both durable and buffered durable linearizable objects. The transform relies on a concrete memory model. We refer the reader to [41] for the model. The model used in this work could be used to



a) State of the list before inserting

b) New volatile and persistent nodes are created.

c) The new volatile node is linked into the list

d) The first validity bit in the new persistent node is set

e) The key is stored in the new persistent node

f) The second validity bit in the new persistent node is set and the persistent node is flushed

g) The state of the new volatile node is updated to inserted

Figure 3.3: Sequential update of a linked list using the SOFT algorithm. This figure shows only the state of volatile memory. Initially persistent memory contains only the persistent nodes A_p and C_p. B_p is written to persistent memory in step f. Keys with the subscript v indicate volatile nodes and the subscript p indicate persistent nodes. The red cells represent the validity bit used for marking the persistent node deleted. The state of volatile nodes is represented as an integer 0-4.

produce a similar concrete memory model. It is still relevant to understand this universal construction to see why the resulting objects would perform worse than hand-crafted alternatives.

The universal construction follows six rules. (1) Immediately after every store, flush the written value. (2) Immediately before every store-release S_r perform a persistence fence then immediately after S_r flush the object that S_r was applied on. (3) Immediately after every load-acquire flush the written value then perform a persistence fence. (4) Immediately before any CAS perform a persistence fence and immediately after any CAS flush the written value then perform a persistence fence. (5) Do not flush or fence on loads. (6) Immediately before the returning from an operation perform a psync. Extra rules are required to achieve buffered durable linearizability. The general idea with this universal construction is to ensure that every write is persistent before the value is read and operations are persisted before returning.

In most cases it is not necessary to persist every write. As a result, objects produced by this universal construction might perform unnecessary psyncs.

Order Now, Linearize Later

The Order Now, Linearize Later (ONLL) universal construction from Cohen et al. [17] transforms a deterministic object and produces a lock-free durably-linearizable implementation of the object.

This universal construction is especially interesting because it produces an implementation in which read-only operations do not perform psyncs. ONLL breaks an operation into three stages, the order stage in which the linearization order of operations is established, the persist stage in which a psync is performed and the linearization stage in which the operation is linearized. ONLL relies on a shared volatile lock-free queue as well as per-process persistent logs originally described in [16]. The state of the object is captured by the volatile queue representing the execution trace. Update operations are first added to the volatile queue. The update operation and any of its dependencies that have not been written to persistent memory are added to the persistent log of the process that invoked the update. Finally the update sets a flag indicating that it has been linearized. If a system crash occurs the volatile queue must be reconstructed from the persistent logs of every process.

ONLL has two clear drawbacks. First, it relies on a shared global queue, which represents an obvious point of contention. Second, recovery is non-trivial. The persistent logs from every process need to be examined to reconstruct the queue.

NVTraverse

Friedman et al. presented a transform for converting a class of data structures which they call traversal data structures to durable linearizable data structures [31]. Traversal data structures are node-based tree data structures where operations on the data structure first perform a traversal followed by a critical path where updates are performed. The main benefit of NVTraverse is that no flushes are ever performed during the traversal of the data structure. During a traversal NVTraverse keeps track of all of the fields that were read and then flushes them after the traversal using only one persistence fence. During the critical path NVTraverse requires a flush for every shared variable that is read and every write or RMW. A fence is required for every write or RMW on a shared variable and before every return. This approach is actually quite similar to the Link-and-Persist technique, however while NVTraverse successfully avoids excess persistence fences, it requires a significant amount of flushes. In general, the results presented by Friedman et al. support the notion that hand-crafted persistent data structures generally perform better compared to implementations produced by transforms (including NVTraverse).

3.3.1 Mirror

Recently Friedman et al. have presented Mirror [33]. Mirror is an automatic transform that converts a linearizable lock-free data structure to a durable linearizable lock-free data structure. Mirror relies on maintaining two copies of the data structure, one which remains transient and one that is persisted. Reads are executed on the transient data structure which is stored in DRAM. Their results showed that Mirror often performed better than the work of Zuriel et al. when the transient copy of the data structure is stored on DRAM. However, the experiments used only up to 16 threads across two NUMA nodes and many of the results use only 8 threads. I do not evaluate Mirror in this thesis but it would be interesting to examine how it performs on a system with more NUMA nodes. Mirror requires that data structures supply a *tracing* operation which can traces all the reachable data from a set of roots which are provided as a parameter. After tracing reachable data, Mirror must reallocate both the volatile and non-volatile copies of the data structure which can result in longer recovery time. Note that if a volatile memory allocator allows specifying the address of the allocated object then Mirror can reallocate only the volatile copy of the data structure however this is not realistic in practice.

3.4 Transactional Approaches

While this thesis does not explore the use of transactions I will note some examples of other works that have proposed using transactional-memory (TM) with NVM to achieve persistent data structures. Avni and Brown presented the first hybrid transactional-memory (TM) for systems with NVM which they called *PHyTM* [6]. PHyTM uses redo logging to facilitate recovery after a crash. PHyTM was inspired by the work of Avni et al. which presented an algorithm called PHTM [7]. PHTM relied on a proposed change to Intel's hardware-transactional-memory (HTM) that would allow a single bit to be flushed to NVM as part of a transaction commit.

Kolli et al, Volos et al. and Coburn et al. each presented methods for using transactions to interface with NVM. Kolli et al. presented a transaction system for NVM that relies on deferring commits to minimize constraints on the order of writes to NVM (which they refer to as *persist dependencies*) [43]. Volos et al. presented a system for exposing persistent memory to user-mode programs which they call *Mnemosyne*. Mnemosyne relies on the use of a transaction system to facilitate in-place updates of a data structure [53]. Coburn et al. presented *Non-volatile Memory Heaps* (*NV-heaps*) which is a system used to implement persistent objects [15]. NV-heaps utilizes memory-mapped structures and relies on transactions to ensure changes to these structures a persisted in a well-defined manner.

3.5 Logging vs. Log-free

The existing hand-crafted persistent sets that I have discussed do not utilize logging. The ONLL universal construction utilizes per-process persistent logs. While logging provides a simple approach to achieving durable linearizability, it has some undesirable overhead. In the case of ONLL the use of logging makes recovery more complicated since the persistent logs of all processes must be searched to reconstruct the execution trace. The use of a shared log would represent a single point of contention which would be a bottleneck for update operations. Moreover, logs are typically finite which means other data along with the log must be persisted in order to recover the object. These factors motivate implementations of persistent objects that do not utilize logging.

3.6 How Much Data to Persist?

The existing literature reveals an interesting question related to persistent sets - how much data should we persist? We must at least persist the keys that were in the set. More specifically, to guarantee durable linearizability we must be able to recover any key that was inserted and not removed by operations that completed before the crash and we must not recover keys that were removed by operations that completed before the crash. In an unrealistic setting, we could achieve this if every possible key has a one-to-one mapping with a unique address in persistent memory and we reserve a single value to represent that a specific key is not in the set. In practice, it is likely that we would want to also persist some metadata along with the keys such that we do not require this type of mapping. This is essentially the approach taken by both the Link-Free and SOFT algorithms. Both of these algorithms persist a small amount of metadata as well as the keys stored in the set. Notably these algorithms do not persist any of the structure (links) of the data structure. Algorithms that do not persist the links of a data structure must reconstruct the data structure after a crash. During recovery, these algorithms will need to determine all of the keys that still belong in the data structure. In the worst case this involves traversing the entire persistent memory space. It is possible that some memory management mechanisms could be added to condense valid keys to a smaller portion of persistent memory but this would require extra psyncs. To the best of my knowledge, this approach has not been explored, however, I believe that it is unlikely that this approach would be beneficial since the added overhead would likely severely reduce performance.

In contrast, algorithms that persist the links of the data structure can avoid the need to fully rebuild after a crash. This is the case with the Link-and-Persist technique. There is a drawback to persisting the links of a data structure. To the best of my knowledge there is no existing algorithm that persists data structure links and has an upper bound of 1 psync per update operation. There is also no existing algorithm that persists links and has persistence-free searches. While the former is likely intrinsic to persisting data structure links, the latter is possible. In chapter 6 I present a persistent set that persists data structure links and allows for persistence-free searches.

These examples reveal a possible trade-off between recovery complexity and runtime performance. Persisting more data requires performing more psyncs but allows for less complex recovery procedures leading to better worst case recovery complexity. On the other hand, persisting less data can reduce the number of required psyncs but requires recovery procedures with very poor worst case recovery complexity.

Chapter 4

Strict Linearizable Sets and Persistence-free Reads

Informally, strict linearizability requires that an operation takes effect before a crash or not at all. This is problematic if we want the set to support read-only operations operations that do not require flushes or persistence fences. Cohen et al. pointed out that for a lock-free object persistence free reads would require linearizing update operations at some point after the update is persisted [17]. Intuitively this makes sense since linearizing an update operation prior to or at the point where the update is persisted would be problematic if the response of a concurrent search depends on an update being persisted. In this chapter, I provide a proof showing that it is impossible to implement a strict linearizable lock-free set for which read-only operations do no perform any explicit flushes or persistence fences. Note that this impossibility result holds only for the classes of strict linearizable implementations considered in this thesis where operations that have a CPE are guaranteed to take effect at or before their CPE.

Definition 11 (Decided Operation)**.** *Consider an execution E of a durable linearizable set. The response of a pending operation π is* ***decided*** *in E if for every possible crash-free extension of E the response of π is the same value v. We say its* ***decided response*** *is v.*

In an implementation of a volatile set an operation is decided when it is linearized. For implementations of persistent sets, it is sometimes helpful to think of an operation as having both a linearization point chosen to guarantee durable (or strict) linearizability as well as what we might call a volatile linearization point. The intuition being that the volatile linearization point is the point at which the operation takes effect in volatile memory. This would correspond to the linearization point of an operation in an implementation of a volatile set. An operation would be decided at its volatile linearization point.

Within the context of sets, insert and remove operations that return `true` are considered update operations. Typically, one might refer to these as *successful* update operations where insert and remove operations that return `false` are considered *unsuccessful* update operations.

For every successful update operation π, we can identify the CPE e of π. Intuitively, if a crash event happens *before* e, then the update π will *not be recovered.* This means if we perform an identical update after recovery, it will *succeed.* On the other hand, if a crash happens *after* e, then π *will be recovered.* So, if we perform an identical update after recovery, it will *fail.* Since the CPE is defined for successful update operations if in some execution, the CPE exists for an update π then the CPE of π exists at some point after π is decided and the first of either the response of π or a crash event.

One might think that since the CPE is defined only for successful update operations that this would be problematic for preserving the history of unsuccessful update operations or search operations. The CPE is defined for successful update operations because only successful update operations will be reflected in the state of the object after recovery. Unsuccessful update operations and searches do not really have a CPE however, they can help perform the CPE of a concurrent update. If we wanted to assign a CPE to every operation then the CPE of a search or an unsuccessful update operation or would correspond to the CPE of some successful update operation.

If a search for the key k returns `true` then k must have been inserted into the data structure (and not removed) before the search was linearized. We can construct a similar case where the search returns `false` because k was removed by an update operation that was linearized before the search is linearized. In both cases, the response of the search is determined by the update operation that inserted/removed k and it reflects the fact that the CPE of the update has occurred. Similar to searches, unsuccessful update operations do not modify the data structure so we can construct the same examples for unsuccessful update operations. Searches that return `false` and unsuccessful `remove` operations where the key to remove has never been inserted are edge cases. In these cases we can think of the initialization of the data structure as the corresponding successful update. (This is obviously an abuse of the definition of an update. Alternatively we could say that the unsuccessful remove has no corresponding CPE but this would be inaccurate for data structures that use mechanisms like sentinel nodes).

The corresponding CPE for searches or unsuccessful update operations can be prior to the invocation of the operation. This is easiest to see in the sequential case where the CPE of every successful operation will always occur before the invocation of another operation.

For a more concrete example, consider an insert operation of persistent linked-list where we want to persist data structure links. Suppose that the list is initially empty and process p runs under no contention and invokes the insert operation π. p will allocate a new node and perform a psync to write the contents of the node to persistent memory. It will then link the new node into the list and perform another psync to write the updated link to persistent memory. This psync is the CPE of π and in this example the CPE of π was a event performed by process p. However, the CPE of π does not need to be a event by p. Suppose p sleeps immediately after it links the new node into the list. Another process p' could help persist the link updated by p by performing a psync. In that case, the CPE of π is an event by p'.

Definition 12 (Persistence-Free Searches)**.** *We say that an implementation of the set abstract data type offers persistence-free searches if there is no execution of the set in which a process that invoked a search, π, performs an explicit flush or persistence fence between the invocation and response of π.*

Definition 13 (Persistence-Help-Freedom)**.** *An execution of a set is persistence-help-free if the following conditions hold. If process p performs an explicit flush on base object b in an operation π, then p must have previously written to b in π. If p performs a persistence fence in an operation π, then p must have previously performed an explicit flush on some base object in π. A set is persistence-help-free if all executions of the set are persistence-help-free.*

Theorem 14. *There exists no strict linearizable lock-free set with persistence-free searches.*

Proof. Consider a strict linearizable lock-free set implementation I. Assume for the purpose of showing a contradiction that every search is persistence-free in every execution of I. Starting from an empty set, construct an execution E of I as follows: Let process p_u invoke an `insert` operation to insert the key k then let p_u progress until immediately prior to the CPE of π_u then sleep. Consider a search operation π_c looking for the key k. This means that π_c does not commute with π_u. The response of π_c depends on whether or not π_u was linearized. Consider the following cases for possible linearization points of π_u.

Case 1: Suppose π_u is linearized at some configuration or explicit event prior to its CPE. Suppose process p_c invokes the search operation π_c. Since the next event of π_u would be its CPE and it was linearized prior to its CPE this means that the linearization point of π_u occurred before the invocation of π_c. Let p_c complete π_c in E. Since π_u was linearized the response of π_c must return `true`. Now suppose that a crash event occurs. Let E' be the crash-recovery extension of E and let π'_c be a search operation identical to π_c. Let E'' be a solo extension of E' by any process in which π'_c is invoked and completes. Since the CPE of π_u never occurred in E (or E') this means that π'_c will return `false` in E''. Let H_1 be the history of E'' and let H' be constructed such that $H_1 \rightarrow H'_1$. If we follow rule 7 of [1] and remove both the invocation of π_u and the crash event then H'_1 will contain the invocation of π_c and corresponding response followed by the invocation and response of π'_c (I omit the recovery procedure for simplicity since it does not return a value). There is no equivalent sequential history for this construction of H'_1 since in a sequential history any two consecutive search operations must return the same value. Alternatively if we instead construct H'_1 by inserting a response for π_u then H'_1 would include all three operations. Since π_c returned `true` the response inserted for π_u must indicate that it returned `true`, otherwise the response of π_c would not be consistent with the sequential specification of a set for any possible sequential history equivalent to H'_1. If the response inserted for π_u indicates that it returned `true` then it must be linearized prior to π_c. Thus a sequential history would be π_u followed by π_c followed by π'_c. Once again there is no possible equivalent sequential history for this construction of H'_1 since two consecutive search operations returned different values. Since for all constructions of H'_1 there is no equivalent sequential history strict linearizability is violated.

Case 2: Suppose π_u is linearized at its CPE. Since the CPE of an operation is a persistence event and persistence events have no effect on volatile shared memory p_c cannot distinguish between the execution E in which the last event was immediately priot to the CPE of π_u and an extension of E in which one more event occurs where that event is the CPE of π_u. Thus this case is equivalent to case 1.

In case 1 and 2 strict linearizability would not be violated if π_c completed the CPE of π_u and returned a value reflecting that the update was linearized. However, if π_c does complete the CPE of π_u this requires performing at least one persistence event which would contradict our assumption. Alternatively p_c could wait for some other process to complete the CPE of π_c and somehow signal the fact that the CPE has completed however this contradicts lock-freedom.

Note that It is trivial to construct equivalent executions for these cases where the update operation is a `remove` and the search is looking for the key removed by the update. Thus there exists no strict linearizable lock-free set with persistence-free search operations. $\square$

Figure 4.1 shows a visualization of the execution described in theorem 14. The proof exposes the fact that persistence-free searches rely on linearizing update operations after the operation is persisted. I have already mentioned two examples of algorithms that achieve persistence-free read-only operations by linearizing update operations after they are persisted in the ONLL universal construction and the SOFT algorithm. Linearizing operations after persisting clashes with the guarantees provided by strict linearizability. The main problem with linearizing after persisting is that there will exist an execution in which an operation that was persisted but never linearized needs to be linearized after a crash. One might think that a recovery procedure in a strict linearizable implementation could simply undo every operation that was persisted but not linearized in order to allow for persistence-free searches while preserving strict linearizability. This approach is flawed because it is not always possible for a recovery procedure to distinguish between an operation that was persisted but not linearized and an operation that was both persisted and linearized. If an update operation is linearized after its CPE then at least one other event would be present in the execution and the linearization point of the update would be at or after that event. Typically this would correspond to the response of some RMW primitive. The update might not be linearized until after several events occur. Either way, the state of the object in volatile shared memory will have changed. Since the CPE of the update has already occurred, there is no requirement for the implementation of the object to perform any persistence events after the linearization point. As a result, the state of the object in persistent memory could reflect that the operation has not been linearized while the state of the object in volatile memory would reflect that the operation has been linearized. Obviously the recovery procedure could not undo operations that have been linearized without violating strict linearizability. Note that if the implementation required persistence events after the linearization point to complete the update then the CPE would correspond with those persistence events and we would arrive at the same conclusion. Fundamentally, the problem here is that volatile memory and persistent memory cannot be updated simultaneously. If this were possible then we could avoid the problems that prevent persistence-free searches in implementations of strict linearizable objects.

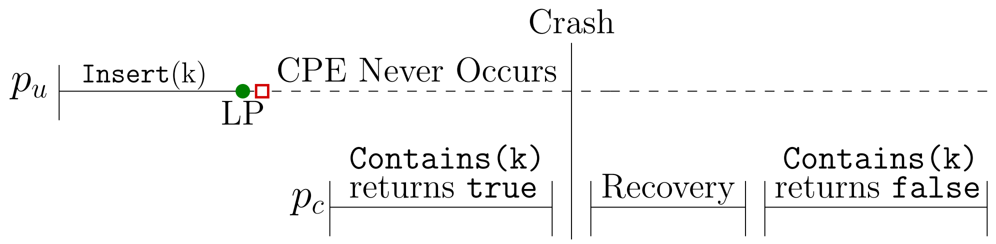


Figure 4.1: Theorem 14 example execution. In this example the search by process p_2 cannot determine if the CPE of the concurrent `Insert` has occurred and the response of the search depends on whether or not the `Insert` is linearized. The `Insert` by p_u is linearized at some at or before its CPE. p_u sleeps immediately before the CPE of the `Insert`. However, the search will observe the effects of the `Insert` in volatile memory. The response of a concurrent search by p_c reflects that the `Insert` has linearized but after the crash an identical search has a different response. This violates strict linearizability.

Chapter 5

Redundant Psync Lower Bound for Durably Linearizable Sets

Cohen et al. show that every update must perform at least one psync in any lock-free durable linearizable object [17]. Understanding when a psync is required is valuable since we know that psyncs are expensive due to the need to perform a persistence fence. Moreover, we have seen that existing persistent sets have focused on reducing the number of required psyncs. This motivates the question of whether or not every psync required by a durably linearizable set is actually useful. What if an implementation is forced to flush the same base object multiple times? If the value of the base object never changes then the flush accomplishes nothing. Any persistence fence that guarantees that the flush has completed is also useless. Informally, this would represent a redundant psync. In this chapter, I show that for any durably linearizable lock-free set there must exist an execution in which n concurrent processes are invoking n concurrent update operations and n-1 processes each perform at least one redundant psync. Note that this lower bound holds also for all classes strict linearizable sets.

Definition 15 (Destructive Write)**.** *A write (or RMW) to the base object b is destructive if it changes the value of b.*

A destructive write represents an event after which the state of volatile shared memory does not match the state of persistent shared memory. If the implementation wants to persist the base object b then a psync is required after a destructive write in which the flush of the psync is on b. Writes that do not change the value of any base object have no effect on volatile and persistent shared memory. These writes will never require a flush or persistence fence.

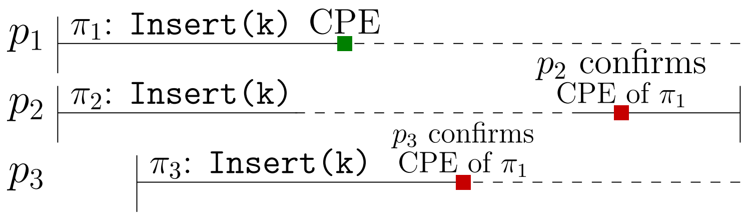


Figure 5.1: A simplified version of the execution described in theorem 17. In this example three processes are concurrently executing identical `insert` operations and two processes perform a redundant psync.

Definition 16 (Redundant-psync)**.** *Consider an execution E of a durably linearizable set.*

An explicit flush f applied to the base object b, is redundant if b was previously flushed by another explicit flush f', and there does not exist a destructive write to b between f' and f in E.

A persistence fence f_p is redundant if there exists another persistence fence f'_p prior to f_p and there does not exist any non-redundant flush between f'_p and f_p in E. A psync which is a flush and a persistence fence, is redundant if the persistence fence redundant.

The concept of a redundant pysnc captures the idea that a destructive write requires exactly one flush and one persistence fence in order to persist the base object modified by the write. Any extra flushes or fences are not necessary.

Theorem 17. *In an n-process system, for every durably linearizable lock-free set implementation I, there exists at least one execution of I in which n processes are concurrently performing update operations and n-1 processes performs a redundant psync.*

Proof. Consider a durably linearizable lock-free set implementation I. Assume for the purpose of showing a contradiction that no operation performs a redundant psync in every execution of I. Starting from an empty set, construct an execution E of I as follows:

Let n processes each invoke identical `insert` operations such that process p_i is performing the `insert` operation π_i ($1 \leq i \leq n$). As long as one of the processes continues to make progress, the decided response will be `true` for only one of the operations. Call this operation π_s. Let p_s run under no contention until immediately after the response of π_s is decided. Consider every other operation π_i, $\forall i \neq s$. Since π_i has the same arguments as π_s, if p_i continues to make progress the decided response of π_i will be `false`. Let every p_i, $\forall i \neq s$, progress until immediately after π_i is decided.

Consider the consequences of letting every p_i, $\forall i \neq s$ return from π_i with a response value of `false` without confirming that the CPE of π_s has occurred. Suppose a crash event occurs immediately after the response of the operation that completed last. Let E' be the crash-recovery extension of E and let π' be identical to π_s. Let E'' be $E' \cdot E_{\pi'}$ where $E_{\pi'}$ is the sequential execution beginning with the invocation of π' and ending with the corresponding response. Since the CPE of π_s did not occur in E and all π_i, $i \neq s$ failed in E, the response of π' in E'' will be `true`. Let H'' be the history of the crash-recovery extension of E''. In this scenario all π_i, $i \neq s$ would have completed in E. This means that $ops(H'')$ is not linearizable because the response of all π_i reflects being linearized after π' but π' was invoked after the response of the π_i that completed last. This violates durable linearizability. To avoid this problem, every process must confirm that the CPE of π_s has occurred.

Let b be the base object involved in the CPE of π_s. Only process p_s performed a destructive write on b. This means that only one flush applied to b will be non-redundant and therefore only one persistence fence following the flush will be redundant. Suppose one process continues until immediately after the CPE of π_s then sleeps indefinitely. If any other process flushes b and performs a persistence fence then that process will have performed a redundant psync. The CPE of π_s does not change the volatile shared memory configuration. This means that if the first process to progress past the CPE of π_s sleeps indefinitely, no process will be able to determine if the CPE of π_s has occurred. Let p_s continue until immediately after the CPE of π_s then sleep indefinitely. Every other process cannot complete until confirming that the CPE of π_s has occurred. However, since p_s already explicitly flushed b and performed a persistence fence any other process explicitly flushing b and performing a persistence fence would violate the initial assumption but no process can make progress without flushing b and performing a persistence fence. This contradicts lock-freedom. In this case n-1 processes p_i, $i \neq s$ must perform a redundant psync demonstrating that the initial assumption leads to a contradiction. $\square$

Izraelevitz et al. briefly mention that a helping mechanism for a non-blocking persistent object would include helping to persist operations [41]. Intuitively, durable linearizability requires that all operations that complete before a crash event are written to persistent memory. If the response of update operation π_1 relies on the durability of a different operation π_2 then π_1 must ensure that the CPE of π_2 has occurred. This is primarily a consequence of lock-free progress since π_1 cannot wait for other operations to ensure that the CPE of π_2 has occurred. Theorem 17 describes just one execution in which all but one process is forced to perform a redundant psync. The execution describes the case n processes all need to explicitly flush the same base object b and perform a persistence fence but only one processes performs a destructive write on b. Since only one destructive write was applied on b, only one explicit flush (and persistence fence) is necessary to guarantee that b is written to persistent memory. Since all n processes explicitly flush b then all but one of the psyncs are redundant. Figure 5.1 provides a simplified version of this example for $n = 3$.

When n is large, the execution in which $n - 1$ processes are forced to perform a redundant psync is obviously an extreme case and in practice it is unlikely that any real scheduler would result in this execution. Regardless, it is still important to note that redundant psyncs cannot be completely avoided without sacrificing either lock-freedom or durable linearizability. It is likely that one would not want to sacrifice durable linearizability since the alternative would be that the implementation guarantees buffered durable linearizability (or the implementation makes no guarantees with respect to crashes) and it is not clear whether or not a lock-free buffered durable linearizable set could completely avoid redundant psyncs. For this reason, to completely avoid redundant psyncs we would sacrifice lock-freedom. It is not difficult to see how we could use locks to avoid redundant psyncs. Consider the same example used in theorem 17 but with locks. In this case p_s could simply claim and hold a lock on b until some point after the CPE of π_s. All of the other processes would then spin until the lock on b is released. Each of these other processes would then claim the lock on b themselves guaranteeing that the value of b will not change therefore requiring no additional psyncs.

It is possible for an operation in sequential execution of a durably linearizable set to perform a redundant psync. This would be the case if the configuration of volatile shared memory does not contain any state information that can be used to identify whether or not a specific base object has been written to persistent memory. In practice, it is likely that most implementations would somehow announce the fact that a base object has been written to persistent memory. All of the existing persistent sets presented in chapter 3 have some mechanism that performs this task (an obvious example is the persistence mark of [24]).

Chapter 6

Upper Bounds

The lower bounds presented in the previous chapters offer insights into the theoretical limits of persistent sets for both durable linearizability and strict linearizability. While these lower bounds demonstrate a clear separation between durable and strict linearizability, it is unclear whether or not we can observe any meaningful separation in practice. In this chapter I present several implementations of durable linearizable sets which I later compare against the existing work. More specifically, I implement several persistent linked-lists. I choose to study linked-lists because they generally do not require complicated volatile synchronization mechanisms. This makes them an ideal candidate for studying the principles of durable linearizable sets without adding unnecessary complexity.

6.1 Design Decisions

The implementations of my persistent lists are based on several important design decisions One of the first considerations is the choice of logging versus log-free. Existing literature favours log-free approaches. Logging would also introduce unnecessary overhead. For these reasons I choose to focus on log-free implementations of persistent sets. Likewise, I need to determine how much data I want to persist. We have seen that there are two general approaches, persisting data structure links or persisting only keys and minimal metadata. I choose to take the approach of persisting data structure links because this approach does not sacrifice recovery complexity. Finally, I choose to implement both strict linearizable and durable linearizable versions of the persistent list. The durable linearizable versions support a persistence-free search operation whereas the strict linearizable versions do not.

6.2 Extended Link-and-Persist

The Link-and-Persist technique from [24] represents the state of the art for hand-crafted algorithms that persist the links of a data structure. Unlike the algorithms in [58], the Link-and-Persist technique can be used to implement persistent sets without compromising recovery complexity. I build on the Link-and-Persist technique by extending it to allow for for persistence-free searches and improved practical performance.

One of the main reasons why the Link-and-Persist technique cannot be used to implement persistence-free searches is the fact that the Link-and-Persist technique offers no way to determine what type of update operation (`Insert` or `Remove`) caused a link to become marked as not persistent. To understand the challenges consider an execution with two processes, p_1 and p_2 where p_1 is performing a `contains` operation π_1 concurrently with p_2 which is performing a `remove` operation π_2. Assume that π_1 is persistence-free. Let k be the key that π_1 is searching for and let k also be the key that π_2 wants to remove. Suppose that k is already in the set and is contained in the node n. Let p_2 progress until it unlinks n then let p_2 sleep indefinitely at some point before the CPE of π_2. p_1 will eventually traverse the link updated as part of π_2 which will lead to the successor of n. The *persistence bit* of the link will indicate that it is has not been written to persistent memory. The *persistence bit* cannot alone be used to determine the correct response for π_1. If a system crash occurs the contents of persistent memory will reflect that k is still the data structure. π_1 is persistence-free so it will not complete the CPE of π_2 and we cannot assume that the CPE of π_2 will occur as a result of a background flush. This means that π_1 must either perform a persistence event or it must be linearized prior to π_2. The former violates the initial assumption. The latter requires that π_2 is linearized after its CPE. Since we are not concerned with guaranteeing strict linearizability, we can assume that π_2 is linearized after its CPE. This means that p_1 must be able to determine the key contained in the node that π_2 unlinked. However, in volatile shared memory, n is not reachable via a traversal from the root. Since p_1 has no way to determine the key contained in n, if π_1 returns `false` the execution is not durable linearizable. For the case where π_1 returns true simply reconstruct the example starting from a configuration wherein the node unlinked by π_1 does not contain k and k is not contained in any node in the list. This means that there is no safe way to linearize π_1 without performing a persistence event.

Implementing persistence-free searches relies on the ability to linearize successful update operations at some point after the CPE of the operation. This idea was pointed out by Cohen et al [17]. When persisting structure, this means that searches must be able to determine if the pointer is not persistent because of an `Insert` operation or a `Remove` operation. My extension addresses these issues with two changes to the original technique.

First, we require that a successful update operation, π_u, is linearized after its CPE. More specifically, if a volatile data structure would linearize π_u at the success of a rmw on a pointer v then we require that π_u is linearized at the success of the rmw that sets the persistence mark in v. This means that if a search traverses a pointer, v, marked as not persistent the search can always be linearized prior to the concurrent update which modified v. This means that if a persistence-free search traverses v while it is marked as not persistent the search can always be linearized prior to the concurrent update which modified v.

Secondly, since successful updates are linearized after their CPE if the response of search operation depends on data that is linked into the data structure by a pointer marked as not persistent then the search must be able to access the last persistent value of that pointer. To achieve this, we a pointer field to every node which we call the *old field* or *old pointer*. A node will have both an *old pointer* and a pointer to its successor (*next pointer*) which effectively doubles the size of every data structure link. The *old field* will point to the last persistent value of the successor pointer while the successor pointer is marked as not persistent. In practice, the *old field* must be initialized to `null` then updated to a non-`null` value when the corresponding successor pointer is modified to a new value that needs to be persisted. Note that modifications like flagging or marking do not always need to be persisted; this depends on the whether or not the update can complete while the flagged or marked pointers are still reachable via a traversal from the root of the data structure. The easiest way to correctly update the *old field* is to update the successor pointer and the *old field* atomically using a DWCAS. Alternatively, a SWCAS can be used but this requires adding extra volatile memory synchronization to ensure correctness. For some data structures such as linked-lists using only SWCAS might also require adding an extra psync to updates. In order to allow search operations to distinguish between pointers that are marked as not persistent because of a `remove` operation versus those that are not persistent because of an `insert` operation we require that the *old field* is always updated to a non-`null` value whenever a `remove` operation unlinks nodes from the data structure. `Insert` operations that modify the data structure must flag either the *old field* or the corresponding successor to indicate that the pointer marked as not persistent was last updated by an insert. When using SWCAS to update the *old field* this flag must be on the successor pointer.

With our extension if the response of a search operation depends on data linked into the data structure to by a pointer marked as not persistent it can be linearized prior to the concurrent update operation that modified the pointer and it can use the information in the *old field* to determine the correct response which does not require performing any psyncs. If the search finds that the update was an insert it simply returns `false`. If the update was a remove but the search was able to find the value that it was looking for then it can return `true` since that key will be in persistent memory. If the update was a remove but the search was not able to find the value that it was looking for then it can check the if the node pointed to by the `old field` contains the value.

6.3 Persistent List Implementations

In order to compare my extension to existing work I provide several different implementations of my persistent list all of which utilize my extended-link-and-persist approach. I use two different methods for achieving synchronization in volatile memory. Specifically I use one based on the Harris list [35] and another based on the work of Fomitchev and Ruppert [30]. The former takes a lazy approach to deletion that relies on marking for logical deletion and helping. As a result, marked pointers must be written to persistent memory which requires an extra psync. The latter does not take a lazy approach to deletions but still relies on helping and requires extra volatile memory synchronization through the use of marking and flagging. Fortunately, I do not need to persist marked or flagged pointers with this approach. I also utilize 2 different synchronization primitives, DWCAS and SWCAS.[1] The names of the different implementations of the persistent list describe the synchronization approach and the synchronization primitive. The Physical-Del list uses the synchronization approach inspired by the work of Fomitchev and Ruppert. The Logical-Del list uses the synchronization approach inspired by the Harris list. When the algorithm name contains no suffix this refers to implementations that use DWCAS. If the name has the suffix *-S* then the implementation uses SWCAS. In total, I provide four implementations, Physical-Del, Physical-Del-S, Logical-Deland Logical-Del-S. Table 6.1 summarizes some details about these different implementations. A detailed description of my persistent list and pseudocode is presented in § 6.3.2. Note that I assume that data structure nodes fit on a single cache line. If data structure nodes do no fit on a single cache line I could augment the implementations to take an approach similar to Cohen et al. [16].

6.3.1 Search Variants

As part of the persistent list, I implement 4 variants of the `contains` operation. I refer to these variants as *persist-all*, *asynchronous-persist-all*, *persist-last* and *persistence-free*.

[1]I use a hardware implementation of DWCAS, namely the cmpxchg16b instruction. Software based implementations such as those from the C++ std::atomic library (which utilizes locks) are a significant performance bottleneck.

Algorithm Name	**Synch. Approach**	**Synch. Primitive**	**Minimum Psyncs Per Insert**	**Minimum Psyncs Per Remove**	**Recovery Step Complexity**
Physical-Del	Fomitchev	DWCAS	1	1	O(N + n)
Physical-Del-S	Fomitchev	SWCAS	2	1	O(N)
Logical-Del	Harris	DWCAS	1	2	O(N + n)
Logical-Del-S	Harris	SWCAS	2	2	O(N)

Table 6.1: Persistent List Details, Recovery is expressed in terms of N nodes and a maximum of n concurrent processes.

Persist All (PA). While traversing the list write any pointer marked as not persistent to persistent memory then attempt to set its persistence bit via a CAS. The maximum psyncs per search with this variant is unbounded.

Asynchronous Persist All (APA). While traversing the list, flush any pointer marked as not persistent to persistent memory but perform only a single persistence fence after the traversal then attempt to set the persistence bit for any flushed pointer that did not change since the flush. This requires the use of asynchronous flush instructions (clflushopt). The maximum psyncs per search with this variant is one.

Persist Last (PL). If the pointer into the terminal node of the traversal performed by the search is marked as not persistent then write it to persistent memory and set its persistence bit via a CAS. This variant is the most similar to the searches in the linked list implementation from [24]. The maximum psyncs per search with this variant is one.

Persistence Free (PF). If the pointer into the terminal node of the traversal performed by the search is marked as not persistent then use the information in the *old field* of the node's predecessor to determine the correct return value without performing any persistence events. Since it does not need to set the durability bit of any link, this variant does not perform any writes and never performs any persistence events.

Theorem 18. *The Physical-Del list is durable linearizable and lock-free.*

I provide a proof of theorem 18 in § 6.4. All of the variants of the persistent list are durable linearizable and lock-free. The proof for the other implementations can be derived from the proof for the Physical-Del list. If I restrict the persistent list such that it never invokes a persistence-free `contains` operation then I can prove that it is strict linearizable and lock-free. However, imposing this restrict effectively eliminates the need for the mechanisms added by the extension to the Link-and-Persist technique. This provides some insight for why researchers may prefer to implement strict linearizable data structures.

6.3.2 Pseudocode Overview

In the following sections I will provide a more in depth description of the different implementations of my persistent list by examining the pseudo code. I will focus primarily on the DWCAS implementation of the *Physical-Del* list since this is version performed best in the experimental tests. I will also briefly describe the DWCAS version of the *Logical-Del* list and the corresponding SWCAS version. Throughout the pseudo code I utilize the functions `UnmarkPtr, IsDurable, MarkDurable` and others of the form *IsX* or *MarkX*. These functions represent simple bitwise operations used to remove marks/flags, check if a node is marked/flagged and apply marks/flags. I omit their full function bodies. I also use the function `Psync`. On Intel systems `Psync` can be a single *clflush* instruction, a *clflushopt* and an *sfence* or a *clwb* and an *sfence*.

Every version of the persistent list uses the same *Node* data type. A *Node* contains four fields: *key, value, next* and *old*. *Next* and *old* are word sized pointers. I assume that the size of the *key* and *value* fields allow a single Node to fit on one cache line meaning only a single *flush* is required in order to write the contents of the node to persistent memory.[2] Nodes represent the *critical* data that I want to persist. The assumption that the critical data fits on a single cache line is common. The Link-and-Persist list, Link-Free list and SOFT list require similar assumptions. It is possible that the persistent list could be modified to allow for the case where nodes do not fit onto a single cache line by adopting a strategy similar to [16].

Every version of the persistent list utilizes a dummy *head* and *tail* node where the root of the list is always head, head has no predecessor and contains the key equivalent to negative infinity. Tail contains the key equivalent to infinity and has no successor. An empty list consists of only the head and tail nodes where *head.next* points to *tail*.

6.3.3 Physical-Del List

Figures 6.1-6.3 show the pseudo code for the Physical-Del list which uses DWCAS. This implementation is based on the volatile list of Fomitchev and Ruppert [30].

With this approach, I utilize the bottom three bits of a node's next pointer for flagging and marking. A link in the data structure can be represented as the tuple ⟨next, dflag, marked, persistence-bit⟩. The bottom bit is the persistence bit which as in Link-and-Persist-technique indicates if the link is persistent. However, in my implementations if the

[2] On Intel systems a typical cache line size is 64 bytes which is usually more than enough to store a single node.

persistence bit is set then the node is persistent meaning the persistence mark is applied to a link after it is written to persistent memory. The *marked* bit is used to mark a node as logically deleted. The *dflag* bit indicates that a remove operation is going to remove the successor of the node. Each of these marks are initialized to 0.

This implementation uses DWCAS. Whenever a node is updated both its *next* and *old* fields are updated atomically via a single DWCAS. Any update that sets the persistence bit in the next pointer of a node will also revert the old field to `null`.

I will now give an overview of the main functions utilized by the Physical-Del list. Many of these functions will be very similar to the volatile list in [30] with the main differences being related to achieving a persistence-free search.

Find. The `Find` function is shown in figure 6.1. `Find` has a single argument, *key* and returns the tuple $\langle gp, p, curr \rangle$ where *gp*, *p* and *curr* are nodes such that $curr.key \geq key$ and there was a time during the execution of the function where *gp.next* pointed to *p* and (a possibly later time where) *p.next* pointed to *curr*. As in the Link-and-Persist-technique, `Find` will also confirm that *gp.next* and *p.next* are persistent.

Insert. The `Insert` function has two arguments *key* and *val* representing the key to insert and its corresponding value. Any execution of `Insert` begins by invoking `Find` yielding the tuple $\langle gp, p, curr \rangle$. If the key contained *curr* is the same as the key to be inserted then we return `false`. In this case there is no need to perform any persistence events because the `Find` function confirms that *curr* was persistent meaning there was a time during the execution of the insert where *curr* was in persistent memory. If *curr* does not contain the key then the insert checks that the *p* is not marked or dflagged using the function $IsClean$. This check ensures that the insert can fail early if the DWCAS on line 19 is guaranteed to fail. If *p* is dirty then the insert will attempt to help the concurrent update then retry. If *p* is not dirty then a new node *n* is created. The $CreateNode$ function sets the next pointer of *n* to *curr*, explicitly flushes *n* to persistent memory and sets the persistence bit in *n.next*. Next, on line 16, we create $iflagCurr$ which is a pointer to *curr* with the iflag bit set. The insert then attempts a DWCAS to update *p.next* to *n* and *p.old* to $iflagCurr$. If the DWCAS is successful the insert explicitly flushes *p.next*, performs a persistence fence and sets the persistence bit via the `Persist` function on line 20. If the DWACS fails then the insert will retry.

Remove. The `Remove` function has a single argument *key* representing the key to be removed. The initial steps of `Remove` are similar to `Insert`. A remove begins by invoking `Find` which again returns the tuple $\langle gp, p, curr \rangle$ and ensures the same guarantees as in `Insert`. If *curr.key* is not *key* then the remove returns `false`. If the key was found then the remove must confirm that both *p* and *curr* are clean. If either is found to be dirty the

```
$Persist(node, expNext, old)$
    $Flush(\&node.next)$
    $durNext = MarkDurable(expNext)$
    if $old == null$ then
        $old = node.old$
    $des = \langle node.next, node.old \rangle$
    $exp = \langle expNext, old \rangle$
    $new = \langle durNext, null \rangle$
    $DWCAS(des, exp, new)$



$Find(key)$
    $gp = null$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$

    while true
        if $curr.key \geq key$ then
            break
        $gp = p$
        $p = curr$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$

    if $gp \neq null$ then
        $gpNext = gp.next$
        if not $IsDurable(gpNext)$ then
            $Persist(gp, gpNext, null)$

    if not $IsDurable(pNext)$ then
        $Persist(p, pNext, null)$

    return $\langle gp, p, curr \rangle$
```

```
$ContainsPersistFree(key)$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$

    while true
        if $curr.key \leq key$ then break
        $p = curr$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$

    if $IsDurable(pNext)$ then
        return $curr.key == key$
    else
        $old1 = p.old$
        $pNext2 = p.next$
        $old2 = p.old$
        $pDiff = pNext \neq pNext2$
        $oldDiff = old1 \neq old2$
        $oldNull = old1 == null$

        $wasDur = pDiff$ or $oldDiff$
        $wasDur = wasDur$ or $oldNull$

        if $wasDur$ then
            return $curr.key == key$
        else
            if $IsIflagged(old1)$ then
                return false
            else
                if $curr.key == key$ then
                    return true
                else
                    $old1 = UnmarkPtr(old1)$
                    return $old1.key == key$
```

Figure 6.1: Persist, Find and Persistence Free Contains Functions for Physical-Del (DW-CAS)

remove will attempt to help the concurrent update then retry. If *p* and *curr* are clean then *dflagCurr* is created. *dflagCurr* is a pointer to *curr* with the dflag bit set. The remove then attempts a DWCAS on line 40 to update *p.next* to *dflagCurr* and *p.old* to `null`. (When we say that the DWCAS updates *p.old* to `null` it is more accurate to say that the DWCAS confirms that *p.old* is `null` since the expected value for *p.old* is also `null`). If the DWCAS succeeds then the `HelpRemove` function is invoked to complete the remove. After the `HelpRemove` function returns the remove the returns `true`. If the DWCAS fails then the remove retries.

```
1  Insert(key, val)
2    while true
3      ⟨gp, p, curr⟩ = Find(key)
4      pNext = p.next

6      if curr.key == key then
7        return false
8      if not IsClean(pNext) then
9        HelpUpdate(gp, p)
10     else
11       newNode = CreateNode(key, val, curr)
12       durCurr = MarkDurable(curr)

14       des = ⟨p.next, p.old⟩
15       exp = ⟨durCurr, null⟩
16       iflagCurr = MarkIflag(curr)
17       new = ⟨newNode, iflagCurr⟩

19       if DWCAS(des, exp, new) then
20         Persist(p, newNode, iflagCurr)
21         return true
```

```
22 Remove(key)
23   while true
24     ⟨gp, p, curr⟩ = Find(key)
25     cNext = curr.next
26     pNext = p.next

28     if curr.key ≠ key then
29       return false
30     if not IsClean(cNext) then
31       HelpUpdate(p, curr)
32     else if not IsClean(pNext)then
33       HelpUpdate(gp, p)
34     else
35       dflagCurr = MarkDflag(curr)

37       des = ⟨p.next, p.old⟩
38       exp = ⟨curr, null⟩
39       new = ⟨dflagCurr, null⟩
40       if DWCAS(des, exp, new) then
41         HelpRemove(p, dflagCurr)
42         return true
```

Figure 6.2: Update Functions for Physical-Del (DWCAS)

Helper Functions. Update operations rely on a helping mechanism to ensure lock-free progress. This mechanism relies on the functions `HelpUpdate`, `HelpRemove` and `HelpMarked`.

`HelpUpdate` is invoked when an update operation finds the node it needs to modify is dirty. The `HelpUpdate` function has two arguments *parent* and *dirtyNode* and examines these values to determine the appropriate helper function to invoke. If *dirtyNode* is dflagged then `HelpRemove` is invoked. If *dirtyNode* is marked then `HelpMarked` is invoked. If *dirtyNode* is clean `HelpUpdate` returns.

```
1  HelpUpdate(parent, dirtyNode)
2     dirtyNext = dirtyNode.next
3     dirtySucc = UnmarkPtr(dirtyNext)
4     if IsDflagged(dirtyNext) then
5        HelpRemove(dirtyNode, dirtySucc)
6     else if IsMarked(dirtyNext)
7        HelpMarked(parent, dirtyNode)



11 HelpMarked(parent, nodeToDel)
12    succ = UnmarkPtr(nodeToDel.next)
13    expNext = Markdflag(nodeToDel)
14    expNext = MarkDurable(expNext)

16    des = ⟨parent.next, parent.old⟩
17    exp = ⟨expNext, null⟩
18    new = ⟨succ, expNext⟩

20    if DWCAS(des, exp, new) then
21       Persist(parent, succ, expNext)
```

```
22 HelpRemove(parent, nodeToDel)
23    while parent.next == nodeToDel
24       succ = nodeToDel.next
25       des = ⟨nodeToDel.next, nodeToDel.old⟩
26       if not IsDurable(succ) then
27          Persist(nodeToDel, succ, null)
28       durSucc = UnmarkPtr(succ)
29       durSucc = MarkDurable(durSucc)
30       markedSucc = MarkDel(durSucc)
31       exp = ⟨durSucc, null⟩
32       new = ⟨markedSucc, null⟩

34       last = ValDWCAS(des, exp, new)
35       lastMarked = IsMarkedForDel(last.next)

37       if last == exp or lastMarked then
38          HelpMarked(parent, nodeToDel)
39          return
40       else if IsDflagged(last.next) then
41          next = UnmarkPtr(last.next)
42          HelpRemove(nodeToDel, next)
```

Figure 6.3: Helper Functions for Physical-Del (DWCAS)

The `HelpRemove` function is responsible for marking the successor of a dflagged node. It takes two arguments *parent* and *nodeToDel*. `HelpRemove` begins by ensuring that *nodeToDel.next* is persistent. If *nodeToDel.next* does not have its persistence bit set then the `Persist` function is invoked to write *nodeToDel.next* to persistent memory and set the persistence bit. Next, we create *markedSucc* which is a marked pointer to *nodeToDel.next*. On line 34 we perform ValDWCAS (a DWCAS that returns the last value at the destination) to update *nodeToDel.next* to *markedSucc* and confirm that *nodeToDel.old* is `null`. If the value returned by the ValDWCAS matches the expected value *exp* or if it is marked then we invoke the `HelpMarked` function then return otherwise we will need to retry. Before retrying we check if the value returned by the ValDWCAS is dflagged. If it was dflagged then we invoke the `HelpRemove` function with the arguments *nodeTodel* and the unmarked *nodeToDel.next*.

The `HelpMarked` function is responsible for physically deleting a node. `HelpMarked` takes two arguments *parent* and *nodeToDel* This requires performing a single DWCAS on line 20 which attempts to atomically update *parent.next* to the unmarked *nodeToDel.next* and *parent.old* to *nodeTodel*. If the DWCAS succeeds the `Persist` function is invoked to write *parent.next* to persistent memory, set the persistence bit in *parent.next* and revert *parent.old* to `null`. There is no need to retry this DWCAS since the only way that it can fail is if another process physically deleted *nodeToDel*.

Contains. We offer four different variants of the `Contains` function. The pseudo code for these variants are show in figure 6.1 and figure 6.4. Each of these functions has a single argument *key*.

The simplest of these variants is the `ContainsPersistAll`. In this version we traverse the list until we reach a node containing a key greater than or equal to *key*. During the traversal, if we find a pointer that does not have its persistence bit set then the `Persist` function is invoked to write the pointer to persistent memory and set its persistence bit. Finally, we return `true` if the terminal node of the traversal contains *key* and `false` otherwise. Unsurprisingly this variant performs poorly in practice.

The `ContainsAsynchPersistAll` function exploits asynchronous flush instructions to require only a single persistence fence. In this case, during the traversal any pointer that does not have its persistence bit set is asynchronously flushed to persistent memory. A statically allocated array is used to keep track of the nodes that were asynchronously flushed.[3] For simplicity the pseudo code expresses this with `append` on line 39. After the traversal a single persistence fence is performed on line 46. Next we iterate the collection of asynchronously flushed nodes. If the next pointer of the node has not changed since the flush then we set the persistence bit and revert the old field to `null` on line 55. Finally, we return `true` if the terminal node of the traversal contains *key* and `false` otherwise.

The `ContainsPersistLast` function is similar to the search function in [24]. In this version, we do not flush anything during the traversal. Instead, we check that link that led to the the terminal node reached by the traversal has its persistence bit set on line 9. If this is not the case then we invoke the `Persist` function to write the pointer to persistent memory and set the persistence bit. Again we return `true` if the terminal node of the traversal contains *key* and `false` otherwise.

Finally we have the `ContainsPersistFree` function. The traversal performed by this function is the same as in the `ContainsPersistLast` function. Similar to the persist last variant, we are only concerned with the last pointer traversed by the `contains`. If pointer that led to the the terminal node, *curr*, found during the traversal has its persistence bit set to then we simply return `true` if *curr* contains *key* and `false` otherwise on line 26. If this link does not have its persistence bit set we must read the value stored in the old field of the predecessor of *curr*, namely *p.old*. Unfortunately, there is no way to perform an

[3]The size of this array should be proportional to the maximum number of concurrent processes and the maximum key. We do not put great emphasis on this since this variant does not perform well in practice.

```
$ContainsPersistLast(key)$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$
    while $curr.key \leq key$
        $p = head$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$
    if not $IsDurable(pNext)$ then
        $Persist(p, curr, null)$
    if $curr.key == key$ then return true
    else return false

$ContainsPersistAll(key)$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$
    while true
        if not $IsDurable(pNext)$ then
            $Persist(p, pNext, null)$
        if $curr.key \geq key$ then
            break
        $p = curr$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$
    if $curr.key == key$ then return true
    else return false
```

```
$ContainsAsynchPersistAll(key)$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$
    $flushList = []$
    while true
        if not $IsDurable(pNext)$ then
            $AsynchFlush(\&p.next)$
            $old = p.old$
            $flushList.append(\langle parent, curr, old \rangle)$
        if $curr.key \geq key$ then break
        $p = curr$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$

    if not $IsEmpty(flushList)$ then
        $SFence()$

    for each $\langle n, next, old \rangle$ in $flushList$
        $nonDur = n.next == next$ and
      $n.old == old$
        if $nonDur$ then
            $durNext = MarkDurable(next)$
            $des = \langle node.next, node.old \rangle$
            $exp = \langle next, old \rangle$
            $new = \langle durNext, null \rangle$
            $DWCAS(des, exp, new)$
    if $curr.key == key$ then return true
    else return false
```

Figure 6.4: Persist Last and Persist All Contains for Physical-Del (DWCAS)

atomic read of *p.next* and *p.old* without using locking or performing a DWCAS. Both of these options would be expensive. Fortunately we can avoid this by exploiting the fact that *p.old* and *p.next* are atomically updated together. We must ensure that the value we read for *p.old* corresponds to the value that we last read for *p.next*. Recall that in this scenario, the `contains` last read that *p.next* points to *curr* and does not have its persistence bit set. Since we can only atomically read one of *p.old* and *p.next* we must reread these fields to ensure that neither field has changed. This requires reading both *p.old* and *p.next* twice then comparing the values. Lines 15 through 23 show these reads and the associated comparisons. If the first read does not match the second read or if *p.old* is `null` then there was a time during the execution of the `contains` that the value we read for *p.next* was persistent so we return `true` if *curr* contains *key* and `false` otherwise on line 26. If the values for both reads of *p.old* and *p.next* match and *p.old* is not `null` then we know that the value we read for *p.old* corresponds to the value we read for *p.next* and that *p.next* is not persistent. In this case, we check if *p.old* was iflagged. If *p.old* was iflagged then *p.old* was updated to a non-`null` value by a concurrent `Insert` so we can return `false`. If *p.old* was not iflagged we know that *p.next* is not persistent because of a concurrent remove operation and we know that *p.old* points to the last persistent value stored in *p.next*. This also means that there was a time during the execution of the `contains` where *curr* was pointed to by a persistent link so if *curr* contains *key* then we can return `true`. If *curr* does not contain *key* then we return `true` if the node pointed to by *p.old* old contains *key* and `false` otherwise.

6.3.4 Logical-Del List

The pseudo code for the Logical-Del list is shown in Figures 6.5- 6.6. This version of the persistent list is based on the volatile Harris list [35]. The volatile synchronization approach is the primary difference compared to the Physical-Del list. The Logical-Del list does not utilize any flagging. Nodes can still be marked indicating that the node is logically deleted. This means that a link in the data structure can be represented as the tuple $\langle next, marked, persistence - bit\rangle$. The persistence-bit is used in the same manner as in the Physical-Del list. Since `remove` operations can complete before a logically deleted node is physically deleted, the logical deletion mark must be written to persistent memory before physically unlinking the node. We will now give an overview of the main functions utilized by the Logical-Del list.

Trim. The *Trim* function has two arguments, *parent* and *curr*. `Trim` performs a DWCAS that attempts to unlink the logically deleted node *curr* by updating *parent.next* to the successor of *curr* and *p.old* to `null`.

```
1  $Insert(key, val)$
2    while true
3      $\langle p, curr\rangle = Find(key)$
4      $pNext = p.next$

6      if $curr.key == key$ then
7        return false

9      $newNode = CreateNode(key, val, curr)$
10     $durCurr = MarkDurable(curr)$
11     $iflagOld = MarkIflag(curr)$

13     $des = \langle p.next, p.old\rangle$
14     $exp = \langle durCurr, null\rangle$
15     $new = \langle newNode, iflagOld\rangle$

17     if $DWCAS(des, exp, new)$ then
18       $Persist(p, newNode, iflagOld)$
19       return true
```

```
1  $Remove(key)$
2    while true
3      $\langle p, curr\rangle = Find(key)$
4      $currNext = curr.next$
5      $pNext = p.next$
6      if $curr.key\ != key$ then
7        return false
8      if not $IsDurable(currNext)$ then
9        $Persist(curr, currNext, null)$

11     $markedNext = MarkDel(curr)$
12     $des = \langle curr.next, curr.old\rangle$
13     $exp = \langle currNext, null\rangle$
14     $new = \langle markedNext, \rangle$

16     if $DWCAS(des, exp, new)$ then
17       $Persist(curr, markedNext, null)$
18       $Trim(p, curr)$
19       return true
```

Figure 6.5: Update Operations for Logical-Del (DWCAS)

Find. The `Find` function in the Logical-Del list is very similar the `Find` in the Physical-Del. `Find` has a single argument, *key* and returns the tuple $\langle p, curr\rangle$ where p and *curr* are nodes such that $curr.key \geq key$ and there was a time during the execution of the function where *p.next* pointed to *curr*. Unlike the Physical-Del list we do not return *gp* however, `Find` will still confirm that *gp.next* and *p.next* are persistent. The traversal performed by `Find` will help cleanup any nodes that are marked as logically deleted. Before physically deleting any logically deleted nodes, `Find` will ensure that the logical deletion mark is persistent by invoking the `Persist` function on line 28. The `Find` will invoke the `Trim` function on line 29 to physically delete the logically deleted node.

Insert. The `Insert` function of the Logical-Del list is almost identical to the `Insert` of the Physical-Del list. Unlike the Physical-Del list this version does not check if nodes are dirty since nodes are never flagged and logically deleted nodes are trimmed during `Find`.

Remove. The `Remove` function also does not check if nodes are dirty. After invoking `Find` and confirming that *curr* contains the *key*, the `Remove` will ensure that *curr.next* is persistent on line 8. If *curr.next* is not persistent the `Remove` will invoke `Persist` on line 9. The `Remove` will attempt to mark *curr* as logically deleted with the DWCAS on line 16. If this DWCAS succeeds, `Persist` is invoked to write the logical deletion mark to

```
$Trim(parent, curr)$
    $succ = curr.next$
    $durCurr = MarkDurable(curr)$
    $des = \langle parent.next, parent.old \rangle$
    $exp = \langle durCurr, null \rangle$
    $new = \langle succ, markedCurr \rangle$
    if $DWCAS(des, exp, new)$ then
        $Persist(p, succ, markedCurr)$


$GetMark(node)$
    $nodeNext = node.next$
    $marked = IsMarked(nodeNext)$
    $markDur = IsDurable(nodeNext)$
    return $marked$ and $markDur$



$Find(key)$
    $gp = null$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$
    while true
        $currNext = curr.next$
        if $IsMarked(currNext)$
            if not $IsDurable(currNext)$ then
                $Persist(curr, currNext, null)$
            $Trim(p, curr)$
        else
            if $curr.key \geq key$ then break
            $gp = p$
            $p = curr$
        $pNext = curr.next$
        $curr = UnmarkPtr(pNext)$
    if $gp! = null$ then
        $gpNext = gp.next$
        if not $IsDurable(gpNextPtr)$ then
            $Persist(gp, gpNext, null)$
    if not $IsDurable(pNext)$ then
        $Persist(p, pNext, null)$
    return $\langle p, curr \rangle$
```

```
$PersistenceFreeContains(key)$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$
    while true
        if $curr.key \geq key$ then break
        $p = head$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$

    $marked = GetMark(curr)$
    if $IsDurable(pNext)$ then
        if $curr.key == key$ then
            return not $marked$
        else
            return false
    else
        $old1 = parent.old$
        $pNext2 = parent.next$
        $old2 = parent.old$
        $pDiff = pNext \neq pNext2$
        $oldDiff = old1 \neq old2$
        $oldNull = old1 == null$
        $wasDur = pDiff$ or $oldDiff$
        $wasDur = wasDur$ or $oldNull$

        if $wasDur$ then
            if $curr.key == key$ then
                $marked = GetMark(curr)$
                return not $marked$
            else
                return false
        else
            if $IsIflagged(old1)$ then
                return false
            else
                if $curr.key == key$ then
                    $marked = GetMark(curr)$
                    return not $marked$
                else
                    $old1 = UnmarkPtr(old1)$
                    return $old1.key == key$
```

Figure 6.6: Helper functions, Find and Flush-Free Contains for (DWCAS) Logical-Del

persistent memory then invoke `Trim` to physically delete *curr*.

Contains. Due to the large amount of overlap with the Physical-Del list, I omit pseudo code for all of the contains functions in the Logical-Del list except `ContainsPersistFree`. The `ContainsPersistFree` function of the Logical-Del list is also quite similar to the `ContainsPersistFree` function of the Physical-Del list. The main difference is that the terminal node found by the traversal, *curr*, can be marked as logically deleted. For this reason, anywhere that we check if *curr* contains *key* we must also confirm that *curr* is not marked as logically deleted and if *curr* is marked as logically deleted we must confirm that the mark is persistent. This is accomplished by the `GetMark` function on lines 53, 71, and 80.

The `GetMark` function has one arguments, *node* and returns `false` if the *node* is not marked as logically deleted or it is marked as logically deleted by the mark is not persistent. Otherwise, `GetMark` returns `true`.

6.3.5 Logical-Del-S List

The pseudo code for the Logical-Del-S list is shown in Figures 6.7-6.8. 6.7-6.8. Without DWCAS we cannot atomically update both the next pointer and old pointer in a node. This eliminates the possibility of using the old pointer to allow insert operations to require a minimum of one psync. Insert operations flag the next pointer instead of the old pointer. This means that only remove operations can change the old pointer to a non-`null` value. The SWCAS never marks/flags the old pointer at all meaning a non-null old pointer always corresponds with a concurrent remove operation. Consequently, the SWCAS implementation requires two psyncs per insert. The extra psync is in the `CreateNode` function. I omit the body of the `CreateNode` function.

In the SWCAS version before attempting to physical delete at node *n* the operation must first CAS the old pointer from `null` to *n*. After unlinking *n* and persisting the updated next pointer, the old pointer can be reverted back to null using a CAS.

The old pointer is also used to help concurrent remove operations. Updates confirm that the old pointer of any node that they want to modify is `null`. If it is non-`null` then the update will invoke the HelpOld function which will help to physically unlink the node pointed to by the old pointer.

In the Logical-Del list, since inserts do not set the old pointer to a non-`null` value, it is possible that a remove operation will set the old pointer of a node *n* then a concurrent insert will update *n.next*. This is not problematic because the old pointer will later be reverted back to `null` without any issues. This is not possible in the Physical-Del list because the predecessor of a marked node is flagged before the old pointer is set so an insert cannot successfully CAS *n.next*.

The old pointer is set back to null in the following functions `Find`, `HelpRemove` (for the Logical-Del-S version only) and `HelpOld`. Consider the example where the old pointer in the node u is non-`null`. In the *HelpOld* function the CAS to revert old back to `null` is only performed after first attempting to help the remove that set the old pointer to a non-`null` value. Call the node pointed to by the old pointer d (for node to delete). Helping the remove requires attempting a CAS on the next pointer from d to *u.next*. If this CAS failed then either 1) some other process completed the CAS to unlink d but no process progressed any further meaning *u.old* still points to the last durable value stored in *u.next* and *u.next* is not durable or 2) some process unlinked d and persisted *u.next* meaning if *u.old* still points to d then *u.old* is stale. Regardless of whether or not the CAS succeeded we persist *u.next*. Finally we attempt to CAS *u.old* from d to *null*. This CAS will only succeed if it was true that *u.old* is stale. In the `Find` and `HelpRemove` functions if the operation relies on the durability of the next pointer in some node v and *v.next* is found to be non-durable then we persist *v.next*. We will also attempt to revert the *v.old* back to `null` via a CAS. After rereading *v.next*, this CAS is attempted if and only *v.old* points to a node other than *v.next* and *v.next* is durable. This will fail if the remove that set *v.old* to a non-`null` value has not been completed. In that case the process might need to help the remove. This helping is done in `Insert` and `Remove`.

Implementing other persistent sets using SWCAS would follow the same approach where the old pointer is never reverted to `null` without doing one of the following 1) helping the concurrent remove that set the old pointer to a non-`null` value, 2) confirming that the concurrent remove has completed or 3) if the response of remove operations relies only on marking for logicial deletion then the old field might need need to be reverted to null as a result of conflicting updates (as described previously with the Logicial-Del list).

```
$Find(key)$
    $gp = null$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$
    $cNext = null$

    **while true**
        $cNext = curr.next$
        **if** $IsMarked(cNext)$ **then**
            **if not** $IsDurable(cNext)$ **then**
                $Persist(curr, cNex)$
            $Trim(parent, curr)$
        **else**
            **if** $curr.key \geq key$ **then break**
            $gp = p$
            $p = curr$
        $pNext = curr.next$
        $curr = UnmarkPtr(pNext)$

    $gpNext = gp.next$
    **if not** $IsDurable(gpNext)$ **then**
        $Persist(gpOld, gpNext)$
        $gpOld = gp.old$
        $gpOldN = UnmarkPtr(gpOld)$
        $gpNext = gp.next$
        $gpNextN = UnmarkPtr(gpNext)$
        **if** $gpOldN \neq gpNextN$ **and**
      $IsDurable(gpNext)$ **then**
            $CAS(gp.old, gpOld, null)$

    **if not** $IsDurable(pNext)$ **then**
        $Persist(pOld, pNext)$
        $pOld = parent.old$
        $pOldN = UnmarkPtr(pOld)$
        $pNext = p.next$
        $pNextN = UnmarkPtr(pNext)$

        **if** $pOldN \neq pNextN$ **and**
      $IsDurable(pNext)$ **then**
            $CAS(p.old, pOld, null)$
    **return** $\langle p, curr \rangle$
```

```
$PersistenceFreeContains(key)$
    $p = head$
    $pNext = p.next$
    $curr = UnmarkPtr(pNext)$

    **while true**
        **if** $curr.key \geq key$ **then break**
        $p = head$
        $pNext = p.next$
        $curr = UnmarkPtr(pNext)$

    $marked = GetMark(curr)$

    **if** $IsDurable(pNext)$ **then**
        **if** $curr.key == key$ **then**
            **return not** $marked$
        **else**
            **return false**
    **else if not** $wasDur$ **and**
      $IsIflagged(pNext)$ **then**
        **return false**
    **else**
        **if** $curr.key == key$ **then**
            $marked = GetMark(curr)$
            **return not** $marked$
        **else**
            $old1 = parent.old$
            $pNext2 = parent.next$
            $old2 = parent.old$

            $pDiff = pNext \neq pNext2$
            $oldDiff = old1 \neq old2$
            $oldNull = old1 == null$

            $wasDur = pDiff$ **or** $oldDiff$
            $wasDur = wasDur$ **or** $oldNull$

            **if** $wasDur$ **then**
                **return false**
            **else**
                $old1 = UnmarkPtr(old1)$
                **return** $old1.key == key$
```

Figure 6.7: Logical-Del-S, Find and Persistence-Free Contains

```
$Trim(p, nodeToDel)$
    if $CAS(p.old, null, nodeToDel)$ then
        $HelpOld(p, nodeToDel)$

$Persist(node, nextPtr)$
    $Psync(\&node.next)$
    $durNext = UnmarkPtr(nextPtr)$
    $durNext = MarkDurable(nextPtr)$
    $CAS(node.next, nexPtr, durNext)$

$Insert(key, val)$
    while true
        $\langle p, curr \rangle = search(key)$
        $pOld = p.old$
        $durCurr = MarkDurable(curr)$

        if $pOld \neq null$ then
            $pNext = p.next$
            $HelpOld(p, old, pNext)$
            continue

        $newNode = CreateNode(key, val, curr)$
        $iflagNew = MarkIflag(newNode)$

        if $CAS(p.next, durCurr, iflagNew)$
      then
            $Persist(p, iflagNew)$
            return true
```

```
$HelpOld(node, old)$
    $succ = UnmarkPtr(old.next)$
    $durNext = MarkDurable(old)$
    $CAS(node.next, durNext, succ)$
      $nodeNext = UnmarkPtr(node.next)$
    if not $IsDurable(nodeNext)$ then
        $Persist(node, succ)$
    $CAS(node.old, old, null)$

$Remove(key)$
    while true
        $\langle p,\ curr \rangle = search(key)$
        $cNext = curr.next$
        $cOld = curr.old$
        $pNext = p.next$
        $pOld = p.old$

        if $curr.key\ != \ key$ then return
       false

        if $cOld! = null$ then
            $HelpOld(curr, cOld, cNext)$
        else if $pOld! = null$ then
            $HelpOld(p, pOld, pNext)$
        else
            $markedCurr = MarkForDel(cNext)$
            $exp = UnmarkPtr(cNext)$
            $exp = MarkDurable(ext)$
            if $CAS(curr.next, exp, markedCurr)$
      then
                $Persist(curr, markedCurr)$
                $Trim(p, curr)$
                return true
```

Figure 6.8: Logical-Del-S, Update functions and helpers

6.4 Correctness

In this section I will provide a proof sketch arguing that the DWCAS implementation of the Physical-Del list is durable linearizable and lock-free. Proving that the Physical-Del list is linearizable and lock-free is very similar to the correctness proof of [30]. It is fairly straightforward to prove lock-freedom and linearizability of the Physical-Del list since the volatile synchronization utilized in the Physical-Del list can be thought of as a simplified version of the linked list in [30] which is linearizable and lock-free. The proof of the DWCAS Harris style list is a natural extension of this proof and the original Harris list correctness proof.

6.4.1 Linearization Points

I will begin by describing how I choose the linearization points for every operation in the Physical-Del list. These linearization points are chosen such that updates are linearized at some point after the CPE of the operation. Note that this assumes that the `ContainsPersistFree` is utilized. If the `ContainsPersistFree` is never invoked then I can argue that the Physical-Del list is strict linearizable. This requires that the linearization point is always chosen to be the CPE of the operation (or for unsuccessful updates and searches the CPE of the corresponding successful update). Before describing the linearization points I provide some necessary definitions.

Definition 19 (Durable)**.** *At configuration c, a pointer ρ is considered durable if the durability bit in ρ is set to 1 at c.*

A node n, is considered durable if the pointer into n is durable at c.

c, we say that n is durably linked in the volatile data structure if n is durable and n is reachable via a traversal starting from the root at c.

Definition 20 (Volatile Data Structure)**.** *The volatile data structure at a configuration c, is the set of nodes that are reachable by a traversal starting from the head.*

Definition 21 (Volatile Abstract Set)**.** *The volatile abstract set at a configuration c is the set of keys in nodes in the volatile data structure, at configuration c.*

Definition 22 (Persistent Data Structure)**.** *Consider an execution E of the Physical-Del list. Let c be the configuration after the last event in E. Let E' be the crash-recovery extension of E. Let c' be the configuration after the last event in E'. The persistent data structure at c is equivalent to the volatile data structure at c'.*

Definition 23 (Persistent Abstract Set)**.** *Consider an execution E of the Physical-Del list. Let c be the configuration after the last event in E. Let E' be the crash-recovery extension of E. Let c' be the configuration after the last event in E'. The persistent abstract set at c is the equivalent to the volatile abstract set at c'.*

Insert. Consider an `Insert` π invoked by process p where the key provided as an argument to π is k. Case 1: π returns `false` at line 7. If π returns `false` then there must be a configuration c that exists during the execution of π where k is in the persistent abstract set at c and we linearize π at a time corresponding to when k is in the persistent abstract set at c. To prove that such a time exists, assume that k is never in the persistent abstract set at any configuration that exists during the execution of π. If this were true either k is not in the volatile data abstract set at any configuration that exists between the invocation and response of π or k is contained in a node that is in the volatile data structure but the node is not durable at any configuration that exits exists between the invocation and response of π. Since the `insert` returned false *curr.key* was equal to k. This means that k must have been in the volatile abstract set since we found it via a traversal starting from the head. The `Find` invoked by π which returned *curr* guarantees that *curr* is durable. This means that there is a configuration that exists between the invocation and response of π where k is in the persistent abstract set. Case 2: π returns `true`. If π returns `true` then the DWCAS on line 19 succeeded inserting the node *newNode* by updating *p.next*. In this case we linearize π at the first successful DWCAS that sets the persistence bit in *p.next* while *p.next* points to *newNode*. This DWCAS must exist at some point before the response of π since the DWCAS performed by the *Persist* invoked on line 20 can fail if and only if some process other than p successfully performed an identical DWCAS setting the persistence bit in *p.next*. See lemma 37 for a proof of this claim.

Remove. Consider a `Remove` π invoked by process p where the key provided as an argument to π is k. Case 1: π returns `false`. If π returns `false` then there must be a configuration c that exists during the execution of π where k is not in the persistent abstract set at c and we linearize π at the time corresponding to when k is not in the persistent abstract set at c. To prove that such a time exists, assume that k is always in the persistent abstract set at any configuration that exists during the execution of π. If this were true the node n containing k must always be in the persistent data structure at every configuration that exists between the invocation and response of π. If n is in the persistent data structure then n is also in the volatile data structure. This means that the traversal performed by the `Find` invoked by π will end at n meaning it will be returned as *curr*. This means that the check on line 28 will fail which is impossible if π returns `false`. Case 2: π returns `true`. If π returns `true` then the DWCAS on line 20 of the HelpMarked function succeeded in physically deleting the node *nodeToDel* by updating *parent.next* to point to the successor of $nodeToDel$. In this case we linearize π at the first successful DWCAS that sets the persistence bit in $parent.next$ while $parent.next$ points to the successor of $nodeToDel$. This DWCAS must exist at some point before the response of π since the DWCAS performed by the *Persist* invoked on line 21 can fail if and only if some process other than p successfully performed an identical DWCAS setting the persistence bit in $p.next$.

Contains that Persist Data. Consider a `ContainsPersistLast` π invoked by process p where the key provided as an argument to π is k. Case 1: π returns `true`. If π returns `true` then there must be a configuration c that exists between the invocation and response of π where k is in the persistent abstract set at c and we linearize at the time corresponding to when k is in the persistent abstract set at c. The proof is the same as an `Insert` that returns `false`. Case 2: π returns `false`. If π returns `false` then there must be a configuration c that exists between the invocation and response of π where k is not in the persistent abstract set at c and we linearize at the time corresponding to when k is not in the persistent abstract set at c. The proof is the same as an `Remove` that returns `false`.

The linearization points and proofs for the other versions of the `Contains` function that perform persistence events follow the same structure.

Persistence Free Contains. Consider a `ContainsPersistFree` π invoked by process p where the key provided as an argument to π is k. Case 1: π returns `true` at line 13. In this case the node *curr* was found via a traversal starting from the head, the pointer into *curr* was durable and *curr* does contain k. This means that there is a time during the execution of π when k was in the persistent abstract set and we linearize at that time.

Case 2: π returns `false` at line 13. In this case the node *curr* was found via a traversal starting from the head and the pointer into *curr* was durable and *curr* does not contain k. This means that there is a time during the execution of π when k was not in the persistent abstract set and we linearize at that time.

Case 3: π returns `true` at line 26. In this case the node *curr* was found via a traversal starting from the head and it contains k, however, when we first traversed the pointer into *curr* it was not durable. After rereading, we found that either the pointer into *curr* or the old field in the predecessor of *curr* has changed. Apply the same structure as in the proof of lemma 37 and note that before the DWCAS the `Persist` function performs a psync. This means that if either of these fields changed then some process must have written *curr* to persistent memory. This means that there is a time during the execution of π when k was in the persistent abstract set and we linearize at that time.

Case 4: π returns `false` at line 26. This is the same as case 3 except that *curr* does not contain k. This means that there is a time during the execution of π when k is not in the persistent abstract set.

Case 5: π returns `false` at line 29. In this case we have not established the existence of a time where *curr* is durable and the old field in the predecessor of *curr* is iflagged. This means that *curr* is being inserted by a concurrent operation. We linearize π at any point after its invocation and before the CPE of the concurrent insert. Since we found that *curr* is not durable and the old field of the predecessor of *curr* is iflagged we know that this time exists at some point during the execution of π.

Case 6: π returns `true` at line 32. In this case we have not established the existence of a time where *curr* is durable but the old field in the predecessor of *curr* is not iflagged and *curr* contains k. This means that *curr* is not durable because a concurrent remove has physically deleted the last predecessor of *curr*. Applying the same argument as in case 3, we establish a time during the execution of π where *curr* is in the persistent data structure meaning k is in the persistent abstract set and we linearize π at that time.

Case 8: π returns `true` at line 35. In this case we have not established the existence of a time where *curr* is durable but the old field in the predecessor of *curr* is not iflagged. This means that *curr* is not durable because of an incomplete remove operation. Let p be the predecessor of *curr*. By theorem 35 the node *old* pointed to by *p.old* was the last durable value in *p.next*. If π returned `true` then *p.old* contained k. Since *p.old* was not *null* and *p.old* contains k then there is a time during the execution of π when k is in the persistent abstract set and we linearize π at that time.

Case 9: π returns `false` at line 35. This is the same as case 8 except *old* does not contain k. This means that there is a time during the execution of π when k is not in the persistent abstract set and we linearize π at that time.

I prove that the Physical-Del list maintains several invariants. First, I define some other necessary terminology used in the proofs.

Definition 24 (Dflagged)**.** *A node n is considered dflagged at configuration c if the dflag bit in $n.next$ is set to 1 and n is reachable via a traversal starting from the head in c.*

Definition 25 (Logically Deleted)**.** *A node n is considered logically deleted at configuration c if the marked bit in $n.next$ is set to 1 and n is reachable via a traversal starting from the root in c.*

Definition 26 (Physically Deleted)**.** *A node n is considered physically deleted at configuration c if n cannot be reached by a traversal starting from the root.*

Definition 27 (Consistency)**.** *The Physical-Del list is consistent with some persistent abstract set P if $\forall k \in P$, k is in a durable node in the volatile data structure.*

Invariant 28. *A node is never both logically deleted and dflagged.*

Invariant 29. *Once a node is marked its next pointer never changes.*

Invariant 30. *If the node n is in the volatile data structure at configuration c and n is logically deleted in c, then the predecessor of n is dflagged and the successor of n is not logically deleted in c.*

Invariant 31. *In a configuration c, if $n.next$ is not durable then the $n.old$ is the last durable pointer stored in $n.next$*

Invariant 32. *Consider an execution E of the Physical-Del list. Let c be the configuration after the last event in E. Let P_c denote the persistent abstract set at c and V_c denote the volatile abstract set at c. $P_t \setminus V_t$ is a subset of the keys that were part of remove operations that have no response in E and $V_t \setminus P_t$ is a subset of the keys that were part of insert operations that have no response in E.*

Theorem 33. *theorem 28 always holds for any configuration produced by an execution of the Physical-Del list.*

Proof. The invariant is trivially true for an empty list. When a new node is created the next pointer in the node is not marked or dflagged. This cannot change until the node is linked in the list such that it is reachable via a traversal from the head. The next pointer of any node is atomically updated by the DWCAS on line 55 of the *ContainsAsynchPersistAll* function, line 9 of the `Persist` function, line 19 of the `Insert` function, line 40 of the *Remove* function, line 20 of the *HelpMarked* function, line 34 of the *HelpRemove* function and line 9 of the *Persist* function. None of these DWCAS will update a node to be both marked and dflagged. □

Theorem 34. *theorem 29 always holds for any configuration produced by an execution of the Physical-Del list.*

Proof. No DWCAS will ever modify a marked pointer. The expected value of every DWCAS in the form $\langle n, o\rangle$ is always explicitly defined such that n is unmarked. □

Theorem 35. *theorem 31 always holds for any configuration produced by an execution of the Physical-Del list.*

Proof. The old field in any node n is updated to a non-`null` value by the DWCAS on line 19 of the `Insert` function or the DWCAS on line 20 of the `HelpMarked` function. Let $\langle e_1, \texttt{null}\rangle$ be the expected value of the DWCAS that updated $n.old$ to a non-`null` value and $\langle x_1, x_2\rangle$ be the new value. In both cases the x_2 is explicitly defined to be equal to e_1 and e_1 is explicitly defined to be a durable pointer. □

Theorem 36. *theorem 30 always holds for any configuration produced by an execution of the Physical-Del list.*

Proof. This is trivially true for an empty list and a list that contains only a single node. For a list containing two or mode nodes consider the following. If a node n is in the volatile data structure at configuration c then n is reachable via a traversal starting from the head. This means that the both the successor (if one exists) and predecessor of n are in the volatile data structure at c. n can only become logically deleted by the DWCAS on line 34 of the `HelpRemove` function. If an execution of `HelpRemove` reached this DWCAS then the condition of the `while` loop in `HelpRemove` must have been true. This means that there was a time during the execution of `HelpRemove` where the predecessor of n was dflagged. The only DWCAS that removes the dflag is the DWCAS on line 20 of the `HelpMarked` function. This DWCAS also physically deletes the successor of the dflagged node. This means that if the DWCAS succeeds, n will no longer be in the volatile data structure and while n is in the volatile data structure its predecessor is dflagged. In order for the successor of n to be logically deleted the DWCAS on line 34 of `HelpRemove` must succeed. The expected value of this DWCAS is explicitly constructed such that it is unflagged and unmarked on line 28. Since n is logically deleted the DWCAS will never succeed meaning the successor cannot be logically deleted. □

Lemma 37. *The DWCAS on line 9 of the `Persist` function invoked by process p will fail if and only if some process other than p successfully completes an identical DWCAS.*

Proof. Consider an execution of the `Persist` function by process p. Let n be the node such that the destination field of the DWCAS on line 9 of the `Persist` by process p is $\langle n.next, n.old \rangle$. Call this DWCAS d. The expected value of d is $\langle x, o \rangle$ where x is a non-durable pointer. d sets the persistence bit in $n.next$ and reverts $n.old$ to `null`. Assume p fails d. This means that the actual value of $n.next$ is either x' where x' is equivalent to x with the persistence bit set or some other value $y \neq x$.

In this case some other process p' must have concurrently updated n when $n.next$ was x. This means that p' must have performed an update involving n or an asynchronous-persist-all `Contains`.

If p' successfully completed the the DWCAS in asynchronous-persist-all `Contains` while $n.next$ was x then it is easy to see that this DWCAS is identical to d. The DWCAS in every other function (excluding `Persist`) has an expected value that is explicitly defined to be a durable next pointer and a `null` old pointer. This means that these DWCAS will always fail if the $n.next$ is not durable. Since x is not durable, any process that attempts a

DWCAS on n in any of the update or helper functions will fail. In each case, if the process fails the DWCAS it must retry the operation. This will eventually lead to the process invoking `Persist` with the first two arguments being n and x. Thus one of these processes will successfully complete a DWCAS identical to d. □

Theorem 38. *Update operations of the Physical-Del list change the volatile abstract set exactly once.*

Proof. We can identify each of the atomic instructions that update the volatile abstract set. Note that unsuccessful update operations do not change either the volatile data structure or the volatile abstract set. For insert operations the volatile abstract set is updated at the success of the DWCAS on line 19 of the `Insert` function. For remove operations the volatile abstract set is updated at the success of the DWCAS on line 20 of the `HelpMarked` function. These atomic instructions also correspond to the points at which the volatile data structure is updated. □

Theorem 39. *Update operations of the Physical-Del list change the volatile abstract set before changing the persistent abstract set.*

Proof. Consider an execution E of the Physical-Del list. Let c be the configuration after the last event in E. The persistent data structure at c contains all of the durable nodes in volatile data structure at c as well as any non-durable node in volatile data structure at c where the node is not durable because of a remove operation.

Update operations modify the volatile abstract set before they update the persistent abstract set. Insert operations modify the volatile abstract set at the success of the DWCAS on line 19 of the `Insert` function. Remove operations modify the volatile abstract set at the success of the DWCAS on line 20 of the `HelpMarked` function. In both cases the DWCAS by definition always happens before the CPE of the update operation.

If a crash occurs after a remove operation π succeeds the DWCAS on line 20 of `HelpMarked` but before its CPE then the persistent abstract set will still contain the key that π removed from the volatile abstract set. More precisely, let R be the set of remove operations that were incomplete in E. Let k_π be the key that was input to $\pi \in R$. For any $\pi \in R$ that completes the DWCAS on line 20 of `HelpMarked` but the CPE of π has not occurred k_π will not be in the volatile abstract set at c but k_π will be in the persistent abstract set at c.

Similarly, if a crash occurs after an insert operation π succeeds the DWCAS on line 19 of `Insert` but before its CPE then the the volatile abstract set will contain the key inserted by π but the persistent abstract set will not. □

Theorem 40. *Invariant 32 holds for any configuration produced by an execution of the Physical-Del list.*

Proof. This follows from theorem 39. □

Theorem 41. *`Insert`, `Remove` and `Contains` are lock-free.*

Proof. `Insert`, `Remove` and `Contains` all perform the same traversal of the list beginning from the head. In each case the traversal is a `while` loop that after finding a node contain a key greater or equal to the search key. Each iteration of the loop executes O(1) steps. If no concurrent process successfully performs the DWCAS on line 19 of `Insert` or line 20 of `HelpMarked` then the volatile data structure and the volatile abstract set is unchanged. This means that the loop will terminate in a finite number of steps. If a concurrent process successfully performs the DWCAS on line 19 of `Insert` then the loop could require one more iteration. This means that the traversal will complete in a finite number of steps or another process completes a DWCAS updating the volatile abstract set.

In the case of `Contains` operations or the `Find` function, the instructions performed after the traversal execute O(1) steps.

Update operations have a retry loop. One iteration of the loop executes O(1) steps. For `Insert` another iteration of the loop is required if the node p node returned by the `Find` function is marked or flagged or if the DWCAS on line 19 fails. If the DWCAS in `Insert` fails then some other concurrent process must have completed a DWCAS on p. If p is dirty then the `Insert` will help the concurrent update. The helper function `HelpMarked` executes O(1) steps. The helper function `HelpRemove` has a retry loop. A single iteration of this loop executes O(1) steps. Another iteration of the loop is required if the DWCAS on line 34 fails and the value returned is not marked. Since the condition of the loop checks that the node $nodeToDel$ is still the successor of the node $parent$, failing the DWCAS means that some other concurrent operation successfully performed a DWCAS on $parent$. If no other process performs a DWCAS on $parent$ then the loop in `HelpRemove` will terminate.

For `Remove` another iteration of the loop is required if the node p or the node $curr$ returned by the `Find` function is marked or flagged or if the DWCAS on line 40 fails. If the DWCAS in `Remove` fails then some other concurrent process must have completed a DWCAS on p. If p is dirty or $node$ is dirty the `Remove` will help complete the concurrent update requiring the same steps as described for the case of `Insert`. □

Theorem 42. *The Physical-Del list is always consistent with some persistent abstract set P.*

Proof. From theorem 38 we know that the volatile abstract set is changed exactly once by any update operation. It follows from theorem 39 that the Physical-Del list will be consistent with P since every completed successful update operation will be reflected in P along with some of the pending update operations. □

theorem 42 and theorem 41 and the way in which I choose the linearization points for the Physical-Del list collectively prove theorem 18.

Proving the SWCAS implementations is significantly more involved due to the extra volatile memory synchronization. A proof of the SWCAS implementations could be constructed similarly to the proof of the Physical-Del list. The main difference would be the proof of theorem 35 since a node's old pointer is updated independently of its next pointer.

6.5 Generic Extended Link-and-Persist Correctness

For most data structures it is not difficult to be convinced that the original link and persist technique provides durability linearizability. David et al. provide a brief discussion of correctness for the original link and persist technique. They note that the link and persist technique provides two properties 1) every update ensures that its modifications are durable before returning and 2) every operation ensures that its dependencies are durable before making any changes. They claim that together these properties guarantee durable linearizability. There are some nuances in by these proprieties that I would like to have more clearly defined. In the first property, this is the idea of making a change durable. To say that a change or modification is durable is to say that the write that updated the value of some object has been successfully flushed to persistent memory. For the second property we need to define what it means for an operation to be durable. I have already defined this for the model used in this thesis when I defined the CPE of an operation. Even with these clearly defined properties, we do not yet guarantee durable linearizability without the need for other assumptions or implementation constraints. Consider a data structure which uses a double-compare-single-swap (DCSS) primitive to perform updates. Suppose that the DCSS utilizes a descriptor and a pointer to the descriptor is installed in every field that is part of the DCSS. ([12] is an example of a data structure that uses DCSS in this way). If a data structure link l is updated via a DCSS then the link will point to a DCSS descriptor until the DCSS completes. The link and persist technique requires that l is flushed to persistent memory. If the descriptor is never flushed to persistent memory then we have a problem. Suppose a system crash occurs immediately after l is updated to point to the DCSS descriptor. Since the descriptor itself was never flushed, the DCSS cannot be completed and it cannot be undone since we do not know what l pointed to before the DCSS descriptor was installed. This means that the recovery procedure might fail to recover operations that completed before the crash, violating durable linearizability. The obvious solution is to flush the contents of the descriptor. In practice this is simple assuming the descriptor fits into a single cache line which is likely to be true for a DCSS. However, other primitives such as a K-CAS [36], could easily span more than one cache line, introducing further complexities that require extra flushes. In general, the link and persist technique requires that any data that a data structure link can point to must be flushed to persistent memory before the link is updated to point to the data. David et al. also claim that their data structures are linearizable since they start from linearizable objects and add only flushes. I agree that the data structures from [24] are indeed linearizable, however, the link and persist technique does add more than just flushes. We have already noted that the link and persist technique requires that operations help persist their dependencies. To perform this efficiently, this requires extra volatile memory synchronization, which, if implemented incorrectly could break both linearizability and lock-freedom.

Proving correctness in general for the extended link and persist technique does not require much more than the original link and persist technique. Operations must still guarantee that they persist their changes before returning and that they help persist any dependencies before making modifications. Likewise, we require that any data that a data structure link can point to is written to persistent memory before the link is updated to point to it. We also need to restrict how the implementation uses the old field. Update operations must guarantee correct usage of the old field. This means that updates can only update the old field to a non-`null` value after guaranteeing that the corresponding link is persistent. Likewise, updates can only update the old field back to `null` after attempting to persist the corresponding link.

Chapter 7

Evaluation

I present an experimental analysis of my persistent list compared to existing persistent lists on various workloads. I test the variants of the `contains` operation separately meaning no run includes more than one of the variants. To distinguish between my implementations of the `contains` operation I prefix the names of my persistent list algorithms with the abbreviation of a `contains` variant. I test the performance of these lists in terms of throughput (operations per second). I also examine the psync behaviour of these algorithms. Specifically, I track the number of psyncs that are performed by searches and the number of psyncs that are performed by update operations.

All of the experiments were run on a machine with 48 cores across 2 Intel Xeon Gold 5220R 2.20GHz processors which provides 96 available threads (2 threads per core and 24 cores per socket). The system has a 36608K L3 cache, 1024K L2 cache, 64K L1 cache and 1.5TB of NVRAM. The non-volatile memory modules installed on the system are Intel Optane DCPMMs. The machine is running Ubuntu 20.04. I utilize the same benchmark as [12] for conducting the empirical tests. I modify the existing work such that all of the data structures that I test utilize the same allocation and reclamation algorithms. Note that for the Link-and-Persist list I only test the list algorithm alone. The link-cache and NV-epochs mechanisms are disabled. Excluding the variants of my list that rely on asynchronous flush instructions, all of the other data structures use the `clflush` instruction to flush data from the volatile cache to persistent memory. Where asynchronous flush instructions are necessary the `clflushopt` instruction is utilized.

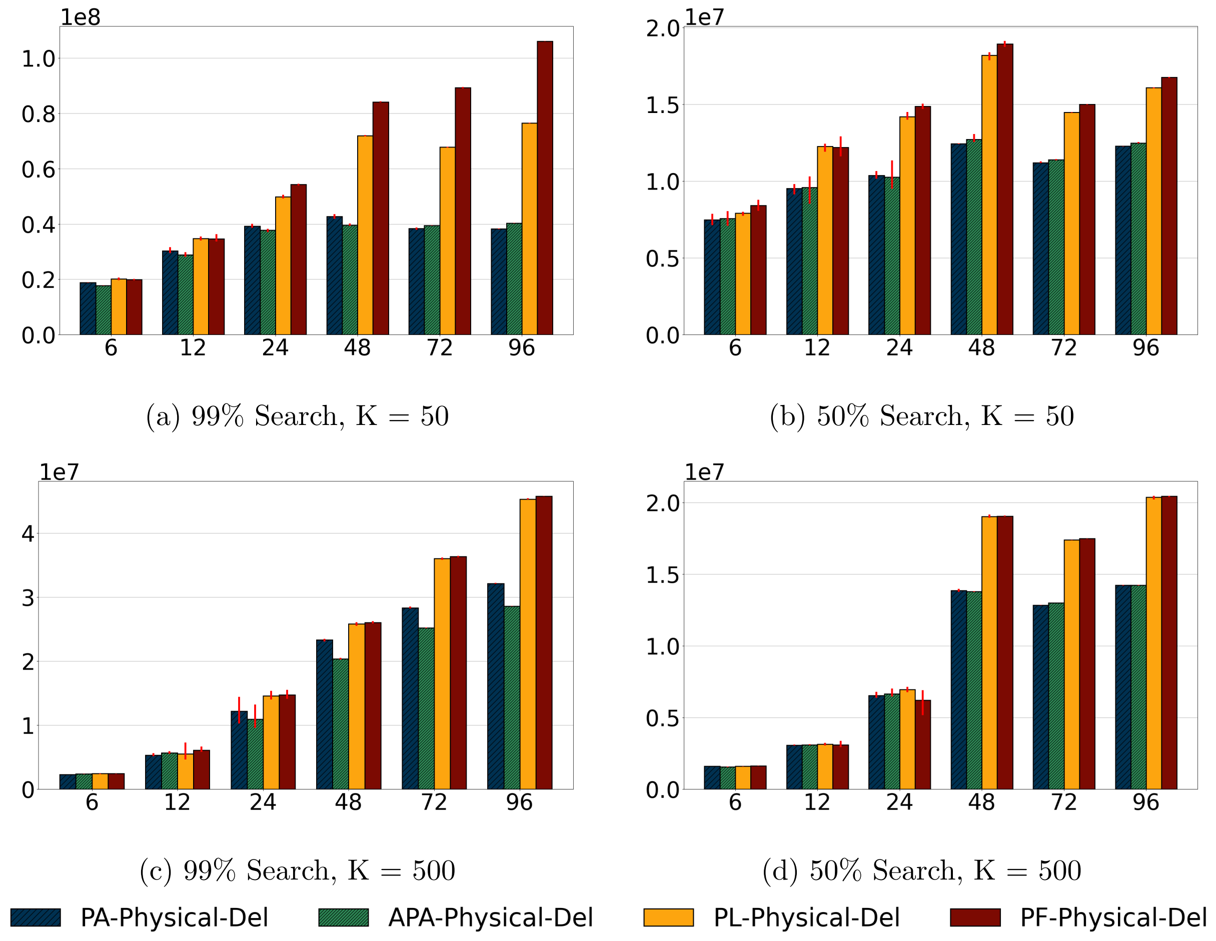
Accessing Persistent Memory. The persistent memory is exposed to the programmer via memory-mapped files. A filesystem which supports direct access (DAX) is mounted on the persistent memory device. I utilize the Intel's memkind library [14] to allocate memory within the address space of the memory-mapped files.[1] Internally the memkind library utilizes jemalloc which is known to perform well on NUMA systems. Note that memkind is not a persistent allocator meaning it does not persist data structures utilized internally by the allocator. This is beneficial since it means the allocator does not add any extra psyncs. On the other hand this does leave open the possibility of persistent memory leaks and ignores the possibility of the virtual address space space changing between restarts. This work is not concerned with issues related to persistent memory leaks which would require the use of a persistent allocator. If the virtual address space of the memory-mapped files can change after a restart extra metadata would need to be persisted. This is true of all of the algorithms that I consider. One approach to preventing problems related to changes in virtual address mapping is to persist offsets as in Intel's libpmemobj library [22]. Efficient persistent memory allocation is a complex problem on its own. It would be possible to modify memkind to ensure that it persists its own metadata similar to the modifications proposed in [24] where jemalloc was configured such that the thread caches are flushed (however the thread caches were not mapped to persistent memory which is obviously a problem).

7.1 Throughput

Figure 7.1 shows the throughput of my DWCAS Physical-Del list variant. Unsurprisingly, the persist-last (PL) and persistence-free (PF) variants perform better than the persist-all (PA) and asynchronous-persist-all (APA) variants. For this reason I omit the persist-all and and asynchronous-persist-all from all other figures. Figure 7.2 shows the throughput of my SWCAS implementations compared to my DWCAS implementations. I observe that the DWCAS implementations of my list out performed the SWCAS implementations.

Figures 7.3-7.5 show the throughput of my best persistent list variants compared to the existing algorithms. The existing approach of [58] performs best when the maximum size of the list is lower than the maximum number of concurrent threads. In all other cases, my persistent list is comparable or better than the existing approaches especially when the list is larger than the maximum number of concurrent threads. Note that I are not able to replicate the same throughput reported in [58] on the evaluation system even when using the author's implementation and benchmark.

[1]The memkind library was recently updated to support persistent memory. In the past it was used as a volatile allocator and it automatically removes the memory-mapped files when the client program terminates. On Linux, memkind accomplishes this with the `unlink` system call. To retain the memory-mapped files I would remove the call to `unlink`.



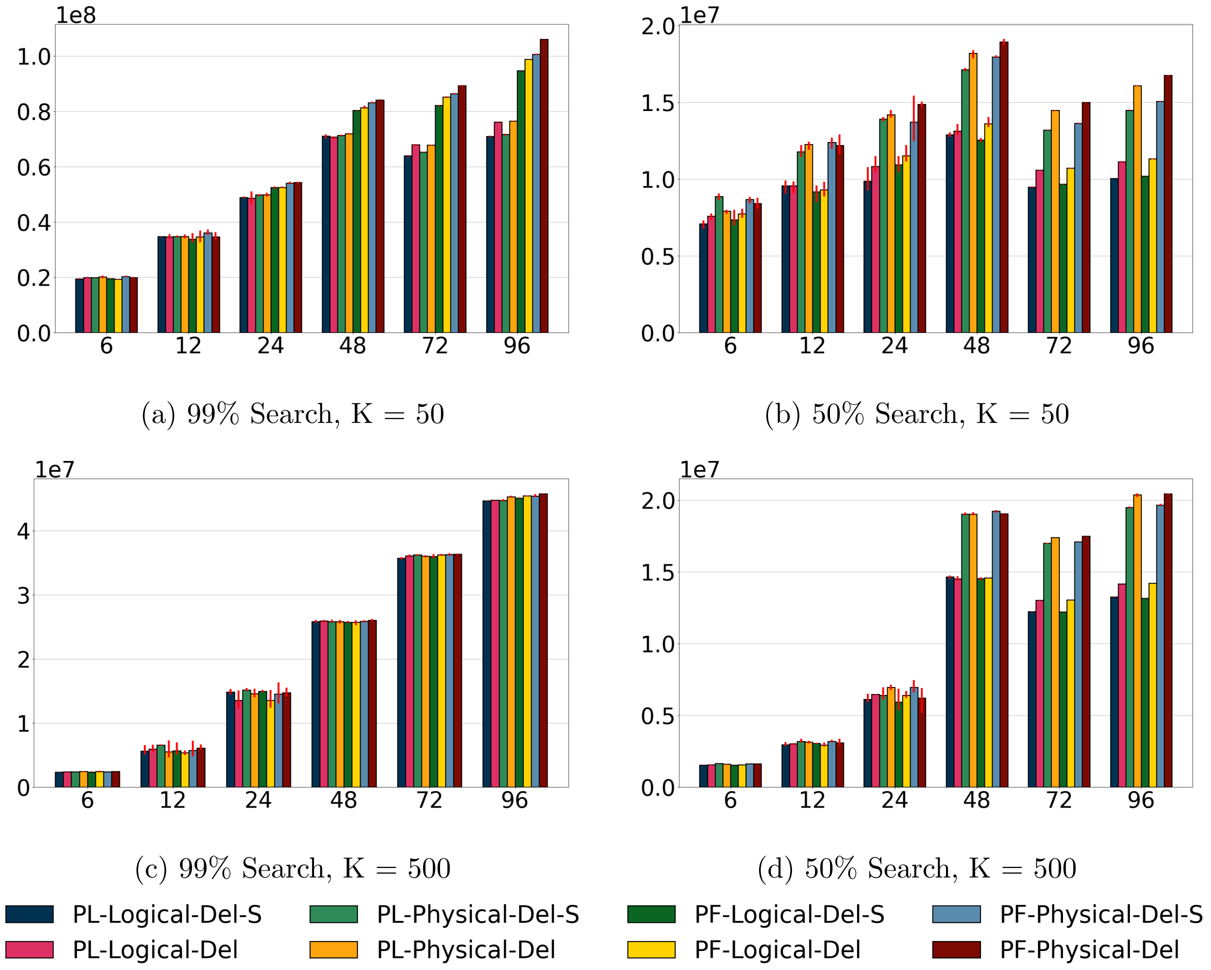
(a) 99% Search, K = 50

(b) 50% Search, K = 50

(c) 99% Search, K = 500

(d) 50% Search, K = 500

Figure 7.1: Persistent list throughput for the different variants of DWCAS Physical-Del implementations. Y-axis represents operations per second. X-axis represents the number of concurrent threads. K is the maximum key.



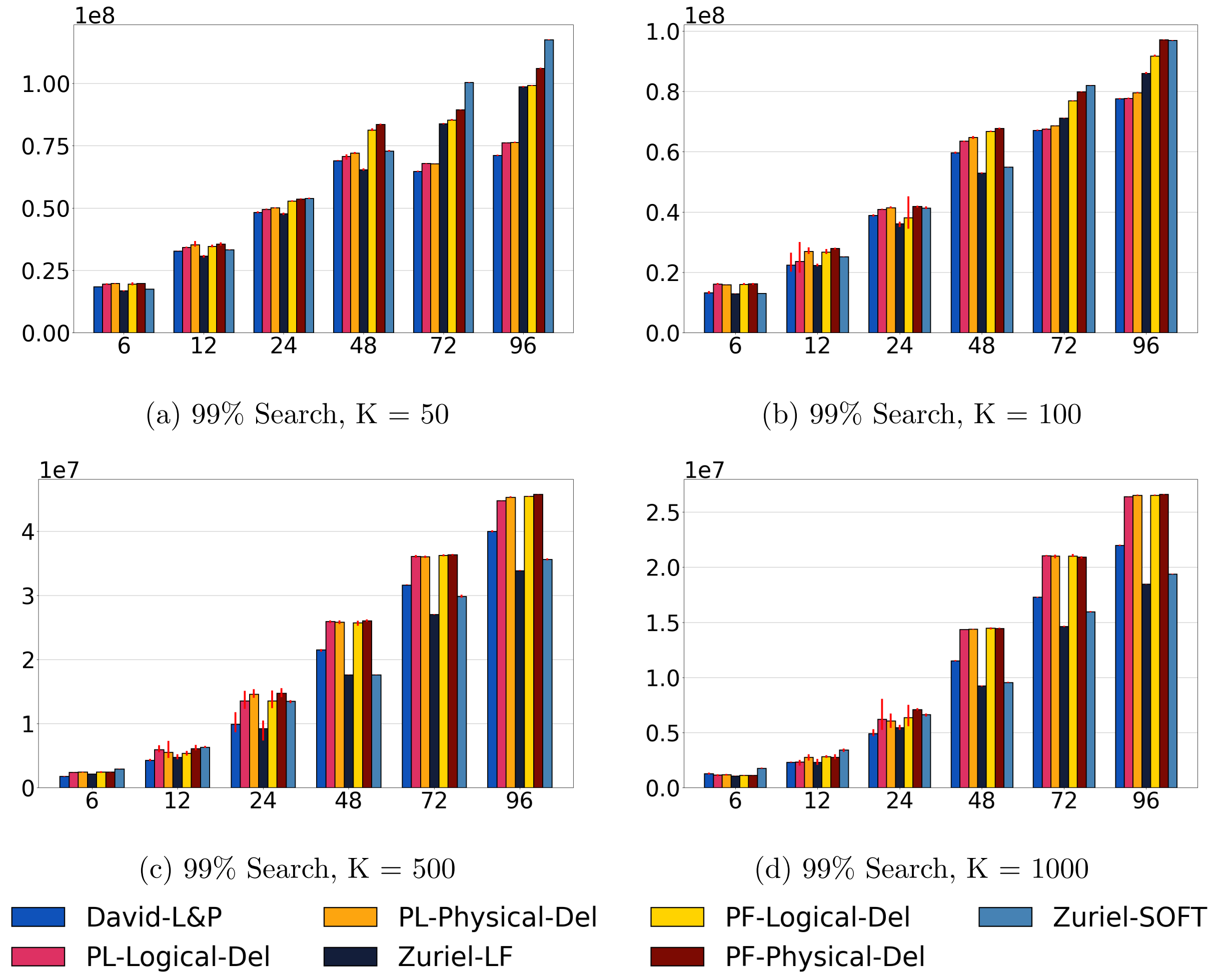
(a) 99% Search, K = 50

(b) 50% Search, K = 50

(c) 99% Search, K = 500

(d) 50% Search, K = 500

Figure 7.2: Persistent list throughput for my SWCAS implementations compared to DW-CAS implementations. Y-axis represents operations per second. X-axis represents the number of concurrent threads. K is the maximum key.



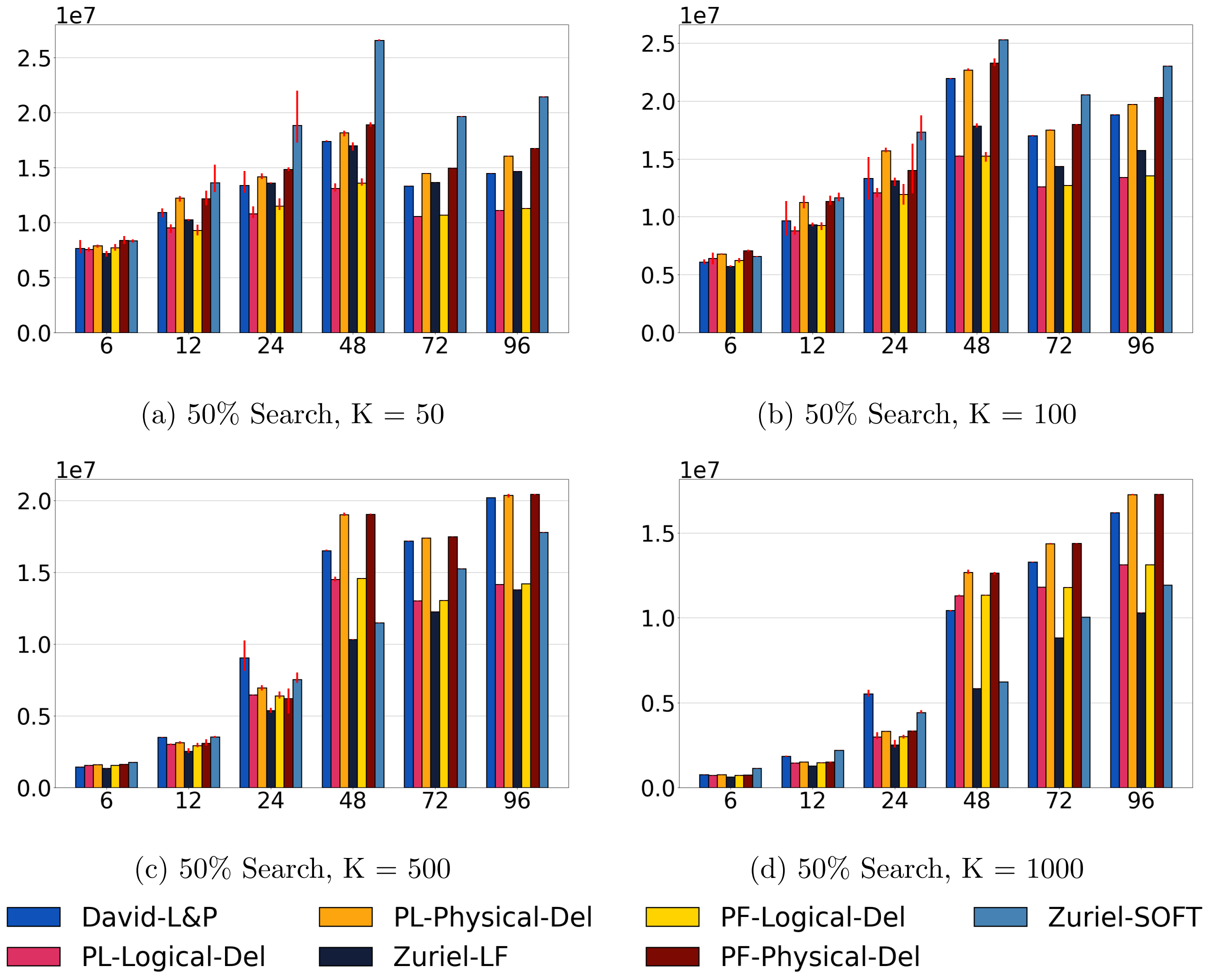
(a) 99% Search, K = 50

(b) 99% Search, K = 100

(c) 99% Search, K = 500

(d) 99% Search, K = 1000

Figure 7.3: Persistent list throughput for 99% search workload. The y-axis represents operations per second. The x-axis represents the number of concurrent threads. K is the maximum key.



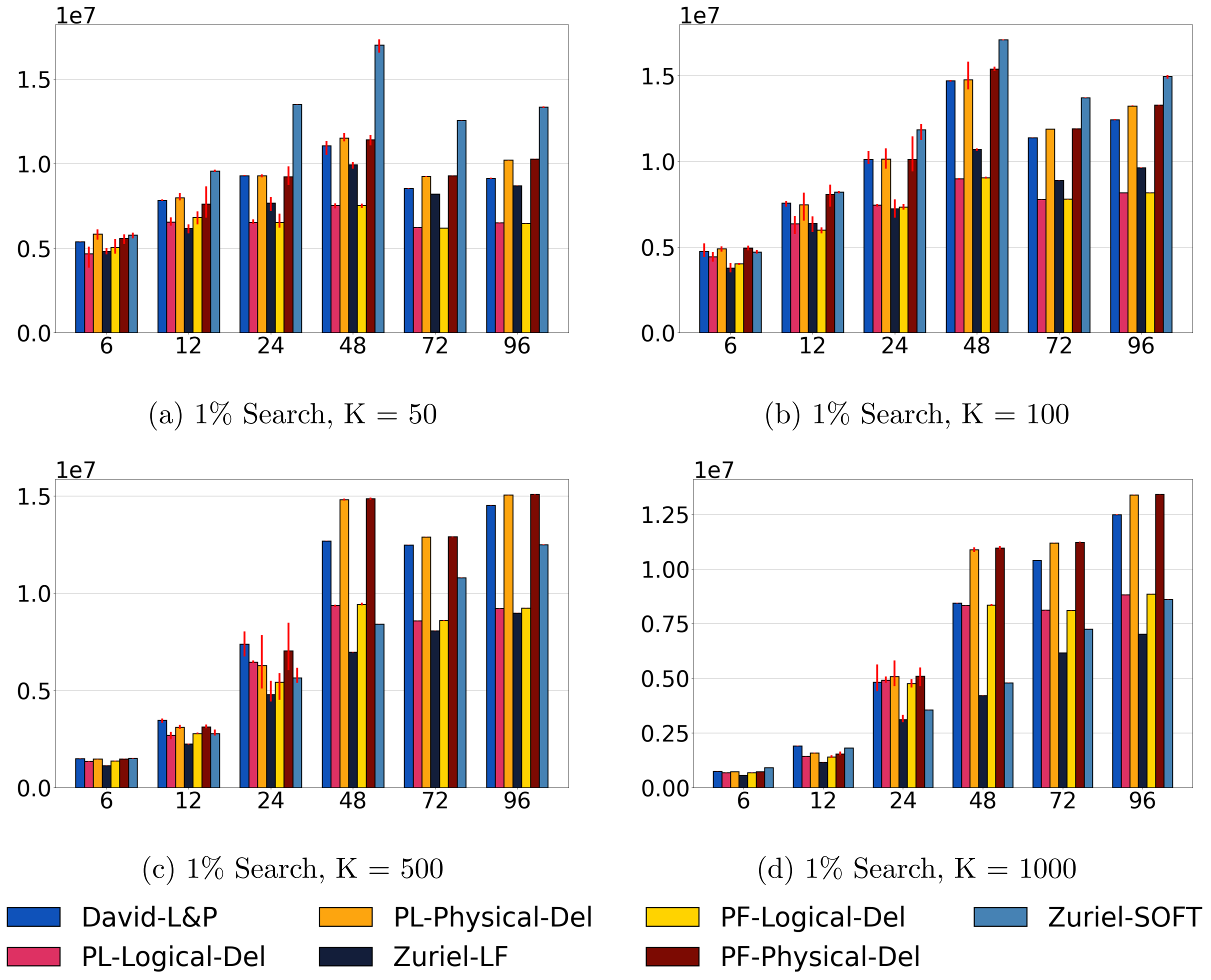
(a) 50% Search, K = 50

(b) 50% Search, K = 100

(c) 50% Search, K = 500

(d) 50% Search, K = 1000

Figure 7.4: Persistent list throughput for 50% search workload. The y-axis represents operations per second. The x-axis represents the number of concurrent threads. K is the maximum key.



(a) 1% Search, K = 50

(b) 1% Search, K = 100

(c) 1% Search, K = 500

(d) 1% Search, K = 1000

Figure 7.5: Persistent list throughput for 1% search workload. The y-axis represents operations per second. The x-axis represents the number of concurrent threads. K is the maximum key.

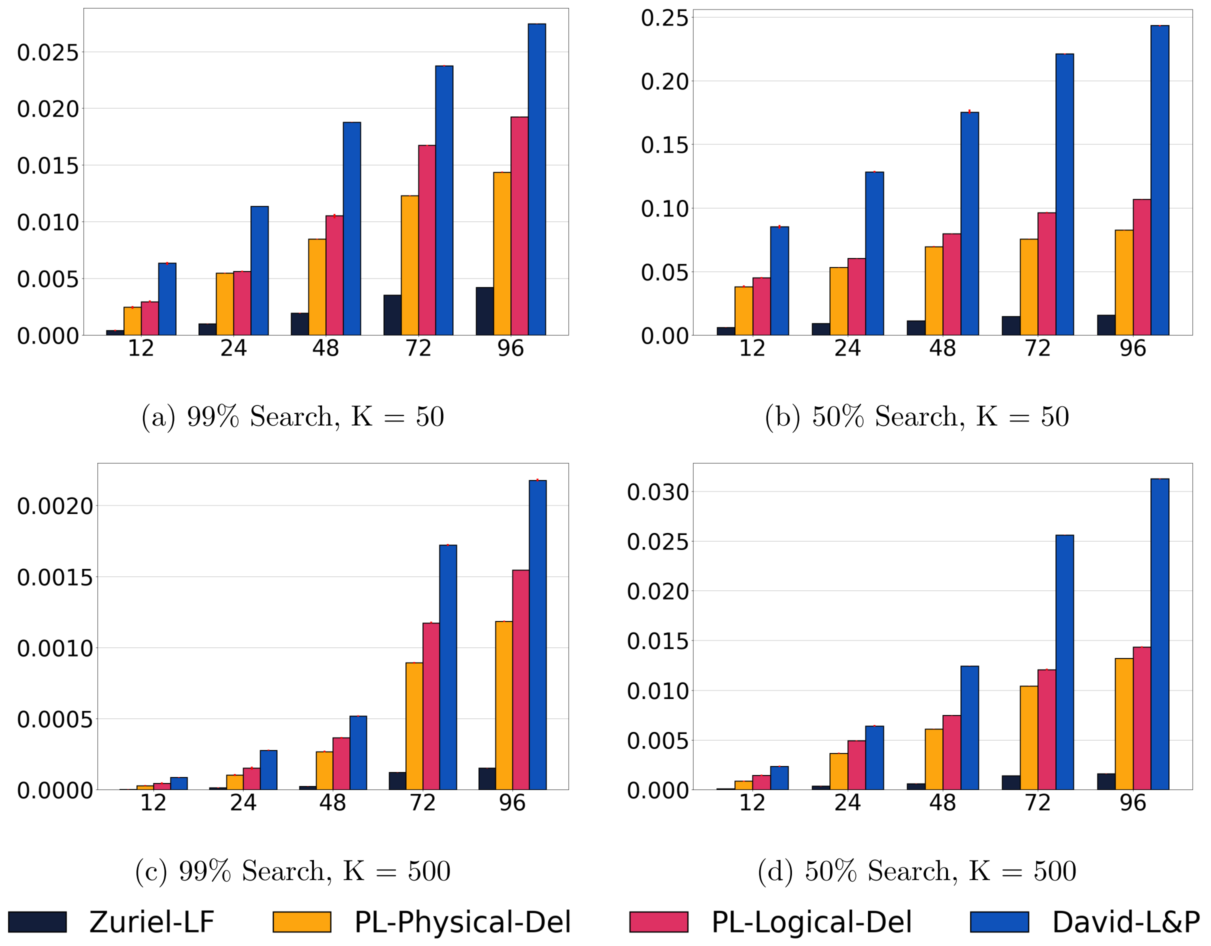
7.2 Psync Behaviour

The recent trend to persist less data structure state has influenced implementations of persistent objects focused on minimizing the amount of psyncs required per operation. We know that strict linearizable algorithms cannot have persistence-free searches without allowing an unconstrained recovery procedure. From [17] we also know that update operations require at least 1 psync. Of the persistent lists that I consider, the persistent lists from [58] are unique in that the the maximum number of psyncs per update operation is bounded. While these results are theoretically interesting, the question of whether or not this matters in practice is unanswered. To better understand the cost incurred by psyncs, I track the number of psyncs performed by search operations (`contains`) and the number of psyncs performed by update operations. Figure 7.6 and figure 7.6 show the average number of pysncs per search and the average number of pysncs per update operation. I observe that searches rarely perform a psync in any of the algorithms that do not have persistence-free searches. Similarly, on average, update operations do not perform more than the minimum number of required psyncs.

Observation: Algorithmic techniques such as *persistence bits* for reducing the number of psyncs are highly effective. A large amount of contention is required for the optimized algorithms to perform extra psyncs. On average, there are very few redundant psyncs in practice.

7.3 Recovery

It is not practical to force real system crashes in order to test the recovery procedure of any algorithm. One could simulate a system crash to evaluate the recovery procedure. It is also possible to evaluate the recovery procedure as a standalone algorithm by running it on an artificially created memory configuration. This is problematic because the recovery procedure of a durable linearizable algorithm is often tightly coupled to some specific memory allocator. This is true of the existing algorithms that I consider. This makes a fair experimental analysis of the recovery procedure difficult. It is easier to describe the worst case scenario for recovering the data structure for each of the algorithms. To be specific, I describe the worst case persistent memory layout produced by the algorithm noting how this relates to the performance of the recovery procedure. First, I briefly introduce the idea of memory *chunks*. It is common for memory allocators to request memory from the operating system in sizes significantly larger than the size of any data structure objects. We say that the memory is allocated in *chunks*. The size of a single



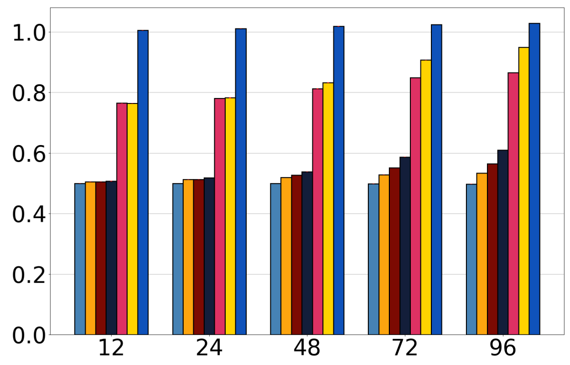
(a) 99% Search, K = 50

(b) 50% Search, K = 50

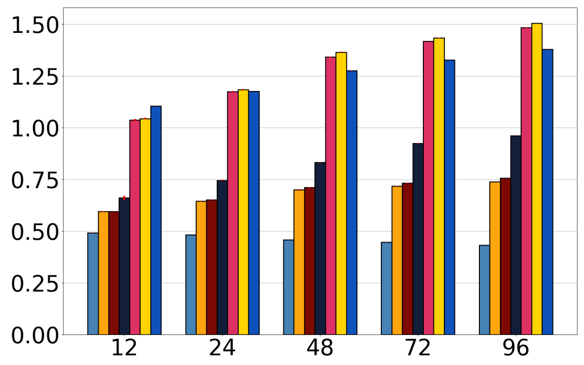
(c) 99% Search, K = 500

(d) 50% Search, K = 500

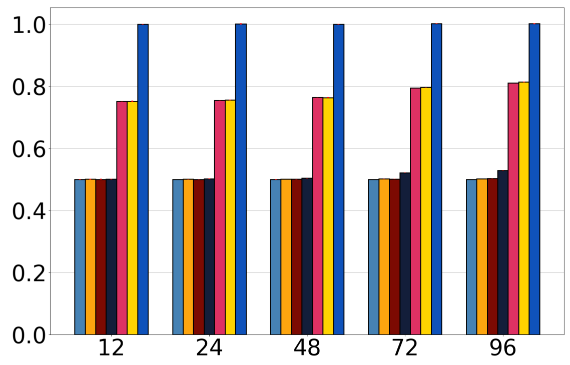
Figure 7.6: Psync Behaviour. Y-axis represents average psyncs per search. X-axis represents the number of concurrent threads. K is the maximum key.



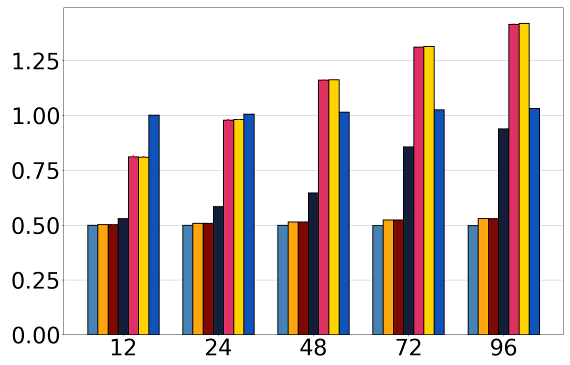
(a) 99% Search, K = 50



(b) 50% Search, K = 50



(c) 99% Search, K = 500



(d) 50% Search, K = 500

Figure 7.7: Psync Behaviour. Y-axis represents average psyncs per update. X-axis represents the number of concurrent threads. K is the maximum key.

memory *chunk* depends on the allocator. In some cases the recovery procedure is forced to traverse all allocated memory. If this is the case, then the size of a memory chunk directly effects the recovery complexity. The ability to determine if a chunk is empty will also effect the recovery complexity. I now describe the recovery complexity in terms of the worst case memory layout for each of the algorithms that I evaluate.

The Link-Free list does not persist data structure links. While the links of the data structure could be written to persistent memory by a background flush since a link is never explicitly flushed to persistent memory we cannot know if the link points to a valid node. As a result, there is no way to efficiently discover all valid nodes meaning the recovery procedure might need require traversing all of the memory. The memory allocator used by [58] requests memory in chunks.[2] We can construct a worse case memory layout for the recovery procedure as follows: Suppose that we completely fill persistent memory by inserting keys into the list. Call the total number of nodes that fit in memory M. M corresponds to the peak size of the list. These M nodes will be stored in C full memory chunks. Now suppose that we remove nodes from the data structure such that every memory chunk contains only one node at an unknown offset from the start of the chunk. Call the number of nodes still in the list N. Since no chunk is empty, in order to discover the N valid nodes, the recovery procedure must traverse the entire contents of all C memory chunks.

The SOFT list also does not persist data structure links. The requirements of the recovery procedure for SOFT list is the same as the Link-Free list. We can construct the worst case memory layout for the recovery procedure in the same way as I did for the Link-Free list yielding the same asymptotic time complexity.

The Link-and-Persist list persists data structure links. Marked pointers are also explicitly flushed meaning persistent memory can contain logically deleted nodes. Similarly, a pointer marked as not persistent might have been written to persistent memory by a background flush. The recovery procedure does not need to cleanup any of these marked nodes since this can be done by any other operation. This means that the Link-and-Persist list can support an empty recovery procedure. The time complexity of this traversal is $\Theta(N)$ for a list containing N nodes. The actual recovery procedure implemented by the authors of [24] traverses memory to free any nodes that are no longer reachable starting from the root.[3]

[2]The allocator used by [58] is SSMEM [25] which is actually a volatile memory allocator. SSMEM would require various modifications to work with real persistent memory.

[3]The original implementation of the recovery procedure relies on the author's memory management algorithm which they call NV-epochs. The NV-epochs algorithm relies on a version of jemalloc which does not support allocating memory from the address space of memory-mapped file. This would require various

I utilize DWCAS and asynchronous flush instructions to achieve a minimum of one psync per `insert` operation. There are some subtleties with this implementation that prevents the use of an empty recovery procedure. In the worst case, the recovery complexity is $O(N + n)$ for a list containing N nodes and a maximum of n concurrent processes. Implementations that use SWCAS (or DWCAS allowing for a minimum of two psyncs per `insert`) can use an empty recovery procedure.

Observation: If structure is persisted, recovery can be highly efficient. Without any persisted structure, recovery must traverse all of shared memory with little or no guidance. Memory reclamation drastically complicates the issue of recovery.

7.4 Strict linearizable vs. Durable linearizable Algorithms

I have demonstrated that that there exists a theoretical separation between strict linearizable and durable linearizable sets when the recovery procedure is constrained. For persistent lists I observe that this separation does not lead to significant performance differences in practice. 4 of the algorithms shown in figures 7.3-7.5 are strict linearizable. Specifically, the list implemented using the Link-and-Persist technique, the Link-Free list and both of the variants of my persistent list that use the persist-last `contains` operation are strict linearizable. The SOFT list, and the two variants of my list that use persistence-free searches are durable linearizable. The high cost of a pysnc and the impossibility of persistence-free searches in a strict linearizable lock-free algorithm would suggest that the strict linearizable algorithms that I test should perform noticeably worse. In practice, it is true that for most of the workloads that I test, the algorithms that have persistence-free searches perform best. However, for many of the workloads, the performance of the strict linearizable algorithms is comparable to the durable linearizable algorithms. In fact, for some workloads my strict linearizable algorithms perform better than the SOFT list.

Observation: Strict linearizable algorithms can be just as fast as (or *faster than*) durable ones, despite any theoretical tradeoffs.

modifications to work with real persistent memory.

Chapter 8

Conclusion

8.1 Summary

In this work I presented two lower bounds. First, I proved that it is impossible to implement strict linearizable lock-free sets with persistence-free searches for the classes of strict linearizable implementations considered in this thesis. This seemed to align with existing work which demonstrated that durable linearizable lock-free sets that did support persistence-free searches outperformed other existing strict linearizable lock-free sets. Second, I proved that for any implementation of a durable linearizable lock-free set there must exist an execution in which some process performs a *redundant* psync as part of an update operation. This bound as seems to agree with trends in existing work which emphasizes persisting less data structure state in favour of performing fewer psyncs. I also presented several implementations of persistent sets and evaluated these implementations against existing persistent sets. I found that in practice, when the recovery procedure is constrained to produce configurations that existed before the crash strict linearizable sets often perform as well as durable linearizable sets. I observed that redundant psyncs rarely occurred during my evaluations. This suggests that psync complexity is not a good predictor of performance in practice. This contrasts the trends of existing work and suggests that there is a need to look for better metrics to compare persistent objects. Based on the results, I recommend that researchers should not immediately sacrifice strict linearizability by prioritizing persistence-free reads. Instead, I recommend that researchers should begin with strict linearizable implementations since a strict linearizable implementation may not have much overhead and it may be sufficient for the application. The results also suggest that persisting extra data structure state can be beneficial.

8.2 Future Work

Memory Management. Memory management is an important factor that effects both recovery and runtime performance. I only briefly discussed the topic of memory allocation and reclamation in this work. Even without the concerns related to persistent memory, safe and efficient memory allocation and reclamation is a complex problem. Solutions for volatile memory have been studied in detail e.g. [13, 9, 45]. Persistent memory adds a new layer of complexity. In particular, the possibility of persistent memory leaks is an important problem to consider. Designing an efficient persistent allocator that addresses these issues would greatly benefit further implementations of persistent objects.

Better Persistent Universal Construction. In chapter 3 I described some universal constructions that convert volatile concurrent objects to persistent concurrent objects. Typically these perform significantly worse than hand-crafted solutions. While hand-crafted solutions usually perform better compared to universal constructions, they can be difficult to implement and might not generalize between different abstract data types. It would be interesting to explore creating a more efficient universal construction capable of producing efficient persistent concurrent objects.

References

[1] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

[2] Marcos Kawazoe Aguilera, Svend Frølund, Vassos Hadzilacos, Stephanie Lorraine Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, pages 23–32, 2007.

[3] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, 2018.

[4] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. *ACM SIGPLAN Notices*, 46(1):487–498, 2011.

[5] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[6] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proceedings of the VLDB Endowment*, 10(4):409–420, 2016.

[7] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware transactions in nonvolatile memory. In *International Symposium on Distributed Computing*, pages 617–630. Springer, 2015.

[8] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[9] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 33–42, 2013.

[10] Anastasia Braginsky and Erez Petrank. A lock-free b+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 58–67, 2012.

[11] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.

[12] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–291, 2020.

[13] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

[14] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

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

[16] Nachshon Cohen, Michal Friedman, and James R Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–24, 2017.

[17] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 259–269, 2018.

[18] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent

memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[19] Intel Corporation. Intel memory latency checker. Available at: https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker. [last accessed: July 1, 2021].

[20] Intel Corporation. Intel optane persistent memory. Available at: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory. [last accessed: July 2, 2021].

[21] Intel Corporation. Intel/micron 3d-xpoint non-volatile memory. Available at: https://www.intel.com/content/www/us/en/architectureand-technology/intel-micron-3d-xpoint-webcast. [last accessed: July 1, 2021].

[22] Intel Corporation. Pmdk: The libpmemobj library. Available at: https://pmem.io/pmdk/libpmemobj/. [last accessed: July 1, 2021].

[23] Intel Corporation. Intel® 64 and ia-32 architectures optimization reference manual, Jun 2016.

[24] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, 2018.

[25] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News*, 43(1):631–644, 2015.

[26] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[27] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[28] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[29] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[30] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, 2004.

[31] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. Nvtraverse: in nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 377–392, 2020.

[32] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

[33] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1218–1232, 2021.

[34] Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407. IEEE, 2004.

[35] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[36] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.

[37] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.

[38] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS). Citeseer*, page 103. Citeseer, 2006.

[39] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[40] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[41] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

[42] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[43] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411, 2016.

[44] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1):143–143, 2010.

[45] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[46] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[47] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

[48] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 385–398, 2013.

[49] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[50] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276. IEEE, 2014.

[51] Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[52] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

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

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

[55] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(2):249–282, 1989.

[56] Tangliu Wen, Jie Peng, Jinyun Xue, Zhen You, and Lan Song. Strict linearizability and abstract atomicity. *International Journal of Foundations of Computer Science*, 32(01):1–35, 2021.

[57] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

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