Detectable Data Structures for Persistent Memory

by

Nan Li

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

Waterloo, Ontario, Canada, 2021

Author's Declaration

Abstract

Persistent memory is a byte-addressable and durable storage medium that provides both performance benefits of main memory and durability of secondary storage. It is possible for a data structure to recover near-instantly after a system failure by accessing recovery data directly in persistent memory through memory operations. A variety of researches have been working on building persistent data structures for persistent memory. Some persistent data structures are said to be detectable, which means they can tell whether the last operation invoked before crash took effect or not. In this thesis, I propose an abstract data type *DetectableT* with its sequential specification, which can be composed with a base data type to make the base data type detectable. To show how to design detectable data structures based on *DetectableT*, a detectable lock-free queue algorithm called Detectable Queue, which composes *DetectableT* with Queue, is presented. One difficulty in the implementation of Detectable Queue is to get the result of a compare-and-swap (CAS) operation after a crash since the result of CAS is stored in volatile CPU registers. To help detectable data structures handle this common problem, I provide a synchronization primitive called CASWithEffect, which executes a CAS operation and stores the result into persistent memory atomically using private variables. With CASWithEffect, another detectable queue algorithm called CASWithEfffect Queue is provided as a substitute for Detectable Queue with a simpler design. Regarding correctness, I prove that both Detectable Queue and CASWithEffect satisfy strict linearizability. The data structure implementations are evaluated using Intel Optane Persistent memory. I compare both Detectable Queue and CasWithEffect queue with another queue algorithm - Log Queue. The result shows that Detectable Queue has the best performance.

Acknowledgements

I would like to thank my supervisor, Professor Wojciech Golab, for his guidance in the selection of a suitable topic and how to proceed with correctness proof and experiments to support the topic.

Dedication

This is dedicated to my mom, Ji Huang, who supports me all the time unconditionally and my daughter Iris who is the biggest joy to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past years, there has been intensive research based on byte-addressable non-volatile memory (NVM), which provides performance comparable to volatile DRAM and preserves its content under transient failures (e.g., power outages). Before the emergence of persistent memory, the traditional computer architecture included volatile byte-addressable main memory (e.g., DRAM) and non-volatile block-addressable secondary storage (e.g., hard disk). This lead to a separation of in-memory data structures and sequential on-disk structures for recovery, such as transaction logs. Eventually, through a single layer in the memory hierarchy, NVM offered the performance benefits of main memory and durability of secondary storage. Unlike secondary storage, NVM can be accessed directly by normal load and store instructions. Instead of slowly transferring logs to secondary storage blocks, the system can access recovery data in NVM through memory operations. Therefore, it potentially allows for almost instant recovery for any data structure after a system failure. This leads to increasing interest in how to design recoverable and durable in-memory data structures working specially with persistent memory.

To take advantage of persistent memory's non-volatility, the data stored in persistent memory have to be in a state which can be recovered to a consistent state after a system crash. One major challenge for designing durable data structures for persistent memory is that the CPU caches still remain volatile. There is no guarantee on when or in what order the data in the caches are written back to persistent memory. A thread may read a value which has not been written back to persistent memory, which may be lost in the case of crash and cause inconsistency. To ensure consistency, specific instructions, such as cache line flush instructions and memory fences, must be used to enforce order and explicitly write a cache line back to persistent memory.

There has been a lot of research work on developing persistent data structures. Some work focuses on building algorithms by leveraging various forms of logging to implement transaction [8–10, 23, 26, 27], while others aim at removing logging entirely to reduce overhead and implementing recoverable and concurrent data structures [6, 11–13, 29, 30, 34, 36]. Some of these persistent data structures [4–6, 13] are said to be also detectable. From [13], a detectable data structure provides a mechanism to determine whether the last operation executed before crash takes effect or not upon recovery. Friedman et al. [13] presents a detectable log queue algorithm, which obtains an operation's status by storing the response of each operation into persistent memory. All of the above detectable data structures request to save additional bookkeeping data so that recovery can obtain the status of the last invoked operation before crash. Based on this feature, I propose an abstract data type $DetectableT$, which can be composed with a base data type (e.g., Queue) to make the base data type detectable, and provide the definition of Detectability based on $DetectableT$. This is the first definition for detectability through a specific data type. Defining detectability through a data type makes it more generic and requiring less stringent system assumptions. In [4], Attiya et al. proposed a linearizability property called nesting-safe recoverable linearizability (NRL) to define detectability. Their definition requires stronger system assumptions, which include saving instructions into persistent memory. Compared to NRL, the definition of detectability in this thesis does not need such strong system assumptions and can be implemented on current hardware system as shown in Chapter 9.

In the first contribution of this thesis, I provide the sequential specification of $DetectableT$. With respect to whether the arguments or returned value of an operation exists, I divide operations into four types. For each operation type, the sequential specification of $DetectableT$ is presented with different bookkeeping data. For example, for an operation without arguments or returned value, the sequential specification only includes the status variable which shows whether the operation has been completed. For an operation with both arguments and returned value, besides the status variable, the sequential specification also includes the arguments and returned value. Based on $DetectableT$, the definition of detectability is provided. To show how to design a detectable data structure according to the proposed sequential specification of $DetectableT$, I develop a detectable algorithm called Detectable Queue with Queue (one of the most fundamental data types) as the base type for $DetectableT$. In Detectable Queue, I use private variables in persistent memory, which can only be accessed by their owner thread, as auxiliary variables to store the bookkeeping data of each operation for every single thread.

Detectable Queue is based on Michael and Scott's lock-free queue (MS queue) [28], which uses compare-and-swap (CAS) for synchronization among threads. One difficulty of designing Detectable Queue is that the result of a CAS operation is in the volatile CPU registers and if a crash happens, the result will be lost. It is not easy to know whether a CAS operation before a crash has been successful or not by simply re-reading the variable which CAS was applied to because the variable may have already been overwritten by another thread before or after the crash. To help detectable data structures handle this common problem, I provide a synchronization primitive called CASWithEffect which executes a CAS operation and stores the effect of this CAS operation into an auxiliary private variable in persistent memory atomically. The private variable can be inspected during recovery to get the effect of its owner thread's last CAS operation. With the help of CASWithEffect, I develop another detectable queue data structure called CASWithEffect Queue, which provides the same detectability as Detectable Queue but with a much simpler design.

Regarding the correctness properties for persistent data structures, a variety of definitions have been proposed [4, 7, 24, 31]. The correctness property I consider for Detectable Queue and CASWithEffect is strict linearizability, proposed by Aguilera and Frølund [1], which requires that an operation that is interrupted by a crash either takes effect before the crash or not at all.

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 introduces the system model and presents some terminologies used in this thesis. Chapter 4 describes a new data type *DetectableT* and provides its sequential specification. The Detectable Queue algorithm and CASWithEffect algorithm are provided in Chapter 5 and 6 respectively. Chapter 7 describes the memory management mechanism used in this thesis. Chapter 8 provides the proof of correctness for the Detectable Queue algorithm. The experimental evaluation for the two algorithms with Intel Optane persistent memory is presented in Chapter 9. Chapter 10 concludes this thesis.

Chapter 2

Related Work

There have been a number of works focusing on designing concurrent persistent data structures. Several previous approaches incorporate transaction memory, relying on redo and undo logging to keep track of the process's status before crash [10, 35]. Brown and Avni [9] proposed a persistent hybrid transnational memory, providing a solution combining both software and hardware transnational memory for persistent memory. These approaches usually provide general and convenient interfaces, however their bookkeeping information in logs causes great overhead and undermines performance. To mitigate this problem, some works proposed optimised log-based algorithms [23, 26]. Izraelevitz et al. [23] presented JUSTDO logging, which only records its most recent store instruction as a minimal log and resumes the operation after failure without rolling back. Other works aim at designing recoverable and concurrent data structures that avoid logging entirely [12, 13, 34]. Friedman et al. [13] proposed three novel implementations of a persistent and lock-free queue. David et al. [12] illustrated how to design log-free concurrent data structures and presented several lock-free algorithms including a persistent linked list and hash table. Some papers tried to improve performance by reducing the overhead of cache write-back instructions. Nawab et al. [30] proposed a persistent hash map that periodically writes data back to persistent memory. The work of [13] presented a relaxed persistent queue with a sync operation to persist a batch of data. Cohen et al. [11] presented a universal construction which uses at most one write-back instruction per one update operation. Nawab et al. [29] introduced timely sufficient persistence, which persists a minimal amount of data just in time to maintain a consistent status. Some of these persistent data structures [4–6, 13] are said to be detectable [13], which makes it possible to retrieve the effect of the last operation being executed before a crash. The work of [13] presented a detectable queue data structure using logs as auxiliary variables to store bookkeeping information. Both Attiya et al. [4] and Ben-David et al. [6] proposed recoverable and detectable CAS operations with unique identifiers that represent each operation being stored. Detectable data structures always need additional space in persistent memory to store auxiliary bookkeeping data. Ben-Baruch et al. [5] investigated the upper and lower bounds on the space complexity of detectable objects and presented a bounded-space detectable CAS algorithm. There is also some research working on persistent synchronization primitives. Wang et al. [36] implemented Persistent Multi-Word Compare-And-Swap (PMwCAS) by adding persistency and recovery on top of Harris et al.'s MwCAS [19].

Recent work of persistent memory proposed several definitions of correctness conditions for persistent concurrent objects with concerns of durability [4, 17, 31]. Aguilera and Frølund [1] proposed strict linearizability, which preserves locality (the strict linearizabe histories of two objects merged together is also strict linearizabe) and the program order, but may forbid some wait-free implementations. Berryhill et al. [7] proposed recoverable linearizability which permits a wait-free implementation and maintains locality, but may lose the program order in some situations. Izraelevitz et al. [24] proposed durable linearizability, which is a special case of recoverable linearizability for system-wide failures. Durable linearizability guarantees that all complete operations before a crash will have their effects visible while the in-flight operations may or may not complete after the crash. To reduce the overhead caused by persistency, they also proposed a weaker correctness condition: buffered durable linearizability, which only keeps a linearizable subhistory including part of the events before a crash.

Attiya et al. [4] proposed nesting-safe recoverable linearizability (NRL) and defined detectability through NRL. Their model requires the following system assumptions: "a recovery function *Op.Recover* is invoked by the system to recover from a crash" and "*Op.Recover* has access to a designated per-process non-volatile variable LI_p, identifying the instruction of *Op* that p was about to execute" [4]. Compared to NRL, the definition of detectability in this thesis through data type *DetectableT* does not require system to invoke recovery. The recovery for *DetectableT* is determined by the application. Besides, no instruction needs to be stored in persistent memory for *DetectableT*. Without these strong assumptions, *DetectableT* can be implemented on current hardware system as shown in Chapter 9.

Other works focused on the study of recoverable mutual exclusion (RME) problems introduced by Golab and Ramaraju [16], which allows process to regain its previously owned mutex during recovering after a crash. Further studies [14, 25] presented different RME algorithms with different time complexity and fault tolerance properties. Ramaraju introduced the Swap-And-Store primitive, which stores the result of a swap operation, in his recoverable mutual exclusion thesis [32]. In Golab and Hendler [15]'s paper, they proposed RME models under system-wide failures instead of one process failure.

Chapter 3

Preliminaries

In this chapter, I will present the system model, some terminologies and algorithms used in my persistent data type design and correctness analysis.

3.1 CPU Support for Persistent Memory

I assume a store model where byte-addressable persistent memory is attached directly to the memory bus. While the main memory is persistent, CPU caches and registers are volatile, i.e., their contents are lost after a failure such as a power outage. One difficulty for volatile caches is that there is no guarantee on when or in what order the data in the caches are written back to the persistent memory. The cache line write-back can actually happen at any time. A cache line flush instruction is used to explicitly flush data to the persistent memory. One such instruction on Intel systems is CLFLUSH, which receives a memory address and flushes the entire cache line containing that address to the memory. When CLFLUSH flushes a cache line, it also invalidates the cache line. There is a new flushing instruction – CLWB, which may not invalidate the cache line when writing back it. I also need to make sure the cache line write-backs are not reordered. SFENCE, a memory fence instruction, ensures the global visibility of every store instruction executed before in program order. SFENCE is used with flushing instructions to guarantee durability and ordering. In the algorithms of this thesis, I use a flush instruction, which receives a memory address, to represent both a cache line flush instruction and memory fence instruction. Another difficulty for volatile CPU registers is that the response of an operation is stored in registers which will be lost during a failure. I will discuss this problem in Chapter 4 with details.

3.2 Histories

The model used in this thesis is closely based on Herlihy and Wing's [22]. I consider a system consisting of n asynchronous processes, denoted p_1, ..., p_n, which communicate through shared objects using memory access primitives. Each process has access to the persistent memory with atomic memory instructions including read, write and read-modify-write primitives. The state of a process, including the program counter and stack pointer, is stored in the processor's registers. As in previous work [7], I consider only one type of failure, a system crash which can happen at any point and may recover later. Upon such a crash, the contents of persistent memory are preserved while the contents of CPU caches and registers are lost. All processes running when a system crash happens lose their states. After the system recovers, a process that crashed before may resume through a recovery routine.

Every shared object has a unique identity and belongs to a data type, which provides a set of operations to define its behavior. A process, after invoking an operation on an object, must wait for the response of that operation before invoking another operation if there is no crash happening. Multiple processes can invoke operations on the same shared object concurrently.

The execution of an operation is modeled by two events: an *invocation event* and a *response event*. The invocation event, denoted $\langle \text{INV}, p_i, X, op(a) \rangle$, represents that process p_i invokes an operation *op* on an object X with a sequence of arguments a. The response event, denoted $\langle \text{RES}, p_i, X, r \rangle$, represents that process p_i receives the response of its last invoked operation on an object X with a sequence of returned values r.[1] For atomic operations, I consider the invocation and response happening at the same time, which can be treated as one event which includes both the invocation and response. In Section 3.1, it mentions that the cache line write-back can actually happens at any time even with out executing a flush instruction. To represent cache line write-back in a history, I make a simplifying assumption that cache line write-back only happens when a flush instruction is executed and define a flush event in a history to indicate that a flush instruction has been executed. There is also a system crash event, denoted $\langle Crash \rangle$, which represents that a system crash occurs. I make an assumption that crash represents both a system failure and the subsequent execution of a single process recovery procedure, which will be explained later in this thesis. The recovery and crash are considered atomic for simplicity. As in [13, 22], an execution of a concurrent system is modeled by a *history*, which is a

[1]The response event is distinct from the return statement of an operation, and occurs after the operation has returned.

finite sequence of invocation events, response events and crash events. A *subhistory* of a history H is a subsequence of the events in the history H. For an invocation event I in a history H, the first following response event R, which is performed by the same process P on the same object before P's next invocation (if any on that object), is I's *matching* response. An invocation which does not have a matching response is *pending* in H. A history H is *sequential* if: (1) each response event is immediately preceded by an invocation event which the response matches; (2) each invocation event, except possibly the last one, is immediately followed by its matching response. A sequential history only includes operation events and excludes crash events.

For a process p, the process subhistory of a history H, denoted $H|p$, is the subsequence of events in H which are performed by p. Similarly, for an object X, the object subhistory of H, denoted $H|X$, is the subsequence of events in H which happen on X. Since crash events apply to all processes in the system, $H|p$ and $H|X$ retain all crash events. Two histories H and H' are equivalent if for every process p, $H|p = H'|p$. A history H is *well-formed* if for every process p: (1) each response event in $H|p$ is immediately preceded by an invocation event which the response matches; (2) each invocation event in $H|p$, except possibly the last one, is immediately followed by its matching response or a crash event. Unlike a sequential history, a well-formed history can contain crash events. All histories in this thesis are assumed to be well-formed.

An *operation* O in a history H is a pair consisting of an invocation event I and I's matching response R, if R exists. If R does not exist, O only includes I. An operation is *complete* if the matching response exists and *pending* if there is no matching response. A complete operation with invocation and response events is denoted $[X, op(a)/res(r), p_i]$, where X is an object and p_i is a process. The object and process can be omitted if they are clear from the context. Let op_0 and op_1 be two complete operations in H. Operation op_0 *happens before* op_1 if the matching response event of op_0 precedes the invocation event of op_1 in H. This *happens-before* relation defines an irreflexive partial order, denoted $<_H$, on the operations of H. If neither $op_0 <_H op_1$ nor $op_1 <_H op_0$ holds, op_1 and op_0 are *concurrent* in H. If H is a sequential history, $<_H$ becomes a total order.

Each object has a type that defines the object's abstract state and the effect of different operations on this state. A type can be formalized using a *sequential specification*, as explained in more detail in Section 3.5.1. A sequential history H is *legal* if for every object X of some type T, the operations and responses in $H|X$ are consistent with a sequence of state transitions allowed by the sequential specification of T.

3.3 Progress Properties

A shared object implementation is *recoverable lock-free* if it guarantees infinitely often some operations complete in a finite number of steps unless there are infinitely many failures. The *recoverable lock-free* property ensures that if any process is taking steps, some operation will make progress or a failure happens.

3.4 Correctness Properties

Linearizability is a widely adopted correct condition for shared objects in concurrent systems. It can only be used for histories without crash events. For a given history H, its completion H' is defined as: first extending H by appending matching responses for a subset of pending invocations in the end; then removing remaining pending invocations.

Definition 3.1 (Linearizability [22])**.** A well-formed history H without crash events is linearizable if there exist a completion H' of H and a legal sequential history S, such that:

L1. H' is equivalent to S, and

L2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops. appear in S then $op_1 <_S op_2$).

Informally, the completion H' shows that a pending invocation may take effect or not before its response returns. Both situations need to be considered. Linearizability requires that each complete operation takes effect at some point of time, called the operation's linearization point, between the invocation and response [18]. Besides, the happens-before order of operations in H must be retained in S according to L2. A correctness property P is local if a history H satisfies P if and only if, for every object X accessed in H, $H|X$ satisfies P. Linearizability is a local property.

For histories with crash events, strict linearizability is one correctness condition. For a given history H, its strict completion H' is defined as: first adding matching responses for a subset of pending operations before the next crash event (if any) or in the end if no such crash event exists; then removing remaining pending invocation and crash events.

Definition 3.2 (Strict Linearizability [1])**.** A well-formed history H is strictly linearizable if there exist a strict completion H' of H and a legal sequential history S, such that:

SL1. H' is equivalent to S, and

SL2. $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$ and both ops. appear in S then $op_1 <_S op_2$).

Informally, strict linearizability requires a pending operation to take effect before a crash or, not at all. No pending operation can take effect after a crash. Strict linearizability is a local property. However, it precludes some wait-free data structures [7, 24].

Persistent atomicity is another correctness property, which allows a pending operation to take effect after a crash. This is particularly relevant to some implementations in which an operation invoked by process p_0 can be completed by process p_1 after p_0 fails through helping mechanisms. For a given history H, its persistent completion H' is defined as: first adding matching responses for a subset of pending operations before the same process invokes another operation (if any) or in the end if no following operation is invoked by the same process; then removing remaining pending invocation and crash events.

Definition 3.3 (Persistent Atomicity [17])**.** A well-formed history H is persistently atomic if there exist a persistent completion H' of H and a legal sequential history S, such that:

PL1. H' is equivalent to S, and

PL2. $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$ and both ops. appear in S then $op_1 <_S op_2$).

It has been proved that persistent atomicity is not a local property [7]. An alternative property, called recoverable linearizability, is proposed which is a local property. Recoverable linearizability deals with happens-before relation differently from strict linearizability and persistent atomicity. Given a history H, suppose there are two operations op_0 and op_1 invoked by the same process on the same object in H. The operation op_0 is invoked before op_1 if the invocation event of op_0 precedes the invocation event of op_1. This invoked-before relation defines an irreflexive partial order, denoted $\ll_H$, on the operations of H.

Definition 3.4 (Recoverable Linearizability [7])**.** A well-formed history H is recoverable-linearizable if there exist a strict completion H' of H and a legal sequential history S, such that:

RL1. For every object X in H', $H'|X$ is equivalent to $S|X$;

RL2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops. appear in S then $op_1 <_S op_2$); and

RL2. $\ll_H \subseteq <_S$ (i.e., if $op_1 \ll_H op_2$ and both ops. appear in S then $op_1 <_S op_2$).

Informally, recoverable linearizability allows a pending operation, which is invoked by process P on object X, to take effect before the linearization point of another operation invoked by P on X after a crash.

Both persistent atomicity and recoverable linearizability are based on the failure model which assumes a single process can recover or reincarnate after a crash event. Durable linearizability only considers a full-system failure mode, in which processes fail together as a part of a full-system crash and are replaced by new processes when the system recovers. For a given history H, I define its durable completion H' as: first adding matching responses for a subset of pending operations in the end; then removing remaining pending invocation and crash events.

Definition 3.5 (Durable Linearizability [24])**.** A well-formed history H is durably linearizable if there exists a durable completion H' of H, such that: H' is linearizable.

Informally, durable linearizability allows a pending operation in a history to take effect anytime after its invocation. It is also a local property.

3.5 Specifying Shared Objects

In this section, I provide the definition of data type, and present some shared objects used in the thesis.

3.5.1 Definition of Data Type

An object has a data type, which defines a set of possible values to reflect the object's state and provides a set of operations as the only means to manipulate the state. I use the same axiomatic style as in [22] to formalize an object's sequential specification. In this style, a sequential specification is described using a state transition relation which shows the object's states before invoking an operation and after receiving the invocation's response.

A data type T is defined by a sequential specification [22] which consists of

- a set S representing all possible states for an object X of type T,
- the initial state $s_0 \in S$,
- a set OP of operations which T provides to manipulate X's state,
- a set R of possible values X can return for OP,
- a state transition function $\delta : S \times OP \to S$ and
- a response function $\rho : S \times OP \to R$.

The sequential specification defines the allowed behaviour in a sequential history for an object X when an operation op of T is invoked on X. If $\delta(s, op) = s'$, it means that when an operation op is invoked on an object in state s, the object moves to state s'. If $\rho(s, op) = r$, it means that when an operation op is invoked on an object in state s, r (a specific value or a void response) is returned to the process which invokes op.

In the axiomatic style, a pre-condition is used to describe an object's state before an operation being invoked, and a post-condition to describe, after the operation is complete, the object's state and returned values. If an operation op with an argument list a is executed by process p on an object X, $[X, op(a)/res(r), p]$ is used to denote it as explained in Section 3.2. If the process p and object X are clear from the context, they can be omitted. The sequential specification of a FIFO (first-in-first-out) Queue [22] is shown in Figure 3.1. In Axiom 3.1, the pre-condition *true* means that *enqueue* can be invoked in every state of the queue object. The post-condition shows that after the enqueue operation is complete, the new queue value is the old queue value with the argument e being appended. The pre-condition of Axiom 3.2 shows that this state transition for *dequeue* only applies when the queue value is not empty. The post-condition indicates that *dequeue* removes the first item of the queue and returns the first item. Axiom 3.3 demonstrates that when the queue value is empty, *dequeue* returns a *NULL* value without changing the state.

$$\begin{gathered}\{true\}\\ [enqueue(e)/res()]\\ \{q' = append(q, e)\}\end{gathered} \tag{Axiom 3.1}$$

$$\begin{gathered}\{q \neq []\}\\ [dequeue()/res(e)]\\ \{q' = rest(q) \wedge e = first(q)\}\end{gathered} \tag{Axiom 3.2}$$

$$\begin{gathered}\{q = []\}\\ [dequeue()/res(e)]\\ \{e = NULL\}\end{gathered} \tag{Axiom 3.3}$$

Figure 3.1: Axioms for queue operations

3.5.2 Verifying that Implementations are Strictly Linearizable

From Herlihy and Wing's model [22], an abstract type ABS is implemented by a representation type REP. An operation in the abstract type (abstract operation) is implemented by a sequence of operations in the representation type (representation operations) that takes place within the abstract operation. Therefore, a history H of an implementation contains events of both ABS objects and REP objects. For abstract operations, each operation has two event: invocation and response (Section 3.2). For representation operations, I consider each operation as an atomic operation with the invocation and response happening at the same time. Therefore, there is only one event happening for each representation operation. The subhistory of ABS evens in H is denoted $H|ABS$ and the subhistory of REP events in H is denoted $H|REP$. For a given history H, the value of an object in H after the end of one strict linearization of H is called a strict linearized value of H. Since H of an ABS object can have more than one strict linearization, $SLin(H|ABS)$ denotes the set of all strict linearized values of H for an ABS object. For an REP object, assuming REP is deterministic, since the operations of REP are considered atomic, there is only one strict linearized value of $H|REP$, denoted $SV(H|REP)$. The allowed values of REP objects are characterized by a predicate I, called the *representation invariant*. Each representation value is mapped to a nonempty set of abstract values by an abstraction function A. According to [22], the technique of verifying that REP implements ABS focuses on proving that for every history H of the implementation, the following statement $s(H)$ holds.

$$\textit{Letting } r \ = SV(H|REP),\ I(r) \textit{ holds and } A(r) \subseteq SLin(H|ABS)$$

Figure 3.2: Statement $s(H)$.

3.5.3 Read-Modify-Write Primitives

Atomic read-modify-write primitives are widely used in concurrent lock-free data structures. One of these primitives is CAS (compare-and-swap), which takes three arguments (a target shared address, an old value, a new value), and returns the current value of the target address [21]. Figure 3.3 shows the procedure of CAS.

CAS puts the new value to the target address if the old value matches the current value of that address atomically. Based on CAS, there are two other primitives: DCAS (Double-Word-Compare-And-Swap) which modifies two words atomically, and MwCAS (multi-word compare-and-swap) which modifies arbitrary words atomically. The return value of DCAS or MwCAS is boolean, indicating whether it updates the addresses. The procedure of MwCAS is shown in Figure 3.4, in which MwCAS updates a list of target addresses to the new values if the old value of every target address equals to the corresponding old value

```
uint64_t CAS(uint64_t * addr, uint64_t o, uint64_t nVal) {
    Atomically {
        uint64_t curV = (*addr);
        if(o == curV) {
            *addr = nVal;
        }
        return curV;
    }}
```

Figure 3.3: CAS Operation

atomically. CAS is provided by contemporary multiprocessors in hardware, while DCAS and MwCAS can be built using CAS through software mechanisms such as [2, 19, 20].

One common problem of using CAS (conditional synchronization operation) with a dynamic memory algorithm is ABA problem [21]. Suppose a memory node A, which holds data of 1, is stored in an address X. Process p_1 reads A from X and gets A's data. Then p_1 intends to replace 1 with 2 using another memory node C through CAS. Before p_1 executes CAS, other processes p_2 and p_3 races ahead of p_1 for some time. During this time, p_2 changes the value of X to B, and A, as a memory node, is recycled. Then p_3 gets A from the memory pool, sets its data to 3 and puts A back to X. After that, p_1 executes CAS successfully on X, which causes a problem. Process p_1 is supposed to replace data 1 with 2, but ends with replacing 3 with 2. To handle this problem, a lock-free memory reclamation approach is always needed.

```
bool MwCAS(uint64_t ** addrList, uint64_t * oList, uint64_t * nList,
    uint8_t n) {
    Atomically {
        for(uint8_t i = 0; i < n; i++) {
            if(*(addrList[i]) != oList[i]) return false;}
        for(uint8_t i = 0; i < n; i++) {
            *(addrList[i]) = nList[i];}
        return true;
    }}
```

Figure 3.4: MwCAS Operation

Another important synchronization primitive is FAS (fetch-and-store), which takes two arguments: a target shared address and a new value as shown in Figure 3.5. FAS puts the new value to the target address and returns the current value of the target address. Unlike CAS, FAS always updates the shared variable. FAS is also provided by modern hardware.

```
uint64_t FAS(uint64_t * addr, uint64_t n) {
    Atomically {
        uint64_t curVal = *addr;
        *addr = n;
        return curVa;
    }}
```

Figure 3.5: FAS Operation

3.6 Background Algorithms

3.6.1 Michael and Scott's Queue

In this thesis, I present several persistent lock-free queue algorithms based on Michael and Scott's lock-free queue (MS queue) [28], which uses CAS for synchronization. MS queue includes a linked list of nodes with two queue pointers: head and tail pointing to the queue's first and last node respectively as shown in Figure 3.6. The head pointer always points to a sentinel node, which is considered as a place-holder instead of a queue node with a meaningful value. When an MS queue is initiated as an empty queue, both the head pointer and tail pointer points to a sentinel node.

```
class QueueNode {
    uint64_t * pData_;
    QueueNode * next_ = NULL;}

class MSQueue {
    QueueNode * phead_;
    QueueNode * ptail_;
    void enqueue(QueueNode * enqNode);
    QueueNode * dequeue();}
```

Figure 3.6: MS Queue classes

```
32 void MSQueue::enqueue(QueueNode * node) {
33     while(true) {
34         QueueNode * last = ptail_;
35         QueueNode * next = last->next_;
36         if(last == (ptail_)) {
37             if(next == NULL) {
38                 if(CAS(&last->next_, NULL, node) == NULL){
39                       CAS(&ptail_, last, node);
40                       return;}}
41             else {
42                 CAS(&ptail_, last, next);}}}}
```

Figure 3.7: Enqueue operation of MS Queue

Figure 3.7 shows the enqueue operation, which receives a new queue node and reads the last node of the queue from the tail pointer. Then, the operation checks whether the next pointer of the last node is NULL, and tries to append the new node to the last node by using CAS if it is (line 38). If the CAS succeeds, it calls another CAS to update the tail pointer to the new node and returns. If appending fails, the operation retries by going back to reading the current value of the tail pointer. If the next pointer of the last node is not NULL, which means another node has already been appended to the last node, the operation helps to update the tail pointer to the new last node (line 42).

```
43 QueueNode * MSQueue::dequeue() {
44     while(true) {
45         QueueNode * first = phead_;
46         QueueNode * next = first->next_;
47         QueueNode * last = ptail_;
48         if(first == phead_) {
49             if(first == last) {
50                 if(next == NULL) {
51                     return NULL;}
52                 CAS(&ptail_, last, next);}
53             else {
54                 if(CAS(&phead_, first, next) == first) {
55                     return next;}}}}}
```

Figure 3.8: Dequeue operation of MS Queue

The pseudo code of the dequeue operation is shown in Figure 3.8. The dequeue operation reads the first node from the head pointer, the next node from the next pointer of the first node, and the last node from the tail pointer (line 45-47). If the first node is equal to the last node and the next node is NULL, which means there is only a sentinel node in the queue, the operation returns a NULL pointer (line 51). If the first node equals to the last node and the next node is not NULL, which means the next node has just been appended to the queue, the operation helps to update the tail pointer to the next node (line 52). Otherwise, the operation tries to update the head pointer to the next node by using CAS (line 54). If the CAS succeeds, it returns the next node because the next node is removed from the queue and becomes the current sentinel node. If CAS fails, the operation will retry until it successfully removes a node or the queue becomes empty.

The memory management for queue nodes will be explained in Chapter 7.

Chapter 4

Detectability

One problem for persistent data structures is how to detect the effect of the last invoked operation before a crash since the response of an operation is stored in volatile CPU registers. Take a persistent lock-free queue structure as an example. If a thread is executing a CAS operation to add a queue node to the tail of the queue before a crash, then after restarting, the thread should be able to know the result of its previous enqueue operation. If the enqueue operation did not take effect, the thread may retry it with the same queue node. In this way, the crash does not become an obstacle to the thread's normal execution path. A data structure is considered detectable if it is possible for a thread to determine the effect of its last invoked operation before a crash [13].

Friedman et al. presented two algorithms for durable concurrent queues, known as Durable Queue and Log Queue, in [13]. While both of them satisfy durable linearizability [24], only Log Queue is noted as a detectable implementation, which provides a mechanism to get the effects of the last operation being executed before a crash. The difference between the two queues is that after a crash, Log Queue provides the ability for a thread to tell whether its previous operation was completed. The thread can finish it if it was not. Compared to Durable Queue, Log Queue generates a log for each thread's operation to store some additional recovery information. In the enqueue operation, before trying to link a new node to the tail of the queue, Log Queue first stores a pointer to the new node into the thread's enqueue log. Then during recovery, if the stored node appears in the queue, the thread discovers that the new node has already been appended to the queue. In the dequeue operation, Log Queue saves the response (a dequeued node) of the operation into the thread's dequeue log so that after a crash, the thread can retry the dequeue operation if there is no dequeued node in the log. In general, while the recovery of Durable Queue ensures the consistency of the persistent queue, the recovery of Log Queue provides an additional ability for threads to complete unfinished operations after a crash.

After analyzing the above two queue algorithms, I find that the detectable implementation requests a certain information to be stored so that recovery can use this information to get the status of the last invoked operation. This feature can be generalized as an abstract data type and composed with other data types (e.g., Queue) to make them detectable. In [4], Attiya et al. describe detectable data structures through a correctness property (nesting-safe recoverable linearizability), which is different from my approach. In the next section, will provide the details of this abstract data type.

4.1 ***DetectableT*** Data Type

I consider detectability as an additional ability for recovery of persistent data types. To build detectable data structures, I design an abstract data type *DetectableT*. *DetectableT* must be composed with another data type, which I call a *base type*. A base type always provides several operations, called *base operations*, to read and update its state. For example, if *DetectableT* is integrated with *queue* type, the base type is *queue*, and base operations are *dequeue* and *enqueue*. If some additional bookkeeping of base operations is stored, recovery can determine a base operation's state based on the bookkeeping data. For an operation executed by a process, there exists an executing status to show the progress of execution, and possibly some arguments and returned values. To represent this execution, *DetectableT* uses a status value to denote the executing status, an argument list to store the arguments and a returned value list to store the returned values. This information, including the executing status, returned values and arguments, can be returned when getting the effect of an operation.

DetectableT adds three functions for a base operation *op* to become detectable: a *prepareOp* function, *detectableOp* function and *retrieveOp* function. I call these operations, which help to achieve detectability for a base operation, *auxiliary operations*. For example, for a *dequeue* operation, *DetectableT* adds a *prepareDequeue* function, *detectableDequeue* function and *retrieveDequeue* function. Process p_i calls the *prepareOp* function of a base operation *op* on object X to indicate its intention of invoking *op*. The *prepareOp* function also stores the arguments of *op* if any. Then in the *detectableOp* function, p_i executes *op* using the stored arguments with the executing status being updated and returned values being saved. The *detectableOp* function has the same returned values as *op*. If a crash happens when *detectableOp* is being executed, during recovery, the state of X has to be recovered by some roll-back or roll-forward procedures to guarantee consistency. A pending operation which is rolled-forward during recovery must have all its bookkeeping data stored and updated in the same way as a complete operation; a pending operation which is rolled-back must revert any changes it made. After that, p_i calls the *retrieveOp* function to obtain *op*'s effect which includes the executing status, argument list and returned value, if any. Recovery implicitly happens before the *retrieveOp* function being called. The *detectableOp* function takes effect either before *regrieveOp* or not at all. Since detectability, as an additional ability, is a plug-in function, *DetectableT* maintains the ability to execute base operations without injecting detectability by not storing bookkeeping data.

When getting the effect of an operation of *DetectableT* objects, the effect consists of two parts: the first part indicating whether the operation takes effect and the second part storing the returned values if the operation completes. For an operation with arguments, the argument list is also returned so that it is possible for an operation to be re-executed with the same arguments when not completed before a crash.

4.2 ***WDetectableT*** Data Type

In some situations, there is no need to return as much information as *DetectableT* requires when getting the effect. Some algorithms do not ask for the argument list. For example, if a process always passes a fixed value to an operation, the process will not ask for the argument value after a crash. Also, some algorithms do not need to know an operation has been completed if the effect of this operation does not update its object's state. Take a dequeue operation as an example. When a dequeue operation returns a NULL pointer without changing a queue's state, a process, which tries to get a queue node from the queue, will retry the dequeue operation until it gets one. After a crash, it is unnecessary to notify the process that its previous dequeue operation completes and returns a NULL pointer. For these algorithms, *DetectableT* returns more information than needed, which may cause more overhead during the implementation. Therefore, I design another data type *WDetectableT* (weak *DetectableT*) to not return unnecessary information when getting the effect for the above two situations. In the next section, I provide the sequential specification for both *DetectableT* and *WDetectableT* objects.

4.3 Sequential Specification of *DetectableT* and *WDetectableT* Objects

A base type B, which is composed to *DetectableT* or *WDetectableT*, is denoted $D\{B\}$ or $D'\{B\}$ respectively. Every operation in B is made detectable in $D\{B\}$ and $D'\{B\}$. For a base operation *op* of B to become detectable, three auxiliary operations (*prepareOp*, *detectableOp* and *retrieveOp*) must be added in $D\{B\}$ or $D'\{B\}$. Since *op* is executed within *detectableOp*, the state transition of the base operation *op* is part of the state transition of *detectableOp*. I use s and s' to denote the state before *op* being invoked and after *op* being completed, respectively. The state transition function and response function of *op* are denoted $\delta(s, op)$ and $\rho(s, op)$ respectively, where s represents the before state, *op* represents the operation. The bookkeeping data of *op* includes a status value, an argument list and returned value list, which are denoted $S_p[op]$, $A_p[op]$ and $R_p[op]$, respectively. The initial value of every bookkeeping data item is $\perp$.

A base operation does not necessarily have arguments or returned values. Regarding whether the argument list or returned value list exists, I divide base operations into four operation types as shown in table 4.1. When getting the effect of operations of different operation types, different information is provided. For example, returned values can only be provided for an operation which has returned values. Therefore, I provide a separate sequential specification for each operation type.

Operation Type	Arguments	Returned Values
NARG_NRET_T	N	N
HARG_NRET_T	Y	N
NARG_HRET_T	N	Y
HARG_HRET_T	Y	Y

Table 4.1: Operation types categorized according to whether the argument list or returned value list exists.

The first operation type, denoted *NARG_NRET_T*, indicates a base operation with no arguments or returned values. One example of *NARG_NRET_T* is the clear operation of a list, which deletes all the entries. The sequential specification of $D\{B\}$ objects for this operation type is shown in Figure 4.1. The bookkeeping data only includes the status value $S_p[op]$ to store the executing status. Axiom 4.6 shows *op* is executed without detectability. To execute a detectable operation, the *prepareOp* function (Axiom 4.1) must be called first to set $S_p[op]$ to *BEGIN_DECT_OP*. Then in Axiom 4.2, $S_p[op]$ is set to *END_DECT_OP* when *op* is completed. Axiom 4.2 also applies to a pending operation which is completed during recovery for this operation. The effect is decided on the value of $S_p[op]$. With *BEGIN_DECT_OP* status, *retrieveOp* returns *NoEffect* (i.e., *op* takes no effect), while with *END_DECT_OP* status, *retrieveOp* returns *HasEffect* (i.e., *op* takes effect) and *AckResponse* (i.e., a void return value), as shown in Axiom 4.3 and Axiom 4.4. It is possible for a process to call *retrieveOp* when there is no *op* invoked before a crash. In this situation, *NoExecutedOp* is returned as shown in Axiom 4.5. Since there is no argument or returned value for operation type *NARG_NRET_T*, the sequential specification of $D'\{B\}$ objects is the same as $D\{B\}$.

$$\begin{gathered}\{true\}\\ [prepareOp()/res(), p]\\ \{S'_p[op] = BEGIN\_DECT\_OP\}\end{gathered} \tag{Axiom 4.1}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [detectableOp()/res(), p]\\ \{s' = \delta(s, op) \wedge S'_p[op] = END\_DECT\_OP\}\end{gathered} \tag{Axiom 4.2}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [retrieveOp()/res(NoEffect, \bot), p]\\ \{\}\end{gathered} \tag{Axiom 4.3}$$

$$\begin{gathered}\{S_p[op] = END\_DECT\_OP\}\\ [retrieveOp()/res(HasEffect, AckResponse), p]\\ \{\}\end{gathered} \tag{Axiom 4.4}$$

$$\begin{gathered}\{S_p[op] = \bot\}\\ [retrieveOp()/res(NoExecutedOp, \bot), p]\\ \{\}\end{gathered} \tag{Axiom 4.5}$$

$$\begin{gathered}\{true\}\\ [op()/res(), p]\\ \{s' = \delta(s, op)\}\end{gathered} \tag{Axiom 4.6}$$

Figure 4.1: The sequential specification for $D\{B\}$ and $D'\{B\}$ objects regarding operation type *NARG_NRET_T*.

The second operation type, denoted *HARG_NRET_T*, indicates a base operation receiving an argument list with no returned values, such as an enqueue operation. The sequential specification of $D\{B\}$ for this operation type, shown in Figure 4.2, is based on the sequential specification for operation type $NARG\_NRET\_T$ with a value $A_p[op]$ being added to record the argument list. Axiom 4.12 shows op is executed without detectability as in the base type. For a detectable operation, in Axiom 4.7, besides setting $S_p[op]$ to $BEGIN\_DECT\_OP$, the argument list of op is passed to $A_p[op]$, such as a new queue node to be enqueued. Then in Axiom 4.8, op is executed with the argument value in $A_p[op]$ and $S_p[op]$ is updated to END_DECT_OP. When $retrieveOp$ is called, besides the effect of op, the argument is also returned so that a re-execution with the same argument is possible for an operation that did not take effect.

$$\begin{gathered}\{true\}\\ [prepareOp(a)/res(), p]\\ \{A'_p[op] = a \land S'_p[op] = BEGIN\_DECT\_OP\}\end{gathered} \tag{Axiom 4.7}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [detectableOp()/res(), p]\\ \{s' = \delta(s, op(A_p[op])) \land S'_p[op] = END\_DECT\_OP\}\end{gathered} \tag{Axiom 4.8}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [retrieveOp()/res(NoEffect, \bot, A_p[op]), p]\\ \{\}\end{gathered} \tag{Axiom 4.9}$$

$$\begin{gathered}\{S_p[op] = END\_DECT\_OP\}\\ [retrieveOp()/res(HasEffect, AckResponse, A_p[op]), p]\\ \{\}\end{gathered} \tag{Axiom 4.10}$$

$$\begin{gathered}\{S_p[op] = \bot\}\\ [retrieveOp()/res(NoExecutedOp, \bot, \bot), p]\\ \{\}\end{gathered} \tag{Axiom 4.11}$$

$$\begin{gathered}\{\delta_p[op(a)]\}\\ [op(a)/res(), p]\\ \{s' = \delta(s, op(a)\}\end{gathered} \tag{Axiom 4.12}$$

Figure 4.2: The sequential specification for $D\{B\}$ objects regarding operation type $HARG\_NRET\_T$.

The sequential specification of $D'\{B\}$ objects for operation type *HARG_NRET_T* is shown in Figure 4.3. Compared to the specification for $D\{B\}$, the only difference is that in the *retrieveOp* function, the argument value is not returned.

$$\begin{gathered}\{true\}\\ [prepareOp(a)/res(), p]\\ \{A'_p[op] = a \wedge S'_p[op] = BEGIN\_DECT\_OP\}\end{gathered} \tag{Axiom 4.13}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [detectableOp()/res(), p]\\ \{s' = \delta(s, op(A_p[op])) \wedge S'_p[op] = END\_DECT\_OP\}\end{gathered} \tag{Axiom 4.14}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [retrieveOp()/res(NoEffect, \perp), p]\\ \{\}\end{gathered} \tag{Axiom 4.15}$$

$$\begin{gathered}\{S_p[op] = END\_DECT\_OP\}\\ [retrieveOp()/res(HasEffect, AckResponse), p]\\ \{\}\end{gathered} \tag{Axiom 4.16}$$

$$\begin{gathered}\{S_p[op] = \perp\}\\ [retrieveOp()/res(NoExecutedOp, \perp), p]\\ \{\}\end{gathered} \tag{Axiom 4.17}$$

$$\begin{gathered}\{\delta_p[op(a)]\}\\ [op(a)/res()\ p]\\ \{s' = \delta(s, op(a)\}\end{gathered} \tag{Axiom 4.18}$$

Figure 4.3: The sequential specification for $D'\{B\}$ objects regarding operation type *HARG_NRET_T*.

The third operation type, denoted *NARG_HRET_T*, indicates a base operation receiving no argument and returning a list of values, such as a dequeue operation. The sequential specification of $D\{B\}$ objects for this operation type is based on the sequential specification of $D\{B\}$ for *NARG_NRET_T* with a value $R_p[op]$ being added to record the returned value list as shown in Figure 4.4. Axiom 4.24 shows the original way of executing *op* with a returned value list r. The specification of *prepareOp* in Axiom 4.19 is the same as Axiom 4.1 with $S_p[op]$ being updated to *BEGIN_DECT_OP*. When *op* is executed in a detectable way, the returned value list is passed to $R_p[op]$ (Axiom 4.20). In *retrieveOp* (Axiom 4.22), the value of $R_p[op]$ is returned. For an incomplete operation, the specification of *retrieveOp* in Axiom 4.21 is the same as Axiom 4.3, returning *NoEffect*.

$$\begin{gathered}\{true\}\\ [prepareOp()/res(), p]\\ \{S'_p[op] = BEGIN\_DECT\_OP\}\end{gathered} \tag{Axiom 4.19}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [detectableOp()/res(\rho(s, op)), p]\\ \{s' = \delta(s, op) \wedge S'_p[op] = END\_DECT\_OP \wedge R'_p[op] = \rho(s, op)\}\end{gathered} \tag{Axiom 4.20}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [retrieveOp()/res(NoEffect, \perp), p]\\ \{\}\end{gathered} \tag{Axiom 4.21}$$

$$\begin{gathered}\{S_p[op] = END\_DECT\_OP\}\\ [retrieveOp()/res(HasEffect, R_p[op]), p]\\ \{\}\end{gathered} \tag{Axiom 4.22}$$

$$\begin{gathered}\{S_p[op] = \perp\}\\ [retrieveOp()/res(NoExecutedOp, \perp), p]\\ \{\}\end{gathered} \tag{Axiom 4.23}$$

$$\begin{gathered}\{true\}\\ [op()/res(\rho(s, op)), p]\\ \{s' = \delta(s, op)\}\end{gathered} \tag{Axiom 4.24}$$

Figure 4.4: The sequential specification for $D\{B\}$ objects regarding operation type *NARG_HRET_T*.

The sequential specification of $D'\{B\}$ objects for the operation type *NARG_HRET_T* is shown in Figure 4.5. For an operation *op*, the specifications of $D\{B\}$ and $D'\{B\}$ are the same in the *prepareOp*, *retrieveOp* and *op* functions. For the *detectableOp* function, I divide operations into two categories: *opc* and *opo*. The *opc* represents operations which will not update the state of an object in some situations, such as a dequeue operation defined in Axiom 3.2, which cannot get a dequeued node from an empty queue. The *opo* represents the other operations except *opc*, such as an enqueue operation defined in Axiom 3.1. For *opo* (e.g., *enqueue*), the sequential specification of *detectableOpo* (e.g., *detectableEnqueue*) is shown in Axiom 4.28, the same as *detectableOp* in Axiom 4.20. For *opc* (e.g., *dequeue*), according to whether or not the operation can update the object's state, there are two axioms of *detectableOpc* (e.g., *detectableDequeue*). I define $UStatus[opc]$ to denote a state value set in which the *opc* will update the state. For example, $UStatus[opc]$ is a queue set excluding the empty queue for a dequeuing operation. In the pre-condition of Axiom 4.26, the state s is in $UStatus[opc]$, which means the *opc* will update the state. In this condition, the state s' after the operation being completed is not equal to s with $S_p[opc]$ being updated to END_DECT_OP and $R_p[opc]$ being updated to the returned value. In the pre-condition of Axiom 4.27, the state s is not in $UStatus[opc]$, which means the *opc* will not update the the state. In this condition, the state s' after the operation being executed is equal to s with $S_p[opc]$ and $R_p[opc]$ not being updated.

The fourth operation type, denoted $HARG\_HRET\_T$, indicates a base operation with an argument list and returned value list. For example, the *remove* operation of a list takes a specified position as the argument and returns the value of the removed entry. The sequential specification of $D\{B\}$ objects for this operation type, as shown in Figure 4.6, is combined from the specifications of $D\{B\}$ for the operation type $HARG\_NRET\_T$ and $NARG\_HRET\_T$ with all three values ($S_p[op]$, $A_p[op]$, $R_p[op]$) to record the bookkeeping data. Axiom 4.38 shows that *op* is executed without detectability. For the detectable operation, the sequential specification of *prepareOp* in Axiom 4.33 is the same as Axiom 4.7, with $S_p[op]$ and $A_p[op]$ being updated. In Axiom 4.34, *op* is executed with the value in $A_p[op]$. Then $R_p[op]$ is updated with the returned value list and $S_p[op]$ is updated to END_DECT_OP. For a complete operation, both $R_p[op]$ and $A_p[op]$ are returned in *retrieveOp* (Axiom 4.36). For an incomplete operation, the specification of *retrieveOp* in Axiom 4.35 is the same as Axiom 4.9, returning *NoEffect* and $A_p[op]$.

The specification of $D'\{B\}$ objects for the operation type $HARG\_HRET\_T$ is shown in Figure 4.7. Compared to the specification of $D\{B\}$, the differences are in the *detectableOp* and *retrieveOp* functions. For the *retrieveOp* function, the argument value is not returned (Axiom 4.43 and Axiom 4.44). For the *detectableOp* function, as in the specification of $D'\{B\}$ for the operation type $NARG\_HRET\_T$, the operations are divided into *opc* and *opo*. The axiom for *detectableOpo* is the same as Axiom 4.34. There are two axioms for *detectableOpc* to deal with two conditions: the *opc* updates the state or not. Axiom 4.40 and Axiom 4.41 show that only when the *opc* updates the state, $S_p[opc]$ and $R_p[opc]$ are updated.

$$\begin{gathered}\{true\}\\ [prepareOp()/res(), p]\\ \{S'_p[op] = BEGIN\_DECT\_OP\}\end{gathered} \tag{Axiom 4.25}$$

$$\begin{gathered}\{S_p[opc] = BEGIN\_DECT\_OP \wedge s \in UStatus[opc]\}\\ [detectableOpc()/res(\rho(s, opc)), p]\\ \{s' = \delta(s, opc) \wedge s' \neq s \wedge\\ S'_p[opc] = END\_DECT\_OP \wedge R'_p[opc] = \rho(s, opc)\}\end{gathered} \tag{Axiom 4.26}$$

$$\begin{gathered}\{S_p[opc] = BEGIN\_DECT\_OP \wedge s \notin UStatus[opc]\}\\ [detectableOpc()/res(\rho(s, opc)), p]\\ \{s' = \delta(s, opc) \wedge s' = s\}\end{gathered} \tag{Axiom 4.27}$$

$$\begin{gathered}\{S_p[opo] = BEGIN\_DECT\_OP\}\\ [detectableOpo()/res(\rho(s, opo)), p]\\ \{s' = \delta(s, opo) \wedge S'_p[opo] = END\_DECT\_OP \wedge R'_p[opo] = \rho(s, opo)\}\end{gathered} \tag{Axiom 4.28}$$

$$\begin{gathered}\{S_p[op] = BEGIN\_DECT\_OP\}\\ [retrieveOp()/res(NoEffect, \perp), p]\\ \{\}\end{gathered} \tag{Axiom 4.29}$$

$$\begin{gathered}\{S_p[op] = END\_DECT\_OP\}\\ [retrieveOp()/res(HasEffect, R_p[op]), p]\\ \{\}\end{gathered} \tag{Axiom 4.30}$$

$$\begin{gathered}\{S_p[op] = \perp\}\\ [retrieveOp()/res(NoExecutedOp, \perp), p]\\ \{\}\end{gathered} \tag{Axiom 4.31}$$

$$\begin{gathered}\{true\}\\ [op()/res(\rho(s, op)), p]\\ \{s' = \delta(s, op)\}\end{gathered} \tag{Axiom 4.32}$$

Figure 4.5: The sequential specification for $D'\{B\}$ objects regarding operation type $NARG\_HRET\_T$.

$$\{true\}$$
$$[prepareOp(a)/res(), p] \tag{Axiom 4.33}$$
$$\{A'_p[op] = a \wedge S'_p[op] = BEGIN\_DECT\_OP\}$$

$$\{S_p[op] = BEGIN\_DECT\_OP\}$$
$$[detectableOp()/res(\rho(s, op(A_p[op]))), p]$$
$$\{s' = \delta(s, op(A_p[op])) \wedge S'_p[op] = END\_DECT\_OP \wedge R'_p[op] = \rho(s, op(A_p[op]))\} \tag{Axiom 4.34}$$

$$\{S_p[op] = BEGIN\_DECT\_OP\}$$
$$[retrieveOp()/res(NoEffect, \perp, A_p[op]), p] \tag{Axiom 4.35}$$
$$\{\}$$

$$\{S_p[op] = END\_DECT\_OP\}$$
$$[retrieveOp()/res(HasEffect, R_p[op], A_p[op]), p] \tag{Axiom 4.36}$$
$$\{\}$$

$$\{S_p[op] = \perp\}$$
$$[retrieveOp()/res(NoExecutedOp, \perp, \perp), p] \tag{Axiom 4.37}$$
$$\{\}$$

$$\{true\}$$
$$[op(a)/res(\rho(s, op(a))), p] \tag{Axiom 4.38}$$
$$\{s' = \delta(s, op(a))\}$$

Figure 4.6: The sequential specification for $D\{B\}$ objects regarding operation type $HARG\_HRET\_T$.

$$\begin{gathered}\{true\}\\ [prepareOp(a)/res(),p]\\ \{A'_p[op]=a \wedge S'_p[op]=BEGIN\_DECT\_OP\}\end{gathered} \tag{Axiom 4.39}$$

$$\begin{gathered}\{S_p[opc]=BEGIN\_DECT\_OP \wedge s \in UStatus[opc]\}\\ [detectableOpc()/res(\rho(s,opc(A_p[op]))),p]\\ \{s'=\delta(s,opc(A_p[op])) \wedge s' \neq s \wedge\\ S'_p[opc]=END\_DECT\_OP \wedge R'_p[opc]=\rho(s,opc(A_p[op]))\}\end{gathered} \tag{Axiom 4.40}$$

$$\begin{gathered}\{S_p[opc]=BEGIN\_DECT\_OP \wedge s \notin UStatus[opc]\}\\ [detectableOpc()/res(\rho(s,opc(A_p[op]))),p]\\ \{s'=\delta(s,opc(A_p[op])) \wedge s' = s\}\end{gathered} \tag{Axiom 4.41}$$

$$\begin{gathered}\{S_p[opo]=BEGIN\_DECT\_OP\}\\ [detectableOpo()/res(\rho(s,opo(A_p[op]))),p]\\ \{s'=\delta(s,opo(A_p[op])) \wedge S'_p[opo]=END\_DECT\_OP \wedge R'_p[opo]=\rho(s,opo(A_p[op]))\}\end{gathered} \tag{Axiom 4.42}$$

$$\begin{gathered}\{S_p[op]=BEGIN\_DECT\_OP\}\\ [retrieveOp()/res(NoEffect,\perp),p]\\ \{\}\end{gathered} \tag{Axiom 4.43}$$

$$\begin{gathered}\{S_p[op]=END\_DECT\_OP\}\\ [retrieveOp()/res(HasEffect,R_p[op]),p]\\ \{\}\end{gathered} \tag{Axiom 4.44}$$

$$\begin{gathered}\{S_p[op]=\perp\}\\ [retrieveOp()/res(NoExecutedOp,\perp),p]\\ \{\}\end{gathered} \tag{Axiom 4.45}$$

$$\begin{gathered}\{true\}\\ [op(a)/res(\rho(s,op(a)),p]\\ \{s'=\delta(s,op(a)\}\end{gathered} \tag{Axiom 4.46}$$

Figure 4.7: The sequential specification for $D'\{B\}$ objects regarding operation type $HARG\_HRET\_T$.

4.4 Definition of Detectability

I consider detectability is only for base operations. For example, the base operations of *DetectableT* integrated with *queue* are *enqueue* and *dequeue*. According to the sequential specification, a *prepareEnqueue* operation is also added to $D\{queue\}$. However, detectability only applies to the *enqueue* operation because *prepareEnqueue* is an auxiliary operation to help get the effect of *enqueue*.

I define two levels of detectability: strong detectability and weak detectability. The type $D\{B\}$ represents strong detectability and the type $D'\{B\}$ represents weak detectability.

Based on the type $D\{B\}$, I give the definition of strong detectability.

Definition 4.1 (Strong Detectability)**.** Given a data type T, T is strongly detectable if there exists a type B such that $T = D\{B\}$.

For example, if $T = D\{queue\}$, T is strongly detectable. The base type of T is *queue*, and the base operations of T are *enqueue* and *dequeue*. There are three auxiliary functions for each base operation in T, such as *prepareEnqueue*, *detectableEnqueue* and *retrieveEnqueue*. When calling the *retrieveEnqueue* function or *retrieveDequeue* function, the information including the returned value and the argument list, defined in the sequential specification of $D\{queue\}$, is returned.

Similarly, based on the type $D'\{B\}$, I give the definition of weak detectability.

Definition 4.2 (Weak Detectability)**.** Given a data type T, T is weakly detectable if there exists a type B such that $T = D'\{B\}$.

Chapter 5

Implementing Detectable Queue

In Chapter 4, I provide the definition of detectability. In this chapter, I will talk about how to implement detectable data types based on the sequential specification defined in Section 4.4 through a queue example.

5.1 Private Variables

I assume that there are two types of variables in the persistent memory: shared variables which can be accessed by all threads, and private variables which belong to one specific thread. Generally speaking, a private variable can only be accessed by its owner thread and a single recovery thread. During recovery, a private variable may be accessed by a thread who is responsible for recovery to regain consistency. Private variables can be applied to recovering persistent data structures. A thread can store some bookkeeping information into its private variables, and inspect them during recovery to arrive at a clear picture of what happened before a crash. For example, in Golab and Hendler's recoverable queue lock [14], when entering the critical section, every thread attempts to append itself to the tail of the shared lock queue, and save the old value of the tail to its private variable. When leaving the critical section, the thread updates its private variable again and removes itself from the queue. Then after restarting, every thread determines whether it is in the lock queue by inspecting the value in its private variable.

5.2 Detectable Execution Using Private Variables

Different persistent and detectable data structures have been designed [6, 11, 13]. To achieve detectability, normally, when an operation happens, some bookkeeping information is written into auxiliary variables to record its status, arguments and returned values. During recovery, the restarting process will read the information from specific memory locations to obtain the effect of previous operations. For example, shared log variables are used to store operations' bookkeeping information in and Friedman et al.'s persistent queue [13].

I consider using auxiliary private variables, which can only be accessed by their owner thread and a single recovery thread, to store the effects of each operation to avoid contention. As there is no read-write contention on such variables, the bookkeeping procedure for the operations can be comparatively simplified and thus sped up to a certain degree.

I assume in the initialization, the application allocates a certain area of persistent memory for private variables and assigns each thread private variables for every operation defined in a data type. Then, a thread can get its private variables for an operation through the getPrivateAddrByOpName function, which takes an operation name and an index as the arguments. The index is for multiple private variables. If there is only one private variable, the index argument can be omitted. The initial value of any private variable is *MEM_INIT_VAL*. Another function provided by the system is selfThreadID which returns the caller thread's ID.

5.3 Detectable Queue Implementations

According to the definition of detectability in Section 4.4, to provide detectability for a base type B, the implementation must be based on the sequential specification of $D\{B\}$, which adds three auxiliary functions for every operation in B. To demonstrate how to implement a detectable data type, I use queue, one of the most fundamental data types, as the base type, and design a detectable queue algorithm, called Detectable Queue, based on MS queue [28] for persistent memory. MS queue, using a linked list to implement a FIFO queue, is a widely-used lock-free concurrent data structure serving as one of queue algorithms in the Java package java.util.concurrent.

The classes of Detectable Queue are shown in Figure 5.1. The DetectableQNode class contains a data pointer (pData_), a next node pointer (next_), and a thread ID (deqThreadID_) marking the node being removed from the linked list by a thread. The initial value of deqThreadID_ is -1. All the queue nodes are allocated from a node pool in the persistent memory through an epoch-based memory management algorithm which will be discussed in Chapter 7. In the DetectableQueue class, the phead_ field points to a sentinel node and ptail_ points to the last node of the queue. If phead_ and ptail_ point to the same node, it means the queue is empty with only a sentinel node. To show the differences between the strong detectable queue and weak detectable queue, I represent both algorithms and define the isStrongDetectability_ field in the queue class. In practice, for an application, normally only one algorithm is needed. In the class, a total of six auxiliary functions for the enqueue and dequeue operations are added.

```
class DetectableQNode {
    uint64_t * pData_;
    DetectableQNode * next_ = NULL;
    uint64_t deqThreadID_ = -1;};

class DetectableQueue {
    DetectableQNode * phead_;
    DetectableQNode * ptail_;
    bool isStrongDetectability_;

    void prepareEnqueue(DetectableQNode * enqNode);
    void detectableEnqueue();
    void retrieveEnqueue(uint64_t* e0, uint64_t* e1, uint64_t* a);
    void enqueue(DetectableQNode * enqNode);

    void prepareDequeue();
    DetectableQNode * detectableDequeue();
    void retrieveDequeue(uint64_t* e0, uint64_t* e1);
    DetectableQNode * dequeue();
};
```

Figure 5.1: Detectable Queue classes

5.3.1 Enqueue Operation

The enqueue operation belongs to operation type *HARG_NRET_T*. According to the sequential specification of $D\{queue\}$, two pieces of information are needed to get the operation's effect: the executing status and the argument. In my implementation, I use one private variable to store the new queue node and decide the status based on the value in the private variable. Since modern 64-bit x86 processors implement 48 address bits [36], the higher 16 bits can be used to store extra information. I define a flag, called *OPCOMPLETE_FLG*, to indicate an operation is complete, which uses the highest bit of a word. The *OPCOMPLETE_FLG* flag is added to the node pointer in the private variable when an enqueue operation completes. When getting the effect of a complete enqueue operation, for strong detectability, the argument is returned while for weak detectability, the argument of an enqueue operation is not returned when getting its effect.

```
void DetectableQueue::prepareEnqueue(DetectableQNode * node) {
    uint64_t * pAddr = getPrivateAddrByOpName(ENQUEUE);
    flush(node);
    *pAddr = (uint64_t)node;
    flush(pAddr);
}
void DetectableQueue::detectableEnqueue() {
    uint64_t * pAddr = getPrivateAddrByOpName(ENQUEUE);
    DetectableQNode * node = (DetectableQNode *)(*pAddr);
    while(true) {
        DetectableQNode * last = ptail_;
        DetectableQNode * next = last->next_;
        if(last == (ptail_)) {
            if(next == NULL) {
                if(CAS(&last->next_, NULL, node) == NULL){
                    flush(&last->next_);
                    *pAddr = (*pAddr) | OPCOMPLETE_FLG;
                    flush(pAddr);
                    CAS(&ptail_, last, node);
                    return;}}
            else {
                flush(&last->next_);
                CAS(&ptail_, last, next);}}}}
```

Figure 5.2: The prepareEnqueue and detectableEnqueue functions of Detectable Queue

In the *prepareEnqueue* function, the new queue node is flushed in to the persistent memory first. Then the node pointer is stored in the private variable and persisted as shown in Figure 5.2. The private variable is obtained by *getPrivateAddrByOpName* with the enqueue operation name as the argument. Then in the *detectableEnqueue* function, the new node is fetched from the private variable to be enqueued.

The executing thread first reads the last node and its next pointer from the queue. If the next pointer is not NULL, which means another node has already been appended, the thread helps to persist the next pointer and move the tail pointer using CAS (lines 98-99). The tail pointer does not need to be flushed because it can be deduced by the last pointer of the queue, which has been persisted. If no node has been appended to the last node, the executing thread tries to update the next pointer to the new node using CAS (line 91). If the CAS is successful, the thread flushes the next pointer and moves the tail pointer to the new node. The thread also adds the complete flag *OPCOMPLETE_FLG* to the private variable and persists it to indicate that this operation is complete. If crash happens after the node being appended and before the complete flag *OPCOMPLETE_FLG* being added, the complete flag will be added through recovery. If the CAS failed, the thread will retry by rereading the last node until it succeeds.

```
struct EnqueueEffect {
    uint64_t e = MEM_INIT_VAL;
    uint64_t r = MEM_INIT_VAL;
    uint64_t a = MEM_INIT_VAL;
}

EnqueueEffect DetectableQueue::retrieveEnqueue() {
    EnqueueEffect effect;
    uint64_t * pAddr = getPrivateAddrByOpName(ENQUEUE);
    if(*pAddr == MEM_INIT_VAL) {
        effect.e = NoExecutedOp;
    }
    else if(*pAddr & OPCOMPLETE_FLG != 0) {
        effect.e = HasEffect;
        effect.r = AckResponse;
        if(isStrongDetectability_) effect.a = *pAddr | ~
            OPCOMPLETE_FLG;
    }
    else {
        effect.e = NoEffect;
        if(isStrongDetectability_) effect.a = *pAddr;
    }
    return effect;
}
```

Figure 5.3: The retrieveEnqueue function of Detectable Queue

A thread calls the *retrieveEnqueue* function to get the effect, which is based on the value in the private variable. If the value is *MEM_INIT_VAL*, which means the *preparedEnqueue* function is never called by the thread, *NoExecutedOp* is returned. If *OPCOMPLETE_FLG*

```
void DetectableQueue::enqueue(DetectableQNode * node) {
    flush(node);
    while(true) {
        DetectableQNode * last = ptail_;
        DetectableQNode * next = last->next_;
        if(last == (ptail_)) {
            if(next == NULL) {
                if(CAS(&last->next_,NULL,node) == NULL){
                    flush(&last->next_);
                    CAS(&ptail_, last, node);
                    return;}}
            else {
                flush(&last->next_);
                CAS(&ptail_, last, next);}}}}
```

Figure 5.4: The enqueue function of Detectable Queue

is added in the value by the executing thread or recovery thread, which means the last enqueue operation is complete before crash or during recovery, *HasEffect* is returned. Otherwise, the value must be a new queue node to be enqueued and *NoEffect* is returned. For strong detectability, the argument stored in the private variable is also returned. There are no differences in the *prepareEnqueue* and *detectableEnqueue* functions for the strong detectability and weak detectability. The reason is that there is only one variable in my implementation representing both the executing status and input argument.

I also provide an *enqueue* function for persistent memory without detectability as shown in Figure 5.4.

Compared to the *detectableEnqueue* function, the *enqueue* function does not need to store the result into the private variable. The lines of code responsible for setting *OP-COMPLETE_FLG* and persisting the private variable are removed. Besides, the queue node needs to be flushed into persistent memory in the beginning. Since it lacks detectability, after restarting, a thread cannot know the result of its previous *enqueue* operation. A memory leak may happen if a new queue node has already been allocated but not appended to the queue in a system without memory garbage collection mechanism.

5.3.2 Dequeue Operation

The dequeue operation belongs to operation type *NARG_HRET_T*. According to the sequential specification of $D\{queue\}$, two variables are needed: a variable to store the executing status and a variable to store the returned queue node. Similarly to the detectable enqueue algorithm, I use one private variable to store the returned node and decide the status based on the value in the private variable. For weak detectability, the private variable is not updated when a NULL pointer is returned from an empty queue.

```
void DetectableQueue::prepareDequeue() {
    uint64_t * pAddr = getPrivateAddrByOpName(DEQUEUE);
    *pAddr = OP_INIT_VAL;
    flush(pAddr);
}
```

Figure 5.5: The prepareDequeue function of Detectable Queue

```
struct DequeueEffect {
    uint64_t e = MEM_INIT_VAL;
    uint64_t r = MEM_INIT_VAL;
}

DequeueEffect DetectableQueue::retrieveDequeue() {
    DequeueEffect effect;
    uint64_t * pAddr = getPrivateAddrByOpName(DEQUEUE);
    if(*pAddr == MEM_INIT_VAL) {
        effect.e = NoExecutedOp;
    }
    else if(*pAddr == OP_INIT_VAL) {
        effect.e = NoEffect;
    }
    else {
        effect.e = HasEffect;
        if(*pAddr == NULL) {
            effect.r = NULL;
        } else {
            effect.r = (*pAddr)->next_;
        }
    }
    return effect;
}
```

Figure 5.6: The retrieveDequeue function of Detectable Queue

In the *prepareDequeue* function (Figure 5.5), *OP_INIT_VAL* is written to the private

```
DetectableQNode * DetectableQueue::detectableDequeue() {
    uint64_t * pAddr = getPrivateAddrByOpName(DEQUEUE);
    while(true) {
        DetectableQNode * first = phead_;
        DetectableQNode * next = first->next_;
        DetectableQNode * last = ptail_;
        if(first == phead_) {
            if(first == last) {
                if(next == NULL) {
                    if(isStrongDetectability_) {
                        *pAddr = NULL;
                        flush(pAddr);
                    }
                    return NULL;
                }
                flush(&last->next);
                CAS(&ptail_, last, next);
            }
            else {
                *pAddr = first;
                flush(pAddr);

                if(CAS(&first->deqThreadID_, selfThreadID(), -1) ==
                    -1) {
                    flush(&first->deqThreadID_);
                    CAS(&phead_, first, next);
                    return next;
                }
                else {
                    if(first == phead_) {
                        flush(&first->deqThreadID_);
                        CAS(&phead_, first, next);
                    }
                }
            }
        }}}
```

Figure 5.7: The detectableDequeue function of Detectable Queue

variable and persisted, which indicates that an operation begins. In the *retrieveDequeue* function (Figure 5.6), the effect is based on the value in the private variable. The effect value of *NoExecutedOp* is returned when the value is *MEM_INIT_VAL* and *NoEffect* is returned when the value is *OP_INIT_VAL*. Otherwise, *HasEffect* is returned. If the value is a queue node, the next pointer of the queue node is returned. If the value is a NULL pointer, a NULL pointer is returned.

The pseudo code of the *detectableDequeue* function is shown in Figure 5.7. The operation reads the first node N_f from the head pointer P_h, the next node N_n from the next pointer of N_f, and the last node N_l from the tail pointer P_t (line 169-171). If N_f and N_l point to the same node and N_n is NULL, which means there is only one sentinel node in the queue, a NULL pointer is returned. For strong detectability, the NULL pointer is set to the private variable and flushed. If N_f equals to N_l and N_n is not NULL, which means a new node has just been appended to the last node, the operation helps by persisting N_n and moving P_t to N_n.

```
DetectableQNode * DetectableQueue::dequeue() {
    while(true) {
        DetectableQNode * first = phead_;
        DetectableQNode * next = first->next_;
        DetectableQNode * last = ptail_;
        if(first == phead_) {
            if(first == last) {
                if(next == NULL) {{
                    return NULL;}
                flush(&last->next);
                CAS(&ptail_, last, next);}
            else {
                if(CAS(&first->deqThreadID_, selfThreadID(), -1) == -1) {
                    flush(&first->deqThreadID_);
                    CAS(&phead_, first, next);
                    return next;
                }
                else {
                    if(first == phead_) {
                        flush(&first->deqThreadID_);
                        CAS(&phead_, first, next);
                    }}}}}}
```

Figure 5.8: The dequeue function of Detectable Queue

If N_f and N_l point to different nodes, which means the queue is not empty, the executing thread, called T_i, will try to set its thread ID to the *deqThreadID_* field of N_f and move P_h to N_n. T_i puts N_f to its private variable and persists it first. Then, T_i executes a CAS to update the *deqThreadID_* field of N_f to its own thread ID. If the CAS succeeds, T_i persist the *deqThreadID_* field, moves P_h to N_n and returns N_n, which is the dequeued node. If the CAS fails, which means that another thread successfully updated the *deqThreadID_* field, T_i will retry the procedure. T_i has to write N_f to its private variable before it executes CAS. The reason is that if T_i sleeps after a successful CAS, since a private variable can only be accessed by its own thread, other threads can not help T_i put N_f to its private variable. Then if a crash happens, N_f will be lost. The recovery procedure will check every thread's dequeue private variable to make sure the private variable only holds a node whose *deqThreadID_* is the variable's owner thread. The head pointer P_h does not need to be persisted after being updated because during recovery, P_h can be moved forward to the first node whose deqThreadID_ field is -1.

I also provide a *dequeue* function without detectability as shown in Figure 5.8. Compared to the *detectableDequeue* function, the code lines of storing the returned node and persisting the private variable are removed. Since no detectability is required, the returned node is not stored, which maybe lost after a crash. The thread cannot obtain the result of its previous *dequeue* operation even if this dequeue operation was actually successful. For a system without garbage collection, this may cause memory leak.

5.3.3 Recovery

When the computer restarts from a crash, a single recovery thread T_i is started by the application to recover the detectable queue. T_i, which has access to all working threads' private variables, will recover pending operations by reading the values of private variables and setting new values to them. Then, working threads can call the *retrieveEnqueue* and *retrieveDequeue* functions to get the effects of their previous operations and then begin new enqueuing and dequeuing operations. T_i traverses the queue from the head pointer and checks the value in the $deqThreadID\_$ field of the current visiting node N_i. If the value is not -1, T_i moves the head pointer to the next pointer of N_i and flushes the head pointer. If N_i is in a thread's enqueue private variable without the *OPCOMPLETE_FLG* flag, *OPCOMPLETE_FLG* flag is set to the node pointer to indicate the enqueue operation is complete. Then the updated value in the enqueue private variable is flushed. If T_i reaches the tail pointer and the current node N_i's next pointer is not NULL, the tail pointer is moved to the next pointer of N_i and flushed to the persistent memory. After traversing the queue, T_i visits every thread's enqueue and dequeue private variables. For a node pointer without the *OPCOMPLETE_FLG* flag in the enqueue private variable, if the *deqThreadID_* field of the node pointer is not -1, which means the node has been appended to the queue and removed by another thread, *OPCOMPLETE_FLG* is added to the node pointer and the updated value is flushed. Because of the memory management, this node cannot be reclaimed before crash. For a node pointer in the dequeue private variable, if the *deqThreadID_* field of the node pointer does not equal to the private variable's owner thread ID, *OP_INIT_VAL* is written to the private variable and flushed to show that the dequeue operation does not take effect. After the recovery procedure, working threads can get effects based on the values in their private variables.

Chapter 6

CASWithEffect Primitive

In chapter 5, I presented how to use private variables to implement detectable persistent data structures through a lock-free MS queue example. To make the data structure persistent and detectable, several store and flush instructions are added explicitly. One difficulty of designing a detectable MS queue is that the result of a CAS operation is in the volatile CPU registers, which will be lost after a crash. To simplify the detectable data structure design, I provide a synchronization primitive called CASWithEffect, which atomically executes a CAS operation on a shared variable and stores the effect of this CAS operation into an auxiliary private variable.

The CASWithEffect primitive performs a CAS operation to update a target variable with a new value and stores the effect into a specified auxiliary private variable. If a crash happens when executing CASWithEffect, the caller process can access its private variable to get the effect of the CAS operation after restarting. CASWithEffect takes a succeeded effect value se and a failed effect fe value as arguments. The initial value of the private variable is iv. Suppose process p executes CASWithEffect with an old value o and a new value n. If the CAS operation is successful, se is put into the private variable and o is returned. Otherwise, fe is put into the private variable and the current value in the target variable is returned. After a crash, p reads the value from its private variable. If the value equals to se, it means the CAS operation has successfully updated the target address. If the value equals to fe, it means the CAS failed in updating the target address because the old value o did not match the value in the target address. If the value equals to iv, it means the CAS did not take effect at all. Some algorithms do not need to know the CAS failed and they can choose not to provide the failed effect value. Then, when a CAS failed, nothing is written to the private variable.

Although CASWithEffect is a primitive, not a data type, it can be transformed into a data type with state. The state for the CASWithEffect data type includes the value tv in the target variable and the value $pv[p]$ in the private variable belonging to process p. Then the arguments of the *CASWithEffect* operation only have four arguments: the old value, the new value, the succeeded effect and the failed effect. The sequential specification is defined in Figure 6.1. Axiom 6.1 shows that when the old value o matches tv, the new value n is put to the target variable and the succeeded effect se is put into the executing process' private variable with o being returned. Axiom 6.2 shows that when the old value does not match tv, the failed effect fe is put into the executing process' private variable with tv being returned. Axiom 6.3 shows the *CASWithEffectRead* operation returns the value in the target variable.

$$\begin{gathered}\{o = tv\}\\ [CASWithEffect(o, n, se, fe)/res(o)]\\ \{tv' = n \land pv'[p] = se\}\end{gathered} \tag{Axiom 6.1}$$

$$\begin{gathered}\{o \neq tv\}\\ [CASWithEffect(o, n, se, fe)/res(tv)]\\ \{pv'[p] = fe\}\end{gathered} \tag{Axiom 6.2}$$

$$\begin{gathered}\{true\}\\ [CASWithEffectRead()/res(tv)]\\ \{\}\end{gathered} \tag{Axiom 6.3}$$

Figure 6.1: Axioms for CASWithEffect

Although the effect of CAS is stored in a private variable, the difficulty of storing the effect atomically is that the system may crash after the target variable being updated and before the effect being stored. After restarting, the private variable can not be updated based on the value of the target variable, because the target variable, as a shared variable, may have already been updated by others before or after the crash. The Persistent Multi-Word Compare-And-Swap (PMwCAS) algorithm can modify arbitrary words in the persistent memory atomically and recover pending operations [36]. Based on the PMwCAS algorithm, I design an algorithm which can atomically update a shared variable (target variable) and a private variable, and recover pending operations.

6.1 PMwCAS Algorithm

Wang et al.'s lock-free PMwCAS algorithm [36], by extending Harris et al.'s MwCAS algorithm [19] to persistent memory, provides a mechanism to atomically modify multiple words and recover any pending operation after a crash. As in MwCAS, PMwCAS uses descriptors to perform atomic CAS operations on arbitrary combinations of words. There are two types of descriptor: the word descriptor and the PMwCAS descriptor. A word descriptor consists of one target address, the old value, and new value of that address. A PMwCAS descriptor represents one PMwCAS operation, holding a list of word descriptors. To indicate the progress of a PMwCAS operation, there is a field called status in a PMwCAS descriptor. When a PMwCAS operation begins, the value of status becomes "Undecided", showing that an operation is in progress. Then the operation is divided into two phases. The first phase is to install the PMwCAS descriptor into each target address if the current value of the address equals the old value in the descriptor. Since PMwCAS is lock-free, if one thread visits a memory location holding a descriptor of another thread, it will help to complete that operation first and then execute its own operation. This ensures that no thread is blocked. If PMwCAS uses CAS to compare the current value with the old value and set the PMwCAS descriptor to the target address if they are equal, an ABA problem may appear. For example, thread t_1 is executing a PMwCAS operation to update the target addresses X_1 and X_2 using a PMwCAS descriptor d and successfully installs d on X_1. Before t_1 executes CAS to install d on X_2 whose value is A, t_1 is delayed. Then other threads, reading d from X_1, help to finish d's operation by updating both X_1 and X_2 to new values of d. After this, some thread performs another PMwCAS operation on X_2 and changes the value back to A. When t_1 wakes up, it executes CAS successfully and changes the value of X_2 to descriptor d whose operation has already been completed. In this situation, the new value of d may be set to X_2 again. To avoid this problem, the word descriptor of a PMwCAS descriptor is installed on a target address first. Then the PMwCAS descriptor is installed by using CAS to check whether the target address holds the word descriptor. In this way, even if a word descriptor w_1 of a PMwCAS descriptor d is installed incorrectly due to the ABA problem, the thread can check d's status and change the value back if d's operation has already been finished.

To distinguish descriptor pointers from ordinary pointers, some of the most significant bits of pointers are repurposed as the PMwCAS descriptor flag and word descriptor flag. If the PMwCAS descriptor has been installed to every target address, the status becomes "Succeeded". Otherwise, the status becomes "Failed", denoting that the operation fails. After the status is determined, the second phase begins. When the status equals to "Succeeded", the operation will use CAS to put new values into all the target addresses which currently store the descriptor pointer; when the status equals to “Failed”, the operation will roll back the target addresses which hold the descriptor pointer to their old values.

A reading operation is also provided for memory locations affected by PMwCAS. If the location stores a descriptor, the reading operation needs to help PMwCAS complete first and then retries until the stored value is no longer a descriptor.

The recovery of PMwCAS depends on the status field of a descriptor. When a target address holds a PMwCAS descriptor, the recovery rolls the value forward to the new value when the status of the descriptor is “Succeeded”. When a target address holds a PMwCAS descriptor or a word descriptor, the recovery rolls the value back to the old value when the status is “Failed” or “Undecided”.

6.2 General CASWithEffect Algorithm

Based on the two phase updates of PMwCAS, I develop a new algorithm (General CASWith-Effect), which atomically modifies a shared variable and private variable. Since the contention is only on a shared address, General CASWithEffect only applies a descriptor pointer to a shared address for atomicity and modifies the private address directly. In the helping procedure, one thread helps another one only with the shared address. In this way, compared to using PMwCAS to update two shared variables, General CASWithEffect can improve performance by reducing nearly half of the overhead caused by installing descriptors.

6.2.1 The CASWithEffect Descriptor

Now I discuss how to implement General CASWithEffect algorithm. First, I define the CASWithEffect descriptor as shown in Figure 6.2. Since CASWithEffect only modifies one shared variable, there is no need to define two descriptors to handle the ABA problem as in PMwCAS. The CASWithEffect descriptor includes the shared and private addresses, new and old values, succeeded and failed effects. As in PMwCAS, the status_ field, whose initial value is *Undecided*, shows the result of the CAS operation. The isFEffectNeeded_ field shows whether the failed effect is needed. The isPrivateValueSet_ field, with an initial value of $false$, indicates whether the effect has been written to the private address. When isPrivateValueSet_ is $false$, the recovery procedure will update the private variable with the effect value for a complete CAS operation. As in PMwCAS, I use the highest bit of a pointer to indicate whether the pointer points to a CASWithEffect descriptor, and denote it *DESCRIPTOR_FLG*. In the beginning, a descriptor pool is allocated with a certain number of descriptors. The descriptor size is calculated based on the number of working threads. In my experiment, each thread is assigned 2000 descriptors. When executing a *CASWithEffect* operation, a thread needs to get a descriptor from the descriptor pool using the *allocNewDescriptor* function. After the operation being completed, the descriptor is released by the thread. I use an epoch-based memory reclamation approach, which can safely recycle descriptors. The memory management of this will be discussed in details in Chapter 7.

```
class CASWithEffectDescriptor {
    uint64_t* shareAddr_;    //the shared address
    uint64_t* privateAddr_;  //the private address
    uint64_t oValue_;          //old value of the shared address
    uint64_t nValue_;      //new value of the shared address
    uint64_t sEffect_;     //the effect value for a succeeded CAS
    uint64_t fEffect_ ;     //the effect value for a failed CAS
    bool isFEffectNeeded_ = false;
    uint64_t status_ = Undecided;
    bool isPrivateValueSet_ = false;}
```

Figure 6.2: The CASWithEffectDescriptor class.

6.2.2 The CASWithEffect Procedure

The pseudo code of the *CASWithEffect* function is shown in Figure 6.3 with all the needed information as arguments including the failed effect value. I also provide another *CASWith-*

```
uint64_t CASWithEffect(uint64_t* shareAddr, uint64_t* privateAddr,
    uint64_t oValue, uint64_t nValue,
    uint64_t sEffect, uint64_t fEffect)
{
    CASWithEffectDescriptor * fd =  genDescriptor(shareAddr,
        privateAddr, oValue, nValue, sEffect, fEffect);
    return exeCASWithEffect(fd);
}
```

Figure 6.3: The CASWithEffect function for General CASWithEffect.

Effect function without the fEffect argument for callers who do not need the failed effect.

The *CASWithEffect* function calls two support functions: *genDescriptor* and *exeCASWithEffect*. The *genDescriptor* function allocates a new descriptor from the descriptor pool and sets values to its members as shown in Figure 6.4. The *genDescriptor* function returns the allocated descriptor and the *exeCASWithEffect* function receives this descriptor to execute a CAS operation and store its effect.

```
CASWithEffectDescriptor * genDescriptor(
          uint64_t* shareAddr, uint64_t* privateAddr,
          uint64_t oValue, uint64_t nValue,
          uint64_t sEffect, uint64_t fEffect)
{
     CASWithEffectDescriptor * fd =  allocNewDescriptor();
     fd->shareAddr_ = shareAddr;
     fd->privateAddr_ = privateAddr;
     fd->oValue_ = oValue;
     fd->nValue_ = nValue;
     fd->sEffect_ = sEffect;
     fd_->isFEffectNeeded_ = true;
     fd_->fEffect_ = fEffect;

     return fd;
}
```

Figure 6.4: The genDescriptor function for General CASWithEffect.

The *exeCASWithEffect* function begins with flushing the CASWithEffect descriptor fd to the persistent memory as shown in Figure 6.5. The operation first tries to install the descriptor into the shared variable by using CAS to compare the old value to the current value in that address. If the returned value is another descriptor pointer, the thread helps to finish that descriptor's operation by calling the *persistStatusAndShareAddr* function to update and persist the operation's status and shared variable. In the *persistStatusAndShareAddr* function (lines 283 - 287), the shared address with the descriptor is persisted first so that if a crash happens, the descriptor in that address is not lost. Then, the status is changed to *Succeeded* and persisted. After that, the new value is set to the shared variable. The new value does not need to be persisted, because with the descriptor and status being in the persistent memory, the recovery can guarantee the new value will be set back to the shared variable if it is lost in the crash. After helping, the thread retries CAS until the returned value is a regular value. If the returned regular value does not equal to the old value, which means a lost race on the shared variable, the operation sets the *status_* field of fd to *Failed* and puts the failed effect value to the private variable of fd if needed.

```
uint64_t exeCASWithEffect(CASWithEffectDescriptor * fd) {
    flush(fd);
    uint64_t installVal = installDescriptorToSharedAddr(fd);
    if(installVal != fd->oValue_) {
        if(fd->isFEffectNeeded_) {
            fd->status_ = Failed;
            flush(&fd->status_);
            setEffectToPrivateAddr(fd, fd->fEffect_);
        }
        return installVal;
    }
    persistStatusAndShareAddr(fd);
    setEffectToPrivateAddr(fd, fd->sEffect_);
    return fd->oValue_;
}

uint64_t installDescriptorToSharedAddr(CASWithEffectDescriptor * fd) {
retry:
    uint64_t ret = CAS(fd->shareAddr_,fd->oValue_, fd|DESCRIPTOR_FLG);
    if(ret & DESCRIPTOR_FLG != 0) {
        persistStatusAndShareAddr(ret&(~DESCRIPTOR_FLG));
        goto retry;
    }
    return ret;
}

void persistStatusAndShareAddr(CASWithEffectDescriptor * fd) {
    flush(fd->shareAddr_);
    fd->status_ = Succeeded;
    flush(&fd->status_);
    CAS(fd->shareAddr_, fd, fd->nValue_);
}

void setEffectToPrivateAddr(CASWithEffectDescriptor * fd, uint64_t
    effect) {
    *(fd->privateAddr_) = effect;
    flush(fd->privateAddr_);
    fd->isPrivateValueSet_ = true;
    flush(&fd->isPrivateValueSet_);
}
```

Figure 6.5: The exeCASWithEffect function and its dependent functions for General CASWithEffect

If the returned value equals to the old value, which means the descriptor is successfully installed, the *persistStatusAndShareAddr* function is executed to set the new value to the shared variable. Then the succeeded effect is set to the private variable. When updating the private variable, the *isPrivateValueSet_* flag needs to be set true and persisted in the end so that the recovery procedure knows it does not need to update the private variable.

To read values from the affected shared addresses, I provide a read function as shown in Figure 6.6. The *CASWithEffectRead* function checks whether the value in that address is a descriptor. If the value is not a descriptor, the function returns this value. If it is, the *persistStatusAndShareAddr* function is called for the help mechanism. Then, the *CASWithEffectRead* function returns the new value in the descriptor. Normally, to avoid inconsistency, a read operation from persistent memory is always followed by a cache line flush to ensure that the value being read is persisted. This flush-on-read principle [36] causes extra overhead compared to a read operation from volatile memory. In my implementation, there is no need to flush a regular value in a read operation, because the recovery mechanism ensures that the value of the shared variable cannot be lost in a crash. For a private variable, the value is always in the persistent memory before being read because there is only one thread reading and writing that address.

```
uint64_t CASWithEffectRead(uint64_t * shareAddr) {
     uint64_t ret = (uint64_t)(*shareAddr);
     if(ret & DESCRIPTOR_FLG != 0) {
          persistStatusAndShareAddr(ret&(~DESCRIPTOR_FLG));
          ret = (ret&(~DESCRIPTOR_FLG))->nValue_;
     }
     return ret;}
```

Figure 6.6: The CASWithEffectRead function for General CASWithEffect

6.2.3 The Recovery Procedure

A crash can occur anytime during a *CASWithEffect* operation. After restarting from such a crash, a recovery thread is started by the application to regain consistency for all pending operations. Similar to the recovery of PMwCAS, the recovery of CASWithEffect inspects all descriptors from the descriptor pool, and completes or aborts every in-flight operation. After recovery is finished, other threads can begin regular *CASWithEffect* operations.

The recovery thread first goes through every descriptor in the descriptor pool and processes the descriptor whose *isPrivateValueSet_* is not true, which indicates the last operation is not complete. For a descriptor with *Succeeded* status, if the shared address of this descriptor holds the descriptor's pointer, the new value of this descriptor will be assigned to the variable and flushed. Then, recovery will update and flush the private address with the succeeded effect value. For a descriptor with the *Undecided* status, recovery will check whether the shared variable holds this descriptor's pointer. If the descriptor pointer has already been installed in the shared address, recovery will change the status of this descriptor to *Succeeded* first and recover it as a succeeded descriptor. For a descriptor with the *Failed* status, recovery will put the failed effect value to the private variable and flush it when the failed effect is needed. After updating the shared and private variables, recovery changes the *isPrivateValueSet_* field in the descriptor to true and flushes it. After all descriptors are recovered, a thread can get the effect of the CAS operation based on the value of its private variable, which may be updated by the recovery procedure to complete a pending operation.

6.3 Fast CASWithEffect Algorithm

The General CASWithEffect algorithm manages to store the effect of one CAS operation to a private variable and keep the effect from being lost in a crash through its recovery mechanism. Although it only installs a descriptor on the shared variable, the procedure of installing the descriptor first and then replacing it with the actual value causes more overhead than using a single CAS to set the actual value. Additionally, the epoch-based memory reclamation algorithm for recycling descriptors, which will be explained in Chapter 7, incurs overhead when putting finished descriptors back to the memory pool. In Ben-David et al.'s Recoverable CAS algorithm [6], the bookkeeping information, including a thread ID t_i and sequence number s_i, is stored directly in the target variable of CAS alongside the actual value, which avoids the procedure of putting the bookkeeping data into the variable first and then changing it to the actual value. The sequence number s_i belongs to the thread t_i and increases every time when t_i executes a CAS operation successfully. The bookkeeping data of t_i and s_i indicate one specific successful CAS operation. Before executing CAS on a target address, each thread reads the thread ID and sequence number from the target variable first. Then, it updates the status of the operation indicated by the thread ID and sequence number in the target variable, to *Succeeded*. Regarding the descriptor reclamation algorithm, Arbel-Raviv and Brown provide a descriptor reuse algorithm in [3], which does not need recycling descriptors. In the descriptor reuse algorithm, every thread is assigned its own descriptor in the beginning by the application and uses its own descriptor to execute multiple operations on different variables. Similarly to the recoverable CAS algorithm, the descriptor reuse algorithm uses a sequence number to indicate one specific operation executed by a certain thread. Inspired by the two algorithms, I design an algorithm – Fast CASWithEffect, which has the same function as General CASWithEffect but with better performance. In Chapter 9, I will compare the performance of General CASWithEffect and Fast CASWithEffect as shown in Figure 9.1.

6.3.1 The General Idea of Fast CASWithEffect

To distinguish the *CASWithEffect* operation in the Fast CASWithEffect algorithm from the operation in the General CASWithEffect algorithm, I call it *FCASWithEffect*. In the Fast CASWithEffect algorithm, every thread, which is assigned a distinct descriptor in the beginning from the descriptor pool for each shared variable, uses its own descriptor to execute multiple *FCASWithEffect* operations on one particular shared variable. In the Recoverable CAS algorithm [6], a sequence number and a thread ID are stored in the target address alongside the actual value to allow helping threads identify a certain operation. For most lock-free data structures, the value in the target address of a CAS operation is a memory address, such as a queue node pointer in the MS queue. Since modern 64-bit x86 processors implement 48 address bits [36], the higher 16 bits become vacant to store the thread ID and sequence number. If the number of running threads is limited to 64, the maximum number for the sequence number is 1023 (10 bits). The sequence number will wrap around after 1024 operations, which can cause ABA problems explained in Section 3.5.3. For some algorithms, the new values of successful CAS operations on a certain shared variable are different. For example, a thread executes a *FCASWithEffect* to insert a new queue node to a MS queue. The new value of every successful *FCASWithEffect* operation executed by a thread is a newly allocated queue node. The memory management algorithm, which will be explained in Chapter 7, makes sure the ABA problem will not happen. Fast *CASWithEffect* only applies to these algorithms, in which the new value in a CASWithEffect descriptor can indicate one particular operation for helping threads. The lower 54 bits of the shared variable are used to store the actual value and the higher 10 bits are used to store the thread ID.

Suppose a thread T_1 tries to execute a *FCASWithEffect* operation on a shared variable X. Thread T_1 reads a thread ID T_0 and actual value v_0 from X, which shows T_0 successfully updated X with value v_0. Then T_1 helps T_0 by updating the *status_* field of T_0's descriptor to *Succeeded* using CAS. From the thread ID and shared variable address, T_1 can get T_0's descriptor because every thread has only one descriptor for one shared variable. To make sure T_1 updates the correct operation of T_0, v_0 is stored in the the status_ field of the T_0's descriptor alongside the status value. Since the status value only needs 2 bits, the higher 62 bits the *status_* field become vacant to store v. Since Fast CASWithEffect only applies to algorithms in which $v0$ can distinguish one particular operation of T_0 for helping threads, if the new value in the *status_* field of T_0 is not equal to v_0, which means T_0 has completed the operation of v_0 and started another operation, the update will fail. In this way, the helping thread can avoid the possibility of updating a wrong operation to *Succeeded* for another thread.

6.3.2 The Algorithm Details of Fast CASWithEffect

Fast CASWithEffect uses the same descriptor class as in General CASWithEffect, except that the *status_* field includes two pieces of information: the new value and the status value. The pseudo-code of the *FCASWithEffect* function is shown in Figure 6.7, which calls the *genDescriptorForFCAS* function to get the descriptor and the *exeFCASWithEffect* function to execute CAS and store the effect.

```
uint64_t FCASWithEffect(uint64_t* shareAddr, uint64_t* privateAddr
    uint64_t oValue, uint64_t nValue,
    uint64_t sEffect, uint64_t fEffect)
{
    CASWithEffectDescriptor * fd =  genDescriptorForFCAS(shareAddr,
       privateAddr, oValue, nValue, sEffect, fEffect);
    return exeFCASWithEffect(fd);
}
```

Figure 6.7: The FCASWithEffect function for Fast CASWithEffect

In the *genDescriptorForFCAS* function(Figure 6.8), the operation descriptor is obtained from the descriptor pool through the caller thread's ID and the shared variable address. The *status_* field is assigned to a value combining the new value of this operation and *Undecided*. Besides, the *isPrivateValuesSet_* field, which was updated to *true* when the last operation of the caller thread completed, needs to be set to $false$.

The pseudo-code of the *exeFCASWithEffect* function is shown in Figure 6.9, which starts by persisting the descriptor pointer and reading the thread ID and actual value from the shared variable. The executing thread first checks whether the thread ID obtained from the shared variable is itself. If not, another thread's descriptor is obtained and a help procedure is called (lines 356 - 362). If another thread's descriptor is NULL, which means no *FCASWithEffect* operation has been executed on this shared variable, the helping thread

```
CASWithEffectDescriptor * genDescriptorForFCAS(
        uint64_t* shareAddr, uint64_t* privateAddr,
        uint64_t oValue, uint64_t nValue,
        uint64_t sEffect, uint64_t fEffect)
{
    CASWithEffectDescriptor * fd =  getDescriptor(selfThreadId(),
       shareAddr);
    fd->shareAddr_ = shareAddr;
    fd->privateAddr_ = privateAddr;
    fd->oValue_ = oValue;
    fd->nValue_ = nValue;
    fd->sEffect_ = sEffect;
    fd_->isFEffectNeeded_ = true;
    fd_->fEffect_ = fEffect;
    fd_->status_ = <fd_->nValue_, Undecided>;
    fd_->isPrivateValueSet_ = false;

    return fd;
}
```

Figure 6.8: The genDescriptorForFCAS function for Fast CASWithEffect

returns from the help procedure. In General CASWithEffect, a thread only helps when there is a descriptor in the shared variable. In Fast DetectableCAS, to avoid unnecessary help, the helping thread first checks whether the descriptor's *isPrivateValueSet_* field is *true*, which means the helped operation has already been finished. If the value is not *true*, the helping thread flushes the shared variable first. Then, if the old value of the help thread's operation equals to the actual value in the shared variable, the helping thread will try to update the status value to *Succeeded* by comparing the actual value to the corresponding values in the status_ field. After the help procedure completes, the executing thread checks whether the actual value in the shared address matches the old value in the thread's descriptor. If not, it sets the status value of the *status_* field to *Failed*, puts the failed effect to the private variable (if needed) and returns (lines 336 - 338). If the values match, the executing thread continues by executing a CAS, which tries to put a value combining its thread ID and the new value to the shared variable (line 340). According to the result of the CAS operation, the status value in the *status_* field becomes *Succeeded* or *Failed*. Also, a succeeded or failed effect is set to the private variable and flushed using the same function *setEffectToPrivateAddr* as in General DetectableCAS.

The read operation of Fast CASWithEffect reads the thread ID T_1 and actual value from

```
uint64_t exeFCASWithEffect(CASWithEffectDescriptor * fd) {
    flush(fd);
    <oThreadId, actualValue> = *(fd->shareAddr_);

    if(oThreadId != selfThreadId()) {
        CASWithEffectDescriptor * ofd = getDescriptor(oThreadId, fd->
            shareAddr_);
        helpFastDetectableCAS(ofd, fd, actualValue);}

    if(actualValue != fd->oValue_) {
        changeFailStatusAndSetFEffect(fd);
        return actualValue;}

    <rThreadId, rActualValue> = CAS(fd->shareAddr_, <oThreadId,
        actualValue>, <selfThreadId(), fd->nValue_>);
    if(<rThreadId, rActualValue> != <oThreadId, actualValue>)  {
        changeFailStatusAndSetFEffect(fd);
        return rActualValue;}

    flush(fd->shareAddr_);
    fd->status_ = <fd->nValue_, Suceeded>;
    flush(&fd->status_);
    setEffectToPrivateAddr(fd, fd->sEffect_);
    return fd->oValue_;}

void changeFailStatusAndSetFEffect(CASWithEffectDescriptor * fd) {
    fd->status_ = <fd->nValue_, Failed>;
    flush(&fd->status_);
    if(fd->isFEffectNeeded_)setEffectToPrivateAddr(fd, fd->fEffect_);}

void helpFastDetectableCAS(DetectableCASDescriptor * ofd,
    DetectableCASDescriptor * fd, uint64_t actualVal)
{
    if(ofd == NULL || ofd->isPrivateValueSet_) return;
    flush(ofd->shareAddr_);
    if(actualVal != fd->nValue_) return;
    CAS(&ofd->status_, <actualVal,Undecided>, <actualVal,Suceeded>);
    flush(&ofd->status_);}
```

Figure 6.9: The exeFCASWithEffect function and its dependent functions for Fast CASWithEffect

a shared variable as shown in Figure 6.10. To ensure consistency, the operation persists the shared variable after reading its value. To avoid unnecessary persists, the executing thread checks whether T_1's operation is complete by looking up the *isPrivateValueSet_* field int T_1's descriptor. Since the read operation will not change the value in the shared variable, there is no need to help T_1 update its status to *Succeeded*. At last, the read operation returns the actual value in the shared variable.

```
uint64_t FCASWithEffectRead(uint64_t * shareAddr) {
    <oThreadId, actualValue> = *shareAddr;
    CASWithEffectDescriptor * ofd = getDescriptor(oThreadId, shareAddr
        );
    if(ofd != NULL && !ofd->isPrivateValueSet_) {
        flush(ofd->shareAddr_);
    }
    return actualValue;
}
```

Figure 6.10: The FCASWithEffectRead function for Fast CASWithEffect

6.3.3 The Recovery Procedure

Similar to General CASWithEffect, after restarting, a recovery thread is started by the application to recover every pending operation in Fast CASWithEffect. Recovery inspects every descriptor, which was assigned to an executing thread, and processes the descriptor whose *isPrivateValueSet_* is not true, which indicates the last operation is not complete. If the status of the descriptor is *Succeeded*, recovery will put the succeeded effect value to the descriptor's private variable and flush it. If the status is *Undecided*, recovery will check whether the thread ID in the shared variable is the descriptor's owner thread and the actual value in the shared variable is the new value in the descriptor. If the values are equal, which means the last operation successfully updated the shared variable, recovery will put the succeeded effect value to the descriptor's private variable and flush it. If the status is *Failed*, recovery will set the failed effect to the private variable and flush it when needed. After updating the private variable, recover changes the *isPrivateValueSet_* field in the descriptor to true and flush it.

6.4 Case Studies

In this section, I provide some cases in which the CASWithEffect (FCASWithEffect) primitive can be used to build some persistent data structures.

6.4.1 Recoverable CAS

A thread can tell whether its last CAS operation before a crash succeeded in updating the target variable through the Recoverable CAS algorithm (RCAS) in [6]. As mentioned in the beginning of Section 6.3, RCAS uses a sequence number and thread ID to indicate a specific CAS operation. As explained in Section 6.3.1, for 64 threads, the sequence number will wrap around after 1024 operations. If the sequence number wraps around, an ABA problem may happen when a thread modifies the status of another thread's operation. To avoid using the sequence number, the General CASWithEffect algorithm can be applied to implement RCAS. The procedure of the implementation is as follows. A thread sets an initial value into its private variable and persists it. Then the thread calls the CASWithEffect function to perform a CAS operation with a *true* value as the succeeded effect value. When a crash happens during the execution, after recovery, the same thread reads the value from its private variable. If the value equals to *true*, the thread's last CAS operation successfully updated the target address.

6.4.2 Fetch-And-Store-And-Store

Golab and Hendler's recoverable mutex [14] uses the Fetch-And-Store-And-Store (FASAS) [32] primitive, which atomically stores the returned value of FAS into a private variable so that the returned value will not be lost when a crash happens. The FASAS primitive can

```
uint64_t FASAS(uint64_t * sAddr, uint64_t nVal, uint64_t * pAddr) {
    while(true) {
        uint64_t oVal = CASWithEffectRead(sAddr);
        if(CASWithEffect(sAddr, pAddr, oVal, nVal, oval) == oval) {
            return oVal;
        }}}
```

Figure 6.11: Implementation of FAS using CASWithEffect

be implemented using CASWithEffect as shown in Figure 6.11. The FASAS begins with reading the current value from the shared address through *CASWithEffectRead*. Then it executes *CASWithEffect* using the current value both as the old value and the succeeded effect value. If the returned value of *CASWithEffect* equals to the current value, which means the CAS operation successfully puts the new value into the shared variable and the current value is set to the private variable, the FASAS returns with the current value. If the returned value does not equal to the current value, the FASAS retries the whole procedure by first reading another current value from the shared variable.

6.4.3 CASWithEffect Queue

In Section 5.3, I described Detectable Queue, which stores the effects of queue operations to a thread's private variables. In this section, I present how to simplify the algorithm of Detectable Queue by using the CASWithEffect primitive. The queue using CASWithEffect, called CASWithEffect Queue, provides the same detectability as Detectable Queue. However, CASWithEffect Queue does not provide undetectable operations: *enqueue* and *dequeue*. The reason is that the node pointers are processed by CASWithEffect with descriptors being installed on them. A regular CAS operation cannot deal with an address holding a descriptor pointer.

```
class CASWithEffectQNode {
    uint64_t * pData_;
    CASWithEffectQNode * next_ = NULL;};

class CASWithEffectQueue {
    CASWithEffectQNode * phead_;
    CASWithEffectQNode * ptail_;
    bool isStrongDetectability_;

    void prepareEnqueue(CASWithEffectQNode * enqNode);
    void detectableEnqueue();
    void retrieveEnqueue(uint64_t* e0, uint64_t* e1, uint64_t* a);
    void prepareDequeue();
    CASWithEffectQNode *  detectableDequeue();
    void retrieveDequeue(uint64_t* e0, uint64_t* e1);};
```

Figure 6.12: CASWithEffect Queue classes

The classes of CASWithEffectQueue are defined as shown in in Figure 6.12. The CASWithEffectQNode class has a data pointer and a next pointer. The CASWithEffec-

```
void CASWithEffectQueue::prepareEnqueue(CASWithEffectQNode * node) {
    uint64_t * pAddr = getPrivateAddrByOpName(ENQUEUE);
    flush(node);
    *pAddr = (uint64_t)node;
    flush(pAddr);}

void CASWithEffectQueue::retrieveEnqueue(uint64_t* e0, uint64_t* e1,
    uint64_t* a) {
    uint64_t * pAddr = getPrivateAddrByOpName(ENQUEUE);
    if(*pAddr == MEM_INIT_VAL) {
        *e0 = NoExecutedOp;}
    else if(*pAddr & OPCOMPLETE_FLG != 0) {
        *e0 = HasEffect;
        *e1 = AckResponse;
        if(isStrongDetectability_) *a =  *pAddr | ~OPCOMPLETE_FLG;}
    else {
        *e0 = NoEffect;
        if(isStrongDetectability_) *a =  *pAddr;}
}
```

Figure 6.13: The prepareEnqueue function and retrieveEnqueue function of CASWithEffect Queue

```
void CASWithEffectQueue::detectableEnqueue() {
    uint64_t * pAddr = getPrivateAddrByOpName(ENQUEUE);
    CASWithEffectQNode * node = (CASWithEffectQNode *)(*pAddr);
    while(true) {
        CASWithEffectQNode * last = *ptail_;
        CASWithEffectQNode * next = CASWithEffectRead(&(last->next_));

        if(last == (ptail_)) {
            if(next == NULL) {
                if(CASWithEffect(&last->next_,pAddr,NULL,node,node|
                    OPCOMPLETE_FLG) == NULL){
                    CAS(&ptail_, last, newNode);
                    return;}}
            else {
                flush(&last->next_);
                CAS(&ptail_, last, next);}}}}
```

Figure 6.14: The detectableEnqueue function of CASWithEffect Queue

```
void CASWithEffectQueue::prepareDequeue() {
    uint64_t * pAddr = getPrivateAddrByOpName(DEQUEUE);
    *pAddr = OP_INIT_VAL;
    flush(pAddr);
}
void CASWithEffectQueue::retrieveDequeue(uint64_t* e0, uint64_t* e1) {
    uint64_t * pAddr = getPrivateAddrByOpName(DEQUEUE);
    if(*pAddr == MEM_INIT_VAL) {
        *e0 = NoExecutedOp;}
    else if(*pAddr == OP_INIT_VAL) {
        *e0 = NoEffect;
    else {
        *e0 = HasEffect;
        *e1 =  * pAddr; }}
```

Figure 6.15: The prepareDequeue function and retrieveDequeue function of CASWithEffect Queue

```
CASWithEffectQNode * CASWithEffectQueue::detectableDequeue() {
    uint64_t * pAddr = getPrivateAddrByOpName(DEQUEUE);
    while(true) {
        CASWithEffectQNode * first = CASWithEffectRead(&phead_);
        CASWithEffectQNode * next=CASWithEffectRead(&(first->next_));
        CASWithEffectQNode * last = ptail_;
        if(first == *phead_) {
            if(first == last) {
                if(next == NULL) {
                    if(isStrongDetectability_) {
                        *pAddr = NULL;
                        flush(pAddr);
                    }
                    return NULL;}
                flush(&last->next_);
                CAS(&ptail_,last, next);}
            else {
                if(CASWithEffect(&phead_, pAddr, first, next, first) =
                    = first) return first;
            }}}}
```

Figure 6.16: The detectableDequeue function of CASWithEffect Queue

tQueue class has a head pointer, a tail pointer and *isStrongDetectability_* as in the DetectableQueue class. Besides, it provides six auxiliary functions required by the sequential specification of $D\{queue\}$. However, it cannot provide *enqueue* and *dequeue* functions.

The pseudo code of the *prepareEnqueue* function and *retrieveEnqueue* function are shown in Figure 6.13. They are the same as the two functions in the Detectable Queue (Section 5.3). In the *prepareEnqueue* function, a new queue node is put into the caller thread's private variable and persisted. In the *retrieveEnqueue* function, the caller thread reads the value from its private variable and gets the enqueue operation's effect according to the value.

The *detectableEnqueue* function uses CASWithEffect to update the next pointer of the queue's last node as shown in Figure 6.14. In the beginning, a new queue node is fetched from the caller thread's private variable to be enqueued. The operation first reads the last node from the tail pointer and the next pointer of the last node. Because another thread may be updating the *next_* field of the last node using *CASWithEffect*, the *next_* field must be read through *CASWithEffectRead* (line 415). If the next pointer is NULL, the operation executes *CASWithEffect* to update the next pointer to the new queue node and save the succeeded effect value to the private variable (line 418). As in the Detectable Queue, the succeeded effect value is a combined value of the data pointer in the new queue node and the *OPCOMPLETE_FLG* flag. If the *CASWithEffect* operation succeeds, the tail pointer is moved to the new node and the operation returns. If the *CASWithEffect* fails, the operation will retry the procedure from reading the last node of the queue.

The pseudo code of *prepareDequeue* function and *retrieveDequeue* function are shown in Figure 6.15. They are the same as the two functions in the Detectable Queue (Section 5.3). In the *prepareDequeue* function, *OP_INIT_VAL* is set to the private variable and persisted. In the *retrieveDequeue* function, the effect is based on the value in the private variable.

The *detectableDequeue* function starts by getting the caller thread's private variable as shown in Figure 6.16. The operation reads the first node from the head pointer and the next pointer of the first node using *CASWithEffectRead* because both addresses are accessed by *CASWithEffect*. It also reads the last node from the tail pointer. The operation first checks whether the queue is empty by comparing the first node to the last. If the queue is empty, the operation returns a NULL pointer and set the NULL pointer to the private variable for strong detectability (line 446 - 451). If the queue is not empty, a *CASWithEffect* operation is executed with the head pointer as the shared variable, next node as the new value, and first node as both the old value and the succeeded effect (line 455). If the *CASWithEffect* operation succeeds, the first node is returned and put into the caller thread's private variable as in Detectable Queue. If it fails, the operation will retry the procedure from reading the first node of the queue.

The CASWithEffect Queue does not need a recovery procedure because the recovery of CASWithEffect guarantees the consistency of the queue and correct values in the private variables. After the recovery of CASWithEffect, working threads can call retrieveEnqueue and retrieveDequeue functions to get the effects of their previous enqueue and dequeue operations.

In the Detectable Queue algorithm, the effects are manually stored into the private variables and flushed to the persistent memory. The algorithm also depends on a specific recovery procedure to recover the queue and restore effect values when a crash happens. In the CASWithEfect Queue algorithm, the effect value is automatically written to the private variable and persisted when the operation takes effect by using the CASWithEffect primitive. Also, there is no need of a specific recovery for the queue. Comparing the code lines of the two algorithms, the CASWithEffect Queue is much simpler, especially for the *detectableDequeue* function. The *detectableDequeue* function in the Detectable Queue needs an additional field (deqThreadID_) to mark the returned node. The *detectableDequeue* function in the CasWithEffect Queue is similar to the *dequeue* function in MS Queue. It replaces CAS with CASWithEffect and reading with CASWithEffectRead. To obtain strong detectability, the *detectableDequeue* function also adds one flush line. Overall, it is much easier to turn a MS queue to a persistent and detectable queue by using the CASWithEffect primitive than using the CAS operation.

Chapter 7

Memory Management

In this chapter, I will describe the memory management for the Detectable Queue algorithm presented in Chapter 5 and the CASWithEffect algorithms presented in Chapter 6.

7.1 Memory Allocation

The memory manage mechanism I use to recycle the queue nodes of the Detectable Queue and the descriptors of the General CASWithEffect algorithm is the same epoch-based memory management algorithm explained in [36]. Both a queue node and a descriptor are considered a memory object. All memory objects of each data type are pre-calculated and pre-allocated as a memory pool (queue node pool or descriptor pool) from the persistent memory by the application during initialization. Then, every memory object in the memory pool is assigned to a thread. Each thread maintains its own free memory object list of each data type, such as a queue node free list or a descriptor free list, and a newly allocated object is appended to the free list of its type of a thread. Each thread only obtains the memory objects from its own free lists to execute operations. If a thread cannot get a memory object because there are no memory objects left in its free list, the thread will try to reclaim the memory objects it released before. The release and reclaim of memory objects will be explained in the next paragraphs and section 7.2.

Regarding the General CASWithEffect algorithm, all the descriptors are allocated from the persistent memory, as a descriptor pool, at one time before any CASWithEffect operation starts based on the descriptor pool size which is passed to the application as a program argument. Then, the allocated descriptors from the descriptor pool are assigned to each thread. Every thread obtains one descriptor from its own free descriptor list to execute a CASWithEffect operation and releases the descriptor when the operation is finished. The released descriptor will be reclaimed and reused based on the epoch-based memory reclamation algorithm, which will be discussed in the next session. In the Detectable Queue algorithm, similarly to General CASWithEffect, all the queue nodes are allocated during initialization from the persistent memory as a queue node pool based on the node size and the nodes from this node pool are assigned to each thread. For the enqueue operation, a thread obtains one queue node from its free queue node list and appends this node to the queue. For the dequeue operation, a thread gets the first node from the queue-node linked list of a non-empty queue and releases this queue node, which will be reclaimed and appended into the dequeue thread's free queue node list using the same epoch-based memory reclamation algorithm.

The memory management of Fast CASWithEffect is different from Detectable Queue or General CASWithEffect. A CASWithEffect descriptor is allocated from the persistent memory for every thread and every shared variable. Each thread will use its own descriptor to execute multiple Fast CASWithEffect operations on one particular shared variable. For example, for thread t and shared variable s, a descriptor, denoted $desc[t][s]$, is allocated to t during the initialization of the application. Then, thread t will keep using $desc[t][s]$ to execute operations of Fast CASWithEffect on s. Therefore, there is no need for memory reclamation for descriptors in the Fast CASWithEffect algorithm.

7.2 Memory Reclamation

For the Detectable Queue or General CASWithEffect algorithm, a thread will release a memory object, such as a queue node or a descriptor, when an operation (dequeue or CASWithEffect) finishes. However, the released object cannot be appended to the releasing thread's free memory object list because other threads may dereference a pointer to the released object. Therefore, every thread maintains a garbage memory object list of each data type and puts a released memory object to its own garbage list, such as a garbage descriptor list or a garbage queue node list. A thread will remove a released memory object from its garbage list and append this object to its free list when there is no other thread dereferencing this object's pointer. I use an epoch-based memory reclamation algorithm [36] to safely reclaim released memory objects. In the epoch-based algorithm, there is a global epoch value for each data type (the queue node or descriptor), called the current epoch value. The current epoch value of a certain data type has an initial value, which is 1, and will be incremented by a thread which has fewer free memory objects of this type than some threshold value. For example, after some enqueue and dequeue operations, if a thread finds the size of its free queue node list is less than 10000 (the threshold value), it will increment this current epoch value to 2. The current epoch value is a shared variable which can be modified by multiple threads. Each thread uses an atomic operation to increment the current value. Besides the global current epoch value, every thread maintains its own epoch value for each data type. When a thread T begins to execute some operations which will use a memory object of type D, this thread sets the current global epoch value E of D to its own epoch value of E_T. When T finishes these operations, it sets E_T to $\perp$. A released object r of type D also has its own epoch value, called the removal epoch value E_r. When r is released by some thread, the releasing thread sets the current global epoch value of D to r's removal epoch E_r. A released memory object r can only be reclaimed when each thread's epoch value is either $\perp$ or greater then E_r, which means that no thread can dereference a pointer to this object. For example, suppose that the current epoch value of the queue node type is 1. Both thread T_1 and T_2 begin a dequeue operation and set their epoch values to the current epoch value 1. Then, T_1 dequeues a node N_1 and releases this node, whose removal epoch value is set to 1. T_1 finishes its dequeue operation and sets its epoch value to $\perp$. N_1 cannot be released because its removal epoch value is equal to T_2's epoch value, which means that T_2 may dereference the pointer to N_1. When T_2 finishes its dequeue operation, T_2 sets its epoch value to $\perp$. Now N_1 can be safely reclaimed by T_1 because T_2 and T_1's epoch values are $\perp$, which means it is impossible for either thread to dereference N_1's pointer.

7.3 Memory Recovery

As explained in Chapter 6, the recovery of CASWithEffect goes through every descriptor from the descriptor pool, and may roll back or roll forward a CASWithEffect operation based on the status of each descriptor as explained in Section 7.1. After the recovery of CASWithEffect, the old descriptor pool can be cleared. Then a new descriptor pool is generated from the persistent memory and initialized to replace the current one. Also, the global current epoch value of the descriptor is set 1.

In the Detectable Queue algorithm, as shown in Section 5.3, the recovery of the queue, which is started after a crash by a single recovery thread T_r, makes sure the queue is in a consistent status and the values in the private variables correctly represent the status of each thread's last operation. After the recovery of the queue, T_r will start the recovery of queue node memory. T_r will go through every queue node in the node memory pool from the persistent memory. If a queue node n is the linked list of the queue, n cannot be appended to any thread's free node list. If n is in one thread's private variable or n is the next pointer of the queue node in one thread's dequeuing private variable, n cannot be appended to any thread's free node list. Otherwise, n can be appended to a thread's free node list with its original data being cleared. Besides, if n is not in the queue or in any thread's private variables, and n is the next pointer of the node in a dequeuing private variable belonging to thread T_d, n will be put to T_d's garbage list with its removal epoch value being set to 1. The reason for this is that n is the actual dequeued node for T_d, which cannot be put to any thread's free node list. Then, T_r will set 1 to the global current epoch value and each working thread's epoch value to make sure that any node which is possible to be dereferened by a working thread will not be recycled. Then, working threads can get the effects of their last invoked enqueue or dequeue operations and begin their operations.

Chapter 8

Correctness

In this chapter, I will present the analysis of safety and liveness for the Detectable Queue algorithm (Section 5.3).

8.1 Correctness Properties of Detectable Queue

8.1.1 Proof of Safety

In this section, I will prove that Detectable Queue satisfies strict linearizability (Definition 3.2).

First, I define linearization points for each operation in Detectable Queue (Figure 5.1): *prepareEnqueue*, *detectableEnqueue*, *enqueue*, *retrieveEnqueue*, *prepareDequeue*, *detectableDequeue*, *dequeue* and *retrieveDequeue*.

Definition 8.1. The linearization point of the *prepareEnqueue* operation is at line 81.

Definition 8.2. The linearization points of the *detectableEnqueue* operation and the *enqueue* operation are defined as follows:

1. If the CAS operation (line 91 and 130) succeeds and the new value of the next pointer of the last node has been flushed to the persistent memory at line 92, 98, 131, 135, 181 and 210, the linearization points of the *detectableEnqueue* and *enqueue* operations are at line 91 and 130 respectively. Otherwise, there are no linearization points.

Definition 8.3. The linearization point of the *retrieveEnqueue* operation is at line 121 where the operation returns.

Definition 8.4. The linearization point of the *prepareDequeue* operation is at line 140.

Definition 8.5. The linearization points of the *detectableDequeue* and *dequeue* operation are defined as follows:

1. If the CAS operation (line 188 and 213) succeeds and the value of the *deqThreadID_* field has been flushed into the persistent memory at line 189, 195, 214 or 220, the linearization points of the *detectableDequeue* operation and *dequeue* operation are at line 188 and 213 respectively.

2. If the *detectableDequeue* operation flushes the value in the private variable at line 177 (strong detectability) or returns NULL at line 179 (weak detectability), the linearization point of the *detectableDequeue* operation is at line 171.

3. If the *dequeue* operation return NULL at line 209, the linearization point of the *dequeue* operation is at line 205.

4. Otherwise, there is no linearization point of *detectableDequeue* or *dequeue*.

Definition 8.6. The linearization point of the *retrieveDequeue* operation is at line 164 where the operation returns.

To prove linearizability of Detectable Queue, for each history H of Detectable Queue, which is composed of enqueuing operations, dequeuing operations and crash events, I define a candidate linearization history $L(H)$ which is constructed as follows.

1. For every pending operation in H, if the linearization point exists according to Definition 8.1 to 8.6, add the matching response before the next crash event (if any) or in the end if no such crash event exists. The definitions of the linearization points show that every operation takes effect before a crash.

   The matching response is added as follows.

   (a) For the *detectableEnqueue*, *enqueue*, *prepareEnqueue* and *prepareDequeue* operations, since the returned type is void, the matching response is an acknowledgement response.

(b) For the *retrieveEnqueue* and *retrieveDequeue* operations, since the linearization point is where the operation returns, the response is already known when the linearization point is reached, and that exact response is used in the construction of L.

(c) For the *detectableDequeue* operation, if the operation reaches line 177 (strong detectability) or line 179 (weak detectability), the matching response event returns NULL, which is the only possible returned value.

(d) For the *dequeue* operation, if the operation reaches line 209, the matching response event returns NULL, which is the value returned at this line.

(e) For the *detectableDequeue* operation, if the operation linearizes at line 188, the matching response returns the next pointer of the first node, which is the only possible returned value computed earlier at line 170.

(f) For the *dequeue* operation, if the operation linearizes at line 213, the matching response returns the next pointer of the first node, which is the only possible returned value computed earlier at line 204.

2. Remove pending operations which do not have linearization points and crash events in H.

3. Arrange the remaining operations in H in the chronological order based on the timestamp of the linearization point of each operation, which makes a sequential history L.

   According to the definitions of linearization points, all linearization points happen between an operation's invocation and response. Furthermore, the linearization points of an operation executed by process p is an instruction taken by p inside that operation. These two observations imply that all linearization points are distinct.

Now, I will prove that Detectable Queue implements D{FIFO Queue} (Section 4.3) based on the model explained in Section 3.5.2.

The representation value r of Detectable Queue contains the cache and persistent value of a linked node list, a head pointer and a tail pointer of the linked node list (Figure 5.1). Besides these values, r also includes the cache and persistent values of private variables for every executing process. For process p executing enqueuing, the value of the private variable is denoted $r.pAddr[p][enq]$ and for p executing dequeuing is denoted $r.pAddr[p][deq]$ in this proof.

The rep invariant $I(r)$ is shown in Figure 8.1. I define the function *getHeadPointer* and *getTailPointer* to get the head and tail pointer of the linked list respectively. Both the head and tail pointers must not be NULL pointers. I define the function *isTailReachable-FromHead* to return true if the node referenced by the tail pointer is reachable from the node referenced by the head pointer through the next pointer of each node and return false otherwise. The *getLastNode* function is defined to get the last node whose next pointer is a NULL pointer by iterating over the linked list from the head pointer through the next pointer of each node. If there is no such node, the *getLastNode* function returns a NULL pointer. In the dequeuing operation of Detectable Queue (Section 5.3.2), a queue node is marked as removed when a dequeuing process set its process identifier to the *deqThreadID_* field of the queue node. I define the *areMarkedRemovedNodesConsecutive* function to return true if the marked removed nodes, whose *deqThreadID_* field is not -1, are consecutive nodes from the head pointer and false otherwise. Also, the node referenced by the tail pointer cannot be a marked removed node.

$$\begin{gathered}
I(r) = getHeadPointer(r) \neq NULL \\
\wedge getTailPointer(r) \neq NULL \\
\wedge isTailReachableFromHead(r) \\
\wedge getLastNode(r) \neq NULL \\
\wedge areMarkedRemovedNodesConsecutive(r) \\
\wedge getPointedNode(r.ptail\_).deqThreadID\_ = -1
\end{gathered}$$

Figure 8.1: The rep invariant $I(r)$ of Detectable Queue.

The abstract value of D{FIFO Queue} is denoted $d\{q\}$. According to the sequential specification of D{FIFO Queue} defined in Section 4.3, $d\{q\}$ contains the value of FIFO Queue, denoted q. Also, for an enqueuing operation executed by process p, $d\{q\}$ contains a status value $S[p][enq]$ and an argument value $ARG[p][enq]$ while for a dequeuing operation executed by p, $d\{q\}$ contains a status value $S[p][deq]$ and a returned value $R[p][deq]$.

The abstraction function $A(r)$, which maps r to $d\{q\}$, is shown in Figure 8.2. There are two functions in $A(r)$: the $seq(q)$ function and the $queueNodeSeq(r)$ function. The $seq(q)$ function is defined to return the sequence of queue elements of q. The $queueNodeSeq(r)$ function for r is defined as follows.

1. The function traverses the linked list from the head pointer until finding a node n such that n has not been flushed at line 189, 195, 214 or 220, or the *deqThreadID_* field of n is -1. The next pointer of n is the beginning node, denoted n_1.

2. If n_1 is not a NULL pointer, the function continues traversing the linked list and ends at the node which has not been flushed at line 92, 98, 131, 135, 181 or 210, or the node whose next pointer is NULL. This ending node is denoted n_2.

3. If n_1 is a NULL pointer, $queueNodeSeq(r)$ will return an empty set.

4. If n_1 is not a NULL pointer, $queueNodeSeq(r)$ will return the sequence of nodes discovered from n_1 to n_2 (inclusive). The order of the nodes in the linked list of r is preserved in $queueNodeSeq(r)$.

To map r to $S[p][enq]$ and $ARG[p][enq]$, the *getEnqStatusFromPAddr* function and *getEnqArgFromPAddr* function are defined for r as follows.

Let v denote the value of *r.pAddr[p][enq]* in the persistent memory.

1. If v is a node pointer with flag *OPCOMPLETE_FLG* or if p's program counter is between line 92 and 94 inclusive of a *detectableEnqueue* operation that has already taken effect, *getEnqStatusFromPAddr* returns *END_DECT_OP* and *getEnqArgFromPAddr* returns v without flag *OPCOMPLETE_FLG*.

2. Else if v is *MEM_INIT_VAL*, both *getEnqStatusFromPAddr* and *getEnqArgFromPAddr* return $\perp$.

3. Otherwise, the function *getEnqStatusFromPAddr* returns *BEGIN_DECT_OP* and the function *getEnqArgFromPAddr* returns v.

To map r to $S[p][deq]$ and $R[p][deq]$, *getDeqStatusFromPAddr* and *getDeqRetValFromPAddr* are defined for r as follows.

Let v denote the value of *r.pAddr[p][deq]* in the persistent memory.

1. If v is *MEM_INIT_VAL*, both *getDeqStatusFromPAddr* and *getDeqRetFromPAddr* return $\perp$;

2. Else if v is a queue node (not a NULL pointer) whose *deqThreadID_* in the persistent memory is not equal to p, *getDeqStatusFromPAddr* returns *BEGIN_DECT_OP* and *getDeqRetValFromPAddr* returns $\perp$.

3. Else if v is a queue node (not a NULL pointer) whose *deqThreadID_* in the persistent memory is equal to p, *getDeqStatusFromPAddr* returns *END_DECT_OP* and *getDeqRetValFromPAddr* returns the next pointer of the queue node.

4. Else if v is a NULL pointer, *getDeqStatusFromPAddr* returns *END_DECT_OP* and *getDeqRetValFromPAddr* return NULL.

5. Otherwise, *getDeqStatusFromPAddr* returns *BEGIN_DECT_OP* and *getDeqRetVal-FromPAddr* returns $\perp$.

$$\begin{gathered} A(r) = \{d\{q\} | seq(q) = queueNodeSeq(r) \\ \wedge d\{q\}.S[p][enq] = getEnqStatusFromPAddr(r.pAddr[p][enq]) \\ \wedge d\{q\}.ARG[p][enq] = getEnqArgFromPAddr(r.pAddr[p][enq]) \\ \wedge d\{q\}.S[p][deq] = getDeqStatusFromPAddr(r.pAddr[p][deq]) \\ \wedge d\{q\}.R[p][deq] = getDeqRetValFromPAddr(r.pAddr[p][deq])\} \end{gathered}$$

Figure 8.2: The abstraction function $A(r)$ of Detectable Queue to D{FIFO Queue}.

Lemma 8.1. For each finite history H of Detectable Queue, $s(H)$ (Figure 3.5.3) holds; for the candidate linearization history L constructed from H, L is legal and $A(r) = SLin(L)$.

Proof. I proceed by induction on the length of the history H. IH is defined as a shorthand for "induction hypothesis".

Basis: $|H| = 0$. Since H is empty, L is the same as H. At initialization, the head pointer and tail pointer of Detectable Queue r point to a sentinel node whose next pointer is NULL. This sentinel node is also the last node of the linked list. The initial value of $deqThreadID\_$ of the sentinel node is -1. Therefore, $I(r)$ holds. For the $queueNodeSet(r)$ function in $A(r)$, since the next pointer of the sentinel node is NULL, it returns an empty sequence, which is the initial value of a FIFO queue. All the other four functions in $A(r)$, which map r to the values of the status, argument and returned-value variables of *DetectableT*, return $\perp$ since the values in the private variable are *MEM_INIT_VAL*. According to Section 4.3, the initial values for the status, argument and returned values are $\perp$. Therefore, the sole element of $A(r)$ is a strict linearized value of L. The statement $s(H)$ holds and $A(r)$ is equal to $SLin(L)$.

Induction step: For any $i \in \mathbb{N}$, suppose that $s(H)$ holds when $|H| = i$, and consider $|H'| = i + 1$. H' is a history including H and another event e as the $i + 1$ event. The representation value of H' is denoted r'. Accordingly, the candidate linearization history of H' is denoted L'.

The event e must be an event of Detectable Queue r, an event of $D\{FIFO\ Queue\}$ or a crash event.

If e is an event of Detectable Queue, according to the model (Section 3.5.2) for representation operations, there is only one event for each operation. Then, the event e must be generated by a certain line of the pseudo code of Detectable Queue. The statement $s(H')$ holds directly by IH if $r = r'$ and $L = L'$. Suppose e is executed by process p. I perform the analysis by each code line which changes r including the head pointer, tail pointer, linked list and private variables of the executing process p.

If a crash event happens after an operation of Detectable Queue takes effect, I will prove that $s(H')$ holds after recovery.

Case A: e is an event of Detectable Queue.

Line 80: At line 80, a new queue node n is set into the enqueuing private variable of p. Based on Definition 8.1, the *prepareEnqueue* operation does not take effect at this line, which means there is no change of L' from L or change of $SLin(L')$ from $SLin(L)$. Since the value in the private variable has not been flushed, there is no change of $A(r')$ from $A(r)$. Also, because this step does not affect the representation invariant, $I(r')$ follows from $I(r)$. Therefore, the statement $s(H')$ follows directly from $s(H)$, which means that $s(H')$ holds and $A(r')$ is equal to $SLin(L')$.

Line 81: At line 81, the new queue node n in the enqueuing private variable of p is flushed. Based on Definition 8.1, the *prepareEnqueue* operation takes effect at this line. Then, L' is an extension of L by one *prepareEnqueue* operation with an argument n. Since *prepareEnqueue* has a trivial response (void response), the linearizability of L' follows from the linearizability of L. According to Axiom 4.7 of $D\{FIFO\ Queue\}$, $SLin(L')$ changes from $SLin(L)$ with *BEGIN_DECT_OP* as the status value and n as the argument value for p. Since the value in the enqueuing private variable is flushed, $A(r')$ changes from $A(r)$ with *BEGIN_DECT_OP* being returned as the status value and n being returned as the argument value for p. The change of $SLin(L')$ is the same as the change of $A(r')$. Therefore, $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. Since this step does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds. Then the statement $s(H')$ holds.

Line 91 and 130: At line 91 and 130, p tries to use CAS to put a new node n to the next pointer of the last node nl, which is read from the tail pointer at line 87 and 126, when the next pointer of nl is NULL. According to Definition 8.2, the enqueuing operation takes effect at line 91 and 130 if the CAS operation at line 91 and 130 succeeds and the new value of the next pointer of nl has been flushed to the persistent memory. Even if the CAS operation at line 91 and 130 succeeds, the new value of the next pointer of nl has not been flushed yet. Therefore, there is no change of L' from L or change of $SLin(L')$ from $SLin(L)$.

If the CAS operation fails, the next pointer of nl has not been modified and the statement $s(H')$ follows directly from $s(H)$. If the CAS operation succeeds, r' changes with n, whose next pointer is NULL, being appended to the linked list. Since the value in the next pointer of nl has not been flushed, the function *queueNodeSeq* ends at nl and n will not be returned in the node sequence. Since the node n is not included in the returned node sequence, which means n has not been enqueued, the returned value of *queueNodeSeq* for $A(r)$ and $A(r')$ is the same.

The *getEnqStatusFromPAddr* of $A(r')$ returns *BEG_DECT_OP* as the status value and *getEnqArgFromPAddr* of $A(r')$ returns n as the argument value. The two returned values are the same as the status value and argument value after line 80 being executed which was analyzed above. Therefore, there is no change of $A(r')$ from $A(r)$. From $A(r)$ being equal to $SLin(L)$, $A(r')$ is equal to $SLin(L')$. For $I(r')$, this change of r only affects the *getLastNode* function, which will return n. Because n is not NULL, $I(r')$ holds. Then the statement $s(H')$ holds.

Line 92, 98, 131, 135, 181 and 210: At line 91 and 130, process $p1$ successfully puts a new node n to the next pointer of the last node nl (defined at line 87, 126, 171 205) by using CAS, and at line 92, 98, 131, 135, 181 or 210, process p flushes the value in the next pointer of nl. For the execution of line 92 and 131, p is equal to $p1$.

Since this flush step does not affect the representation invariant, $I(r')$ follows from $I(r)$. Based on Definition 8.2, the enqueuing operation of $p1$ takes effect at line 91 and 130 if the CAS operation at line 91 and 130 succeeds, and the value in the next pointer of nl is flushed at line 92, 98, 131, 135, 181 or 210. Then, L' has one more enqueuing operation (with the argument n), denoted *op*, executed by $p1$ than L, which takes effect at line 91 or 130. The enqueuing operation has a trivial response (void response).

After process $p1$ successfully puts the new node n to the next pointer of nl at line 91 and 130, the value of the next pointer is flushed by $p1$ and helping processes at line 92, 98, 131, 135, 181 and 210. Then, the tail pointer is moved to n by $p1$ and helping processes at line 95, 99, 132, 136, 182 and 211 using CAS. Other enqueuing processes will read the last node from the tail pointer at line 87 and 126, then they will try to put their nodes to the next pointer of the last node at line 91 and 130 using CAS. Only after the tail pointer is moved to n, which means *op* has already taken effect, it is possible for another enqueuing process to successfully put its queue node to the next pointer of n. Therefore, for the history L', *op* must be the last enqueuing operation.

The node referenced by the head pointer is denoted nf. After process $p1$ successfully put the new node n to the next pointer of *nl* at line 91 and 130, for a dequeuing process *pd*, if *nl* and nf are the same node, since the next pointer of *nl* is not a NULL pointer, *pd* will help *op* take effect first by flushing the value of the next pointer of *nl* at line 181 and 210. Then *pd* will move the tail pointer to n at line 182 and 211 using CAS and try to get a dequeued node again by reading the value from the tail pointer at line 171 and 205. Therefore, for the history L', *op* does not precede any dequeuing operation that returns a NULL pointer at line 176 and 209. If nf and *nl* are different nodes, *pd* will try to get a dequeued node from nf, which can not be n. Therefore, for the history L', *op* does not precede any dequeuing operation that returns n. The linearizability of L' follows from the linearizability of L.

Since *op* with an argument n is the last enqueuing operation in L' and n cannot be dequeued within L', according to Axiom 3.1, $SLin(L')$ changes from $SLin(L)$ with a queue element sequence including n as the last node. According to Axiom 4.8, for *detectableEnqueue*, $SLin(L')$ also changes from $SLin(L)$ with *END_DECT_OP* as $p1$'s status value and n as $p1$'s argument.

The change of $A(r')$ from $A(r)$ is as follows. Because the next pointer of *nl* has been flushed at line 92, 98, 131, 135, 181 or 210, the traversal of the linked list in the function $queueNodeSeq$ of $A(r')$ passes *nl*. Since the next pointer of n has not been flushed or is a NULL pointer, the traversal ends at n, which is the ending node. Therefore, $queueNodeSeq$ returns a sequence of nodes ending with n.

For *detectableEnqueue*, after p executing line 92, 98, 131, 135, 181 or 210, the program counter of $p1$ is possibly at line 92 or 93. Since the *detectableEnqueue* operation executed by $p1$ has already taken effect, *getEnqStatusFromPAddr* of $A(r')$ returns *END_DECT_OP* as the status value for $p1$, and *getEnqArgFromPAddr* of $A(r')$ returns n as the argument value for $p1$. Therefore, the change of $SLin(L')$ is the same as the change of $A(r')$ and $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. The statement $s(H')$ holds.

Line 93 and 94: At line 93, p sets the *OPCOMPLETE_FLG* flag to the value in its enqueuing private variable. Since the *detectableEnqueue* operation executed by p has already taken effect, there is no change of L' from L or $SLin(L')$ from $SLin(L)$. Since the program counter of p is at line 94 after line 93 and 94 being executed, and *detectableEnqueue* has already taken effect, *getEnqStatusFromPAddr* returns *END_DECT_OP* as p's status value, and *getEnqArgFromPAddr* returning n as p's argument value. Therefore, there is no change of $A(r')$ from $A(r)$. From $A(r)$ being equal to $SLin(L)$, $A(r')$ is equal to $SLin(L')$. Since this step does not affect the representation invariant, $I(r')$ follows from $I(r)$. Then the statement $s(H')$ holds.

At line 94, p flushes the value in the private variable, which has the *OPCOMPLETE_FLG* flag. The analysis for this line is similar to line 93. After this line being executed, the value in the private variable is in the persistent memory. Then *getEnqStatusFromPAddr* returns *END_DECT_OP* as p's status value, and *getEnqArgFromPAddr* returns n as p's argument value. Therefore, there is no change of $A(r')$ from $A(r)$. Since this step does not affect the representation invariant, $I(r')$ follows from $I(r)$. Therefore, the statement $s(H')$ follows directly from $s(H)$.

Line 95, 99, 132, 136, 182 and 211: At these lines, p tries to use CAS to move the tail pointer to the new queue node n (defined at line 85 and 123) which was appended to the linked list. Since the enqueuing operation has taken effect before this line, there is no change of L' from L or $SLin(L')$ from $SLin(L)$. If this CAS operation fails, the tail pointer has not been modified and the statement $s(H')$ follows directly from $s(H)$. If this CAS succeeds, the tail pointer points to n. There is no change of $A(r')$ from $A(r)$. From $A(r)$ being equal to $SLin(L)$, $A(r')$ is equal to $SLin(L')$. For $I(r')$, because n is not NULL, *getTailPointer* does not point to a NULL pointer. Since n has been appended to the linked list, n is reachable from the head pointer through the next pointer of the node in the linked list. Then, $I(r')$ holds and the statement $s(H')$ holds.

Line 96 and 133: At line 96 and 133, the enqueueing operation returns without any returned value. Since the enqueuing operation has a trivial response, the statement $s(H')$ follows directly from $s(H)$.

Line 107 to 121: The *retrieveEnqueue* operation is invoked by process p to get the effect of p's last *detectableEnqueue* operation.

According to Definition 8.3, the *retrieveEnqueue* takes effect at line 121. Then, L' is an extension of L by one *retrieveEnqueue* operation with a response. The *MEM_INIT_VAL* value used in Detectable Queue is mapped to $\perp$ used in D{FIFO Queue}. The value of *r.pAddr[p][enq]* is not modified during the execution of line 107 to 121. Based on the value of *r.pAddr[p][enq]*, different values are returned at line 121.

When the value in *r.pAddr[p][enq]* is *MEM_INIT_VAL*, *NoExecutedOp* is returned as the effect value, and the memory initial value is returned both as the returned value and argument value at line 121. In this case, *getEnqStatusFromPAddr* of $A(r')$ returns $\perp$ as the status value. When the status value is $\perp$, based on Axiom 4.11, the response of *retrieveEnqueue* includes *NoExecutedOp* as the effect value, $\perp$ as both the returned value and argument value.

When the value in *r.pAddr[p][enq]* is a queue node with flag *OPCOMPLETE_FLG*, *HasEffect*, *AckResponse* and the enqueued node are returned as the effect value, returned value and argument value at line 121. In this case, *getEnqStatusFromPAddr* of $A(r)$ returns *END_DECT_OP* as the status value. When the status value is *END_DECT_OP*, based on Axiom 4.10, the response of *retrieveEnqueue* includes *HasEffect*, *AckResponse* and the enqueued node as the effect value, returned value and argument value.

When the value in *r.pAddr[p][enq]* is a queue node without flag *OPCOMPLETE_FLG*, *NoEffect*, the memory initial value and the enqueued node are returned as the effect value, returned value and argument at line 121. In this case, *getEnqStatusFromPAddr* of $A(r)$ returns *BEG_DECT_OP* as the status value. When the status value is *BEGIN_DECT_OP*, based on Axiom 4.9, the response of *retrieveEnqueue* includes *NoEffect*, $\perp$ and the enqueued node are returned as the effect value, returned value and argument value.

Therefore, the response at line 121 is the same as the response defined in the sequential specification of $D\{FIFO\ Queue\}$, which means the history L' is legal. There is no change from $SLin(L')$ from $SLin(L)$. Also, there is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. Therefore, $s(H')$ follows directly from $s(H)$.

Line 139: At line 139, *OP_INIT_VAL* is set into the dequeuing private variable of p. Based on Definition 8.4, the *prepareDequeue* operation does not take effect at this line, which means L' is equal to L and $SLin(L')$ is equal to $SLin(L)$. Since the value in the private variable has not been flushed, there is no change of $A(r')$ from $A(r)$. Also, because this step does not affect the representation invariant, $I(r')$ follows from $I(r)$. Therefore, the statement $s(H')$ follows directly from $s(H)$, which means that $s(H')$ holds and $A(r')$ is equal to $SLin(L')$.

Line 140: At line 140, the value in the dequeuing private variable is flushed. Based on Definition 8.4, the *prepareDequeue* operation takes effect at this line. Then, L' is an extension of L by one *prepareDequeue* operation. Since *prepareDequeue* has a trivial response (void response), the linearizability of L' follows from the linearizability of

L. According to Axiom 4.19 of $D\{FIFO\ Queue\}$, $SLin(L')$ changes from $SLin(L)$ with $BEGIN\_DECT\_OP$ as the status value and $\perp$ as the returned value for p. Since the value in the dequeuing private variable is flushed, $A(r')$ changes from $A(r)$ with $BEGIN\_DECT\_OP$ as the status value and $\perp$ as the returned value for p. The change of $SLin(L')$ is the same as the change of $A(r')$. Therefore, $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. Since this step does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds. Then the statement $s(H')$ holds.

Line 171 and 205: At line 171, p reads the last node of the linked list, which is referenced by the tail pointer, and puts it into a variable called *last*. According to Definition 8.5, the dequeuing operation takes effect at line 171 and 205 if for *detectableDequeue*, p flushes the value in the private variable at line 177 (strong detectability) or returns NULL at line 179 (weak detectability), and for *dequeue*, p returns NULL at line 209. After the execution of line 171 and 205, neither of these conditions in the definition of the linearization point is satisfied. Therefore, L' is equal to L and $SLin(L')$ is equal to $SLin(L)$. There is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. The statement $s(H')$ follows directly from $s(H)$.

Line 176: At line 176, a NULL pointer is set to the dequeuing private variable by process p when the head and tail pointers point to the same node and the next pointer of the node is NULL (checked at line 173 and 174). According to Definition 8.5, *detectableDequeue* executed by p takes effect at line 171 if p flushes the value in the private variable at line 177 (strong detectability) or returns NULL at line 179 (weak detectability). After the execution of the current line, neither of these conditions in the definition of the linearization point is satisfied. Therefore, there is no change of L' from L or $SLin(L')$ form $SLin(L)$. Also, since the value in the private variable has not been flushed to the persistent memory, $A(r')$ returns $BEGIN\_DECT\_OP$ as the status value and $\perp$ as the returned value for p. There is no change of $A(r')$ from $A(r)$. From $A(r)$ being equal to $SLin(L)$, $A(r')$ is equal to $SLin(L')$. Since this step does not affect the representation invariant, $I(r')$ follows from $I(r)$. Then the statement $s(H')$ holds.

Line 177: At line 177, the NULL pointer in the dequeuing private variable is flushed to the persistent memory by process p. Since this step does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds. From Definition 8.5, *detectableDequeue* executed by p takes effect at line 171 if the value in the private variable is flushed at the current line. Then, L' has one more *detectableDequeue* operation with a matching response executed by p than L, which takes effect at line 171. Based on the construction of the candidate linearization history, the matching response is a NULL pointer.

After the execution of line 171 (the linearization point), p has read the first node f of the linked list through the head pointer, the next pointer fn of f and the last node l through the tail pointer. The representation value after line 171 being executed is denoted r''. The node f and l are the same node and fn is NULL (checked at line 173 and 174 respectively). From the $I(r'')$, the *deqThreadID_* of the node referenced by the tail pointer is -1. Therefore, the *deqThreadID_* of f is -1. From the definition of $queueNodeSeq$ in the $A(r'')$, according to clause 1, $queueNodeSeq(r'')$ transverses the linked list from f and stops at f since the *deqThreadID_* of f is -1. Then, fn becomes the beginning node. Since fn is a NULL pointer, according to clause 3, $queueNodeSeq(r'')$ will return an empty sequence, which indicates the queue is empty. In this case, according to Axiom 3.3, the response is a NULL pointer. Since this added dequeuing operation does not modify the state of queue, the linearizability of L' follows from the linearizability of L and L' is a legal history. According to Axiom 4.20, when the queue is empty, $SLin(L')$ changes from $SLin(L)$ with *END_DECT_OP* as p's status value and a NULL pointer as p's returned value. Since the value in the dequeuing private variable is flushed, $A(r')$ changes from $A(r)$ with *END_DECT_OP* as the status value and the NULL pointer as the returned value for p. The change of $SLin(L')$ is the same as the change of $A(r')$. Therefore, $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. Then the statement $s(H')$ holds.

Line 179 (Strong Detectability): At line 179, *detectableDequeue* returns a NULL pointer. From Definition 8.5, the dequeuing operation has already taken effect before this line being executed. There is no change of L' from L or change of $SLin(L')$ from $SLin(L)$. Also, there is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. The statement $s(H')$ follows directly from $s(H)$. From the analysis of line 177, the matching response added to L is a NULL pointer, which is the same response returned at this line.

Line 179 (Weak Detectability) and 209: After the execution of line 171 and 205, p has read the first node f of the linked list through the head pointer, the next pointer fn of f and the last node l through the tail pointer. When f and l are the same node, and fn is NULL (checked at line 173, 174, 207 and 208), according to the analysis of line 177, it indicates the queue is empty after line 171 and 205 being executed. The representation variable after line 171 and 205 being executed is denoted r''. Then, a NULL pointer is returned at line 179 and 209.

From Definition 8.5, the dequeuing operation takes effect at line 171 and 205 if p returns NULL at line 179 (weak detectability) and at line 209. Then, L' has one more dequeuing operation with a NULL pointer as response executed by p than L, which takes effect at line 171 and 205. When the queue is empty for r'', according to Axiom 3.3, the response is a NULL pointer. Since this added dequeuing operation does not modify the state of queue, the linearizability of L' follows from the linearizability of L and L' is a legal history. According to Axiom 3.3 and Axiom 4.20, there is no change of $SLin(L')$ from $SLin(L)$. Also, there is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. The statement $s(H')$ follows directly from $s(H)$.

Line 185 and 186: When the head and tail pointers of the linked list point to different nodes, at line 185, the first node of the linked list, denoted f, is set to the dequeuing private variable and at line 186, the value is flushed to the persistent memory. According to Definition 8.5, *detectableDequeue* has not taken effect yet. Therefore, for line 185 and 186, there is no change of L' from L or of $SLin(L')$ from $SLin(L)$. For line 185, since the value in the private variable has not been flushed, $A(r')$ does not change from $A(r)$. For line 186, since process p has not updated the *deqThreadID_* field of f to p at line 188, $A(r')$ does not change from $A(r)$. Therefore, $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. Since this step does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds. Then the statement $s(H')$ holds.

Line 188 and 213: To mark the first node f (defined at line 169 and 203) of the linked list a removed node, at line 188 and 213, process p tries to use CAS to put its process identifier (p) to the *deqThreadId_* field of f when the value of *deqThreadId_* is -1. The queue node referenced by f's next pointer is denoted fn. According to Definition 8.5, the dequeuing operation takes effect at line 188 and 213 if the CAS operation at line 188 and 213 succeeds and the value of the *deqThreadID_* field has been flushed. After the execution of line 188 and 213, the value of the *deqThreadID_* field has not been flushed yet. Therefore, there is no change of L' from L or $SLin(L')$ from $SLin(L)$. If the CAS operation fails, the *deqThreadID_* field has not been modified and the statement $s(H')$ follows directly from $s(H)$. If the CAS operation succeeds, r' changes with f being marked as a removed node from the linked list. Since the new value of the *deqThreadId_* field has not been flushed, $A(r')$ returns with a node sequence starting from fn, *BEGIN_DECT_OP* as the status value and $\perp$ as the returned value for p. There is no change from $A(r')$ to $A(r)$. Therefore, $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. For $I(r')$, this change affects functions reading the marked removed nodes. Because f is the first node of the linked list read from the head pointer, the function *areMarkedRemovedNodesConsecutive* in $I(r')$ returns *true*. Since f is not the node referenced by the tail pointer (checked at line 172 and 206), which is denoted nt, the execution of line 188 and 213 does not modify the *deqThreadID_* filed of nt. Since $I(r)$ holds, which means the *deqThreadID_* filed of nt is -1, for $I(r')$, the *deqThreadID_* filed of nt is also -1. Therefore, $I(r')$ holds. Then the statement $s(H')$ holds.

Line 189, 195, 214 and 220: At line 188 and 213, process $p1$ successfully uses CAS to put its process identifier ($p1$) to the $deqThreadId\_$ field of the first node f (referenced by the head pointer at line 169 and 203) when the value of $deqThreadId\_$ is -1. The queue node referenced by f's next pointer is denoted fn. At line 189, 195, 214 and 220, process p executes a flush operation to persist the value of the *deqThreadId_* field of f. For the execution of line 189 and 214, p is equal to $p1$. Since this flush step does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds.

Based on Definition 8.5, the dequeuing operation executed by $p1$ takes effect at line 188 and 213 if the CAS operation at line 188 and 213 succeeds, and the value of the $deqThreadID\_$ field has been flushed. Then, L' has one more dequeuing operation, denoted *op*, with a matching response executed by $p1$, which takes effect at line 188 and 213. Based on the construction of the candidate linearization history, the matching response is fn.

At line 172 and 206, it is checked that the first node f and the last node l (referenced by the tail pointer at line 171 and 205) are not the same queue node. At line 188 and 213, the CAS operation executed by $p1$ succeeds, which means the $deqThreadID\_$ field of f is -1. From the definition of $queueNodeSeq(r)$ in $A(r)$, according to clause 1, $queueNodeSeq(r)$ transverses the linked list from f and stops at f since the *deqThreadID_* of f is -1. Then, fn becomes the beginning node. From $I(r)$, l referenced by the head pointer is not a NULL pointer and l is reachable from f through the next pointer of each node. Based on this, fn must be a queue node instead of a NULL pointer. When fn is not a NULL pointer, according to clause 4, $queueNodeSeq(r)$ returns a node sequence starting with fn, which indicates the queue is not empty. According to Axiom 3.2 for an non-empty queue, the correct response is the first element of the queue node sequence, which is fn. The correct response is equal to the matching response.

After process $p1$ successfully uses CAS to put its process identifier to the $deqThreadId\_$ field of f, the value of the $deqThreadId\_$ field is flushed by $p1$ and other helping dequeuing threads at line 189, 195, 214 and 220. Then, the head pointer is moved to fn by $p1$ and other helping dequeuing threads at line 190, 196, 215 and 221 using CAS. Only after the head pointer is moved to fn, which means *op* has already taken effect, by reading fn from the head pointer at line 169 and 203, it is possible for another dequeuing process to successfully put its process identifier to the *deqThreadId_* field of fn using CAS at line 91 and 130 or to return a NULL pointer at line 176 and 209. So, for the history L', *op* must be the last dequeuing operation. Also, the matching response of *op* is equal to the correct response fn. Therefore, L' is a legal history and the linearizability of L' follows from the linearizability of L. According to Axiom 3.2 and Axiom 4.20, $SLin(L')$ changes from $SLin(L)$ with the node sequence excluding fn, *END_DECT_OP* as the status value and fn as the returned value for $p1$.

Since the value of the *deqThreadId_* of fn, which is $p1$, has been flushed, the *queueNodeSeq* function of $A(r')$ returning a sequence of nodes from the next pointer of fn, which exclude fn. Also, since the value in $p1$'s dequeuing private variable is f, whose *deqThreadId_* in the persistent memory is equal to $p1$, *getDeqStatusFromPAddr* returning *END_DECT_OP* as $p1$'s status value and *getDeqRetValFromPAddr* returning the dequeued node fn as $p1$'s returned value. The change of $SLin(L')$ is the same as the change of $A(r')$. Therefore, $A(r)$ being equal to $SLin(L)$ implies that $A(r')$ is equal to $SLin(L')$. Then the statement $s(H')$ holds.

Line 190, 196, 215 and 221: At these lines, process p tries to use CAS to move the head pointer to the next pointer of the first node (defined at 169 and 203) whose *deqThreadId_* is not -1. The first node is denoted f and its next pointer is denoted fn. According to the analysis of line 189, 195, 214 and 220, fn is a dequeued node. Since the dequeuing operation executed by p has already taken effect at line 188 and 213 after the value of the $deqThreadID\_$ field has been flushed at line 177 and 214, there is no change of L' from L or $SLin(L')$ from $SLin(L)$.

If this CAS operation fails, the head pointer has not been modified and the statement $s(H')$ follows directly from $s(H)$. If the CAS succeeds, the head pointer is moved to fn. For the function $queueNodeSet$ in $A(r)$, since the value of the *deqThreadId_* of fn has been flushed at line 189, 195, 214 and 220, the *queueNodeSeq* function returns a sequence of nodes from the next pointer of fn, which excludes fn. For the function $queueNodeSet$ in $A(r')$, since the head pointer points to fn, the $queueNodeSet$ function returns a sequence of nodes from the next pointer of the head pointer without fn, which is the same as the node sequence returned by $A(r)$. There is no change of $A(r')$ from $A(r)$. From $A(r)$ being equal to $SLin(L)$, $A(r')$ is equal to $SLin(L')$. For $I(r')$, since fn is not a NULL pointer (checked at line 174 and 208), the head pointer does not point to a NULL pointer. Besides, the node pointer referenced by the tail pointer is reachable from fn through the next pointer of each node. Therefore, $I(r')$ holds and the statement $s(H')$ holds.

Line 191 and 216: At line 191 and 216, the next pointer of the first node f (defined at

169 and 203) is returned. The next pointer of f is denoted fn. Since the dequeuing operation executed by p has already taken effect at line 188 and 213 after the value of the $deqThreadID\_$ field has been flushed at line 177 and 214, there is no change of L' from L or $SLin(L')$ from $SLin(L)$. Also, there is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. The statement $s(H')$ follows directly from $s(H)$. According to the analysis of line 189, 195, 214 and 220, the matching response added to L is fn, which is the same response returned at line 191 and 216.

Line 148 to 164: The *retrieveDequeue* operation is invoked by process p to get the effect of p's last *detectableDequeue* operation.

According to Definition 8.6, the *retrieveDequeue* takes effect at line 164. Then, L' is an extension of L by one *retrieveDequeue* operation with a response. The value of *r.pAddr[p][deq]* is not modified during the execution of line 148 to 164. Based on the value of *r.pAddr[p][deq]*, different values are returned at line 164. The *MEM_INIT_VAL* value used in Detectable Queue is mapped to $\perp$ used in D{FIFO Queue}.

When the value in *r.pAddr[p][deq]* is *MEM_INIT_VAL*, *NoExecutedOp* is returned as the effect value, and the memory initial value is returned as the returned value at line 164. In this case, *getDeqStatusFromPAddr* of $A(r')$ return $\perp$ as the status value. When the status value is $\perp$, based on Axiom 4.23, *NoExecutedOp* is returned as the effect value, and $\perp$ is returned as the response value.

When the value in *r.pAddr[p][deq]* is *OP_INIT_VAL*, *NoEffect* and the memory initial value are returned as the effect value and returned value at line 164. In this case, *getDeqStatusFromPAddr* of $A(r')$ returns *BEG_DECT_OP* as the status value. When the status value is *BEGIN_DECT_OP*, based on Axiom 4.21, *NoEffect* and $\perp$ are returned as the effect value and returned value.

Otherwise, the value in *r.pAddr[p][deq]*, denoted v, is a queue node pointer or a NULL pointer. At line 164, *HasEffect* is returned as the effect value. If v is a node pointer, the next pointer of v is returned as the returned value at line 164. From the analysis of line 188 to 191, the node referenced by the next pointer of v is the dequeued node when the queue is not empty. If v is a NULL pointer, the NULL pointer is returned as the returned value at line 164. From the analysis of line 176 to 179, a NULL pointer is put into *r.pAddr[p][deq]* only when the queue is empty. In this case, *getDeqStatusFromPAddr* of $A(r')$ returns *END_DECT_OP* as the status value. When the status value is *END_DECT_OP*, based on Axiom 4.22, *HasEffect* is returned as the effect value. When the queue is not empty, the dequeued node is returned as the returned value, and when the queue is not empty, a NULL pointer is returned as the returned value.

Therefore, the response at line 164 is the same as the response defined in the sequential specification of $D\{FIFO\ Queue\}$, which means the history L' is legal. There is no change of $SLin(L')$ from $SLin(L)$. Also, there is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. Therefore, $s(H')$ follows directly from $s(H)$.

Case B: e is an event of $D\{FIFO\ Queue\}$. The event e must be an invocation or response of an operation of $D\{FIFO\ Queue\}$. If e is an invocation, since it is the last event with no code lines in the Detectable Queue being executed, there is no state change. If e is a response, e is added in the history after an enqueuing or dequeuing operation completes at line 96, 133. 179, 191, 209 or 216. Therefore, there is no change of $A(r')$ from $A(r)$ or $I(r')$ from $I(r)$. According to the construction of L, there is no change of L' from L or $SLin(L')$ from $SLin(L)$. Then, $s(H')$ follows directly from $s(H)$.

Case C: e is a crash event. The recovery of Detectable Queue is started (Section 5.3.3) to make sure $A(r')$ is equal to $SLin(L')$. Based on the definition of the linearization points, none of them happens in the recovery procedure. Since no operation reaches its linearization point during recovery, L' is equal to L. Therefore, $SLin(L')$ is equal to $SLin(L)$, and L' is legal because L is legal.

The recovery procedure first goes through the linked list from the head pointer and checks whether the current visited node n is a marked removed node (the value in the $deqThreadID\_$ field of n is -1). If it is, the head pointer will be moved to the next pointer of n, denoted nn, and flushed to the persistent memory. Since the value of the $deqThreadID\_$ field is in the persistent memory, it means the value has already been flushed at line 189, 195, 214 or 220 before crash. From clause 1 of the definition of the function $queueNodeSet$ in $A(r)$, $queueNodeSet$ transverses the linked list from the head pointer and stops at a node ns if ns has not been flushed or the $deqThreadID\_$ of ns is -1. The next pointer of ns becomes the start node of the node sequence returned by $queueNodeSet$. Therefore, the function $queueNodeSet$ in $A(r)$ returns a sequence of nodes without marked removed nodes, and moving the head pointer to nn will not affect the returned value of $queueNodeSet$. There is no change of the returned value of $queueNodeSet$ in $A(r')$ from $A(r)$. Since $I(r)$ holds, the node referenced by the tail pointer, denoted nt, of the linked list is reachable from the node referenced by the head pointer, denoted nf, through the next pointer of each node. Then, for each node ni before nt in the linked list (if there is any), nt is also reachable from the next pointer of ni and the next pointer of ni cannot be a NULL pointer. According to $I(r)$, nt cannot be the marked removed node and the marked removed nodes must be consecutive nodes from nf, which means the marked removed nodes must appear before nt in the linked list. Then, the next pointer of each marked removed node must not be a NULL pointer. Therefore, after moving the head pointer to nn, the returned value of the *getHeadPointer* function of $I(r')$ must not be a NULL pointer and the returned value of the *isTailReachableFromHead* function of $I(r')$ must be true. This step does not affect the returned values of the other functions in $I(r')$ and $I(r')$ holds.

If the value in a process p's enqueing private variable ($r.pAddr[p][enq]$) does not have flag *OPCOMPLETE_FLG* and the value is equal to the current visited node n, the flag *OPCOMPLETE_FLG* will be added to the value in $r.pAddr[p][enq]$ and the value in $r.pAddr[p][enq]$ will be flushed. Then, according to the definitions of the *getEnqStatusFromPAddr* and *getEnqArgFromPAddr* functions, *getEnqStatusFromPAddr* of $A(r')$ returns *END_DECT_OP* as p's status value, and *getEnqArgFromPAddr* of $A(r')$ returns n as p's argument value. Since n has been appended to the linked list in the persistent memory, it means that the *detectableEnqueue* operation executed by p with the argument n has already taken effect before crash at line 92, 98, 131, 135, 181 or 210. According to Axiom 4.8, for *detectableEnqueue*, $SLin(L)$ has *END_DECT_OP* as p's status value and n as p's argument, which are equal to the returned values of *getEnqStatusFromPAddr* and *getEnqArgFromPAddr* of $A(r')$ for process p. Since the modification of the values in private variables does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds.

If the traversal of the linked list reaches the tail pointer and the current node n's next pointer, denoted nn, is not NULL, the tail pointer will be moved to nn and flushed to the persistent memory. Since the value of n's next pointer is in the persistent memory, it means n has already been flushed before the crash at line 92, 98, 131, 135, 181 or 210. From clause 2 of the definition of the function $queueNodeSet$ in $A(r)$, $queueNodeSet$ goes through the linked list and ends at a node ne if ne has not been flushed or the next pointer of ne is NULL. The node ne becomes the end node of the returned node sequence by $queueNodeSet$. Therefore, the function $queueNodeSet$ in $A(r)$ returns a sequence of nodes including nn, and moving the tail pointer to nn will not affect the returned value of $queueNodeSet$. There is no change of the returned value of $queueNodeSet$ in $A(r')$ from $A(r)$. Since nn is not NULL, the returned value of the *getTailPointer* function of $I(r')$ will not be NULL. Since $I(r)$ holds, the node referenced by the tail pointer, denoted nt, is reachable from the node referenced by the head pointer through the next pointer of each node in the linked list. Then, nn, which is referenced by the next pointer of nt, is also reachable from the node referenced by the head pointer. The returned value of the *isTailReachableFromHead* function of $I(r')$ must be true. This step does not affect the returned values of the other functions in $I(r')$ and $I(r')$ holds.

After the traversal of the linked list, the recovery procedure will check the value in each process p's enqueuing private variable ($r.pAddr[p][enq]$). If the value in $r.pAddr[p][enq]$, denoted v, is a queue node without flag *OPCOMPLETE_FLG* and the *deqThreadId_* field of v is a value, denoted $p1$, which is not -1, recovery will add *OPCOMPLETE_FLG* to the value in $r.pAddr[p][enq]$ and flush the value to the persistent memory. Then, the *getEnqStatusFromPAddr* function of $A(r')$ returns *END_DECT_OP* as p's status value, and *getEnqArgFromPAddr* of $A(r')$ returns v as p's argument value. Since the value of the *deqThreadId_* field of v is $p1$ and the recovery procedure never modifies the *deqThreadId_* field, it means another process $p1$ executed a dequeuing operation, in which process $p1$ modified the *deqThreadId_* field of v to its process identifier at line 188 or 213 after p executed a *detectableEnqueue* operation with v as the argument. Therefore, v has already been appended to the linked list and the *detectableEnqueue* operation executed by p with v as the argument must have taken effect at line 92, 98, 131, 135, 181 or 210 before the crash. According to Axiom 4.8, for *detectableEnqueue*, $SLin(L)$ has *END_DECT_OP* as p's status value and v as p's argument, which are equal to the returned values of *getEnqStatusFromPAddr* and *getEnqArgFromPAddr* of $A(r')$ for process p. Since the modification of the values in private variables does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds.

Recovery will also check the value in each process p's dequeuing private variable ($r.pAddr[p][deq]$). If the value in $r.pAddr[p][deq]$, denoted v, is a node whose *deqThreadID_* is not equal to the process identifier p, recovery will set *OP_INIT_VAL* to $r.pAddr[p][deq]$ and flush the value. According to clause 2 of the definitions of the *getDeqStatusFromPAddr* and *getDeqRetValFromPAddr* functions, *getDeqStatusFromPAddr* of $A(r)$ returns *BEGIN_DECT_OP* and *getDeqRetValFromPAddr* of $A(r)$ returns $\perp$. According to clause 5 of the definitions, *getDeqStatusFromPAddr* of $A(r')$ returns *BEGIN_DECT_OP* and *getDeqRetValFromPAddr* of $A(r')$ returns $\perp$. There is no change of the returned value of *getDeqStatusFromPAddr* and *getDeqRetValFromPAddr* in $A(r')$ from $A(r)$. Since the modification of the values in private variables does not affect the representation invariant, $I(r')$ follows $I(r)$ and holds. After this check, the recovery procedure finishes.

From the above analysis, first $I(r')$ always holds in case C. Also, there is no change of the returned value for *queueNodeSeq*, *getDeqStatusFromPAddr* and *getDeqRetValFromPAddr* of $A(r')$ from $A(r)$ in case C. Since the returned values of these functions in $A(r)$ are the strict linearized values of L, the returned values of these functions in

$A(r')$ are also the strict linearized values of L.

For the *getEnqStatusFromPAddr* function and *getEnqArgFromPAddr* function of $A(r)$ in case A and B, according to their definitions, before crash, when a process p has not executed a *detectableEnqueue* operation (clause 2), or p's *detectableEnqueue* operation has not taken effect (clause 3), or the value in p's enqueuing private variable has been updated with *OPCOMPLETE_FLG* (the first condition of clause 1), the returned values of the two functions are decided only by the values in p's enqueuing private variable in the persistent memory. From the analysis of recovery, the value of p's enqueuing private variable has not be changed during recovery in these situations. For these situations, there is no change of the returned values for the *getEnqArgFromPAddr* function and *getEnqArgFromPAddr* function in $A(r')$ from $A(r)$ in case C. Since for these situations, the returned values of these functions in $A(r)$ are the strict linearized values of L, the returned values of these functions in $A(r')$ are also the strict linearized values of L.

Except the three situations described in the last paragraph, there is one situation left (the second condition of clause 1): p's *detectableEnqueue* operation has taken effect but p's enqueuing private variable has not been updated with *OPCOMPLETE_FLG* before crash. For this situation, from the analysis of recovery, p's enqueueing private variable will be updated with *OPCOMPLETE_FLG*, and the returned values of *getEnqStatusFromPAddr* and *getEnqArgFromPAddr* are the strict linearized values of L. Therefore, the returned values of *getEnqStatusFromPAddr* and *getEnqArgFromPAddr* of $A(r')$ in case C are the strict linearized values of L in all situations.

Above all, after recovery, $A(r')$ is equal to $SLin(L)$. Since $SLin(L)$ is equal to $SLin(L')$, $A(r')$ is equal to $SLin(L')$.

□

Theorem 8.2. *Detectable Queue is strict linearizable.*

Proof. Implied by Lemma 8.1, Detectable Queue is strict linearizable. □

8.1.2 Proof of Liveness

In this section, I prove the progress property of Detectable Queue.

Theorem 8.3. *Detectable Queue is recoverable lock-free.*

Proof. Suppose there are only finitely many failures in a history H.

For the *detectableEnqueue* and *enqueue* operations of Detectable Queue, enqueuing processes read the last node from the tail pointer (line 87 and line 126) and try to put a new node to the next pointer of the last node using CAS (line 91 and line 130). One enqueuing process, denoted p, will succeed in executing this CAS operation and the other enqueuing processes will fail due to contention. Process p and other failed processes try to advance the tail pointer to the new node using CAS (line 99, line 136, line 95 and line 132) and one of them must succeed. Then process p returns and failed enqueuing processes read the updated value from the tail pointer, trying to append a new node again. From the above analysis, enqueuing processes will make progress and at least one process will succeed in enqueuing a node unless infinitely many failures happen.

For the *detectableDequeue* and *dequeue* operations of Detectable Queue, dequeuing processes read the first node, next pointer of the first node, and last node from the head pointer and tail pointer (line 169 - 171 and line 203 - 205). Dequeuing processes first check whether the first node and last node point to the same pointer. If they point to the same node and the next pointer of the first node is NULL, a NULL pointer is returned directly at line 179 and 209 after the checks. If the next pointer of the first node is not NULL, which means a node n has already been appended to the linked list but the tail pointer has not been advanced, dequeuing processes help to advance the tail pointer to n using CAS (line 182 and line 211) and then retry from reading the values of the head and tail pointers. When the head pointer and tail pointer point to different nodes, dequeuing processes try to put their identifier to the $deqThreadID\_$ field of the first node using CAS (line 188 and line 213). One dequeuing process, denoted p, will succeed and the other dequeuing processes will fail. Process p and failed dequeuing processes try to advance the head pointer to its next pointer using CAS (line 196, line 221, line 190 and line 215) and one process must succeed. Then process p returns the next pointer of the first node as the dequeued node and failed dequeuing processes retry from reading the values of the head and tail pointers. From the above analysis, dequeuing processes will make progress. For an empty queue, processes with return with a NULL pointer and for a non-empty queue, at least one process will succeed in dequeuing a node unless infinitely many failures happen.

Therefore, when there are finitely many failures, some executing process will always finish its enqueuing/dequeuing operation as long as some process continues to take steps.

For the *prepareEnqueue* and *prepareDequeue* operation, a process sets values to its private enqueuing and dequeuing variables and flushes the values. For the *retrieveEnqueue* and *retrieveDequeue* operations, a process returns values based on the values in its enqueuing and dequeuing variables. Since there is no loop in these operations, a process will always make progress and complete the operations unless infinitely many failures happen.

For the recovery of the Detectable Queue, it is executed by a single recovery process in the absence of concurrency. The recovery traverses the linked list from the head pointer to the last node, and modifies the head pointer, tail pointer and values in the private variables if necessary. From the proof of safety of Detectable Queue (Section 8.1.1), the last node of the linked list always exists and the traversal of the linked list will end once the recovery reaches the last node. Therefore, the recovery process will always make progress and complete the operation unless infinitely many failures happen.

□

Chapter 9

Experimental Results

In this chapter, I evaluate the performance and scalability of the Detectable Queue algorithm (Section 5.3), General CASWithEffect algorithm (Section 6.2) and Fast CASWithEffect algorithm (Section 6.3) with different testing threads.

9.1 Experimental Setup

I run the experiments on a server with four Intel(R) Xeon(R) Gold 6230 processors with a clock speed of 2.10GHz and 20 hyperthreaded cores per processor. The machine has 768GB RAM and 3TB of Optane persistent memory. The turbo boost of the processor is disabled to reduce random variation in running times. The operation system is Ubuntu 20.04. The experiments are executed on Optane persistent memory.

The source code of the algorithms and testing code are written in C++ and built through g++ 9.3.0. I use Intel PMDK library [33] (version 1.8) to access Optane persistent memory. The memory-mapped file containing data structures is in an XFS file system. The size of the mapped file is pre-calculated by the data structures employed in the experiments. I use the *pmem_persist* function of PMDK to flush the entire cache line to the persistent memory.

9.2 CasWithEffect Primitive

Inspired by Wang et al.'s PMwCAS algorithm [36], I presented two CASWithEffect algorithms: General CASWithEffect and Fast CasWithEffect to hold the effect of a CAS operation in Chapter 6. To compare the performance of General CASWithEffect, Fast CasWithEffect and PMwCAS, I set up a fixed-size array in the persistent memory. Each testing thread used General CASWithEffect, Fast CASWithEffect or PMwCAS to modify one random word in that array and stored the effect into the private variable of the thread when the CAS operation is successful. Every test with a different number of testing threads is executed for 10 seconds and repeated for 5 times without failure injection. Then, the average throughput of CAS operations is recorded and displayed in a plot. Since the machine has 4 processors and 20 cores for each processor, I measure 1 to 80 threads. The array size is 100 and 1000 in my experiments to provide different contention levels.

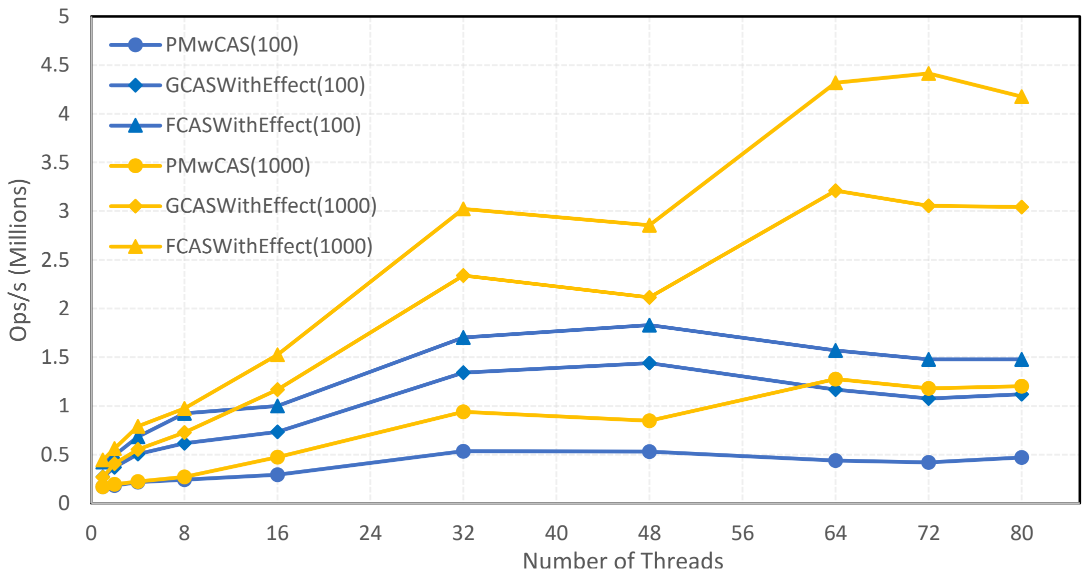

Figure 9.1: Performance of CASWithEffect and PMwCAS with different numbers of threads.

The result of the experiment is shown in Figure 9.1. Each throughput value in the plot is an average throughput as explained above. The standard deviation for each throughput value is also calculated based on the test results with the same thread number and array size, which is less than 0.001. As expected, General CASWithEffect outperforms PMwCAS in every round of testing. For both 100-size and 1000-size array, when there is one thread running, CASWithEffect outperforms PMwCAS by 1.6X. With more threads becoming running, the performance improvement becomes more than 2.5x. On average, the throughput increase for both 100-size array and 1000-size array is 2.4x. The main reason for this improvement is the reduced usage of flushes related to installing descriptors. In General CASWithEffect, only the shared address is installed with the descriptor while in PMwCAS, each address needs to be installed with the descriptor. The test results also show that Fast CASWithEffect improves the performance of General CASWithEffect by 1.4x on average for both 100-size and 1000-size array tests since Fast CasWithEffect avoids the descriptor installing phase. The trend in Figure 9.1 shows the throughput of both General and Fast CASWithEffect modifying an array rises as the number of running threads increases. Then the throughput declines slowly as more threads compete for updating the testing array and the contention becomes high. In 1000-size array tests, for General CASWithEffect, the throughput reaches 3.2 million/s when there are 64 threads running, and for Fast CASWithEffect, the throughput reaches 4.4 million/s where there are 72 threads running. The experiment shows that both General CASWithEffect and Fast CASWithEffect are scalable algorithms.

9.3 Detectable Queue

In Chapter 5, I presented the algorithm of Detectable Queue which provides strong detectability, weak detectability and non detectability of a lock-free queue. When *detectableEnqueue* and *detectableDequeue* operations are invoked, their effects can be retrieved after recovery through *retrieveEnqueue* and *retrieveDequeue* respectively. Besides, the *isStrongDetectability_* flag is used to set which detectability (strong/weak) to provide during run time. When *enqueue* and *dequeue* operations of Detectable Queue are invoked, the effects of these operations have not been recorded and cannot be retrieved after recovery.

In the experiment, I compare the performance of different detectability using Detectable Queue. Since the queue is not a scalable data structure, I test the performance with threads varying from 1 to 20 on one processor. The queue is initialized with 16 queue nodes. All executing threads execute a pair of enqueue and dequeue operations for 30 seconds and repeat it for 10 times with the average throughput being recorded. As explained in the memory management of queue nodes in Chapter 7, every thread gets a queue node from its free queue node list and executes an enqueuing operation with this node. The performance results are shown in Figure 9.2. As in the CASWithEffect experiments, the standard deviation of each throughput value is computed based on the queue performance with the same thread number, which is around 0.003. The plot shows that both the strong detectable and weak detectable implementations have nearly the same throughput while the non-detectable implementation has higher throughput. In the Detectable Queue algorithm, the only difference between the weak detectable and strong detectable operations is that the strong detectable needs to flush the returned value of a NULL pointer when the queue is empty. Since the queue is initialized with 16 queue nodes, with a maximum of 20 threads

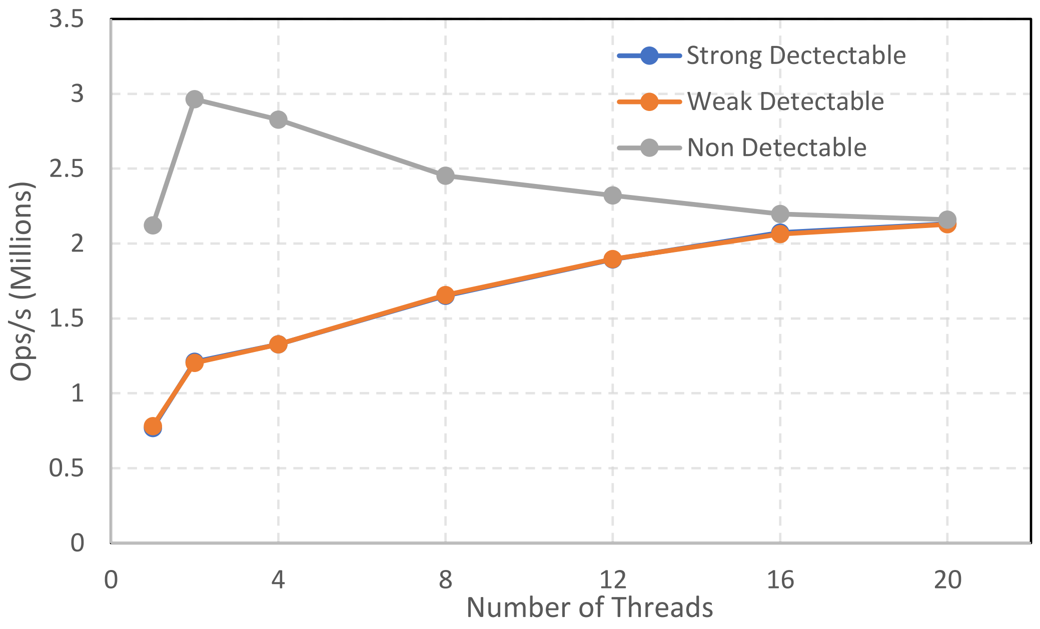

Figure 9.2: Performance of Detectable Queue with different detectability.

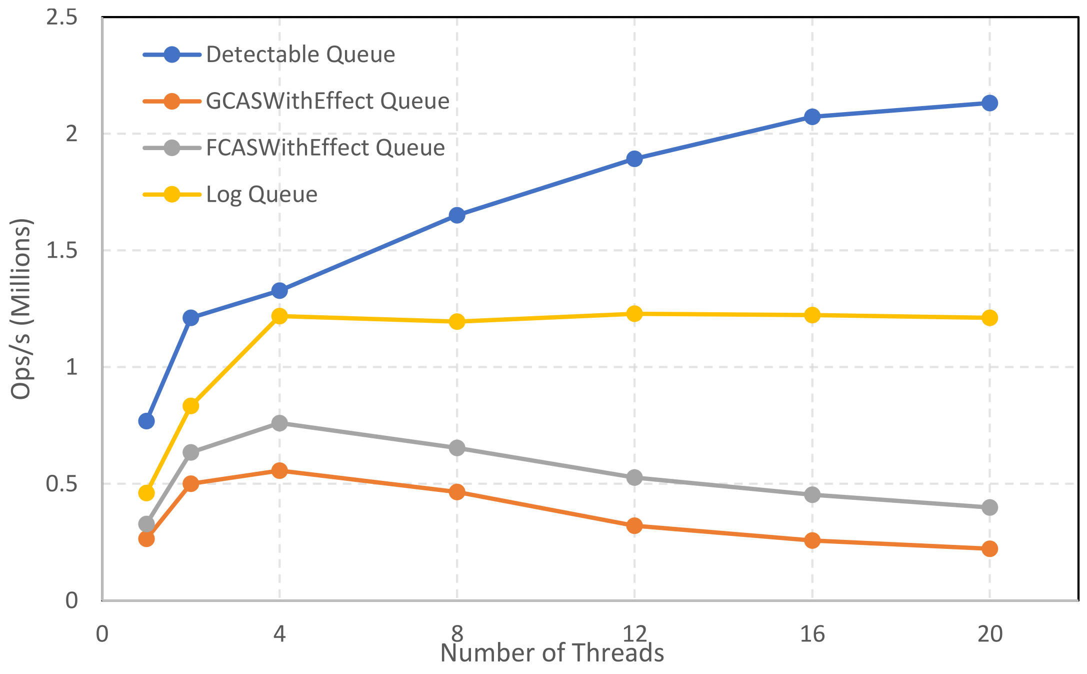

Figure 9.3: Performance of various queue implementations.

running, a thread will run a dequeuing operation with a returned queue node in most cases. Therefore, the extra overhead caused by strong detectability to weak detectability seems to be quite trivial. When there is one thread running without any contention, the non-detectable operations outperform detectable operations by 2.8x. For the non-detectable operations, neither the argument of the enqueuing operation nor the returned value of the dequeuing operation needs to be persisted and this saves non-trivial overhead caused by multiple flushes. With more threads running with higher contention, the performance of non-detectable operations decreases. When the total amount of running threads reaches 20, the throughout of non-detectable and detectable are nearly the same of 2.16 million/s and 2.13 million/s respectively. On average, the non-detectable operations outperform detectable operations by 1.7x.

I further measure and compare the performance of several queue implementations with strong detectability including Detectable Queue (Chapter 5), CASWithEffect Queue (Section 6.4.3) and Log Queue [13]. Log Queue is a specially-designed queue for persistent memory, which uses log variables to record the status of enqueuing and dequeuing operations. I choose Log Queue to compare against, because according to the definition of Detectability (Section 4.4), Log Queue provides equivalent strong detectability with the returned values and arguments being stored in its log variables. In the experiments, the memory management algorithm explained in Chapter 7 is employed to manage queue nodes and log variables for Log Queue. The performance results are shown in Figure 9.3. The performance of Detectable Queue in Figure 9.3 is the same as the "Strong Detectable" series of Figure 9.2. From the plot, both the General CASWithEffect queue and Fast CASWithEffect queues have lower performance because CASWithEffect, as a general algorithm, requires more flushes to keep a consistent status. Fast CASWithEffect Queue outperforms General CASWithEffect Queue by 1.5x on average since Fast CASWithEffect is an optimised algorithm of General CASWithEffect. Although CASWithEffect Queues perform lower than Detectable Queue and Log Queue, the implementation using CASWithEffect is much easier compared to other queues' complex design. Detectable Queue has the highest throughput, which performs better than Fast CASWithEffect Queue by 3.1x on average. Compared to Log Queue, when only one thread running without any contention, Detectable Queue outperforms Log Queue by 1.7x. When the running threads increasing from 4 to 20, the performance improvement of Detectable Queue becomes larger from 1.1x to 1.8x. On average, Detectable outperforms Log Queue by 1.5x. The reason of this performance improvement is that Detectable Queue uses private variables to record bookkeeping data which causes less overhead than maintaining log variables in Log Queue. The private variable for each thread is pre-allocated from persistent memory. However, in the Log Queue algorithm, each log variable is acquired and released dynamically in the beginning of an enqueuing/dequeuing operation. With more threads executing more operations, this overhead will become larger accordingly. Also, in Log Queue, multiple threads try to modify the same log variable by using CAS, which creates contention among threads. In Detectable Queue, there is no such contention since private variables can only be accessed by their owner threads. With less contention, the performance of Detectable Queue is improved.

According to the queue experiments described above, in general, the detectable data structures cause more overhead than non-detectable data structures with additional data being persisted. In the Detectable Queue algorithm, this overhead is 70% on average. However, with the number of running threads increasing from 1 to 20, the overhead decreases from 117% to 1%. This means the overhead brought by detectability can be minimized by specific detectable algorithms in situations of certain contention.

Chapter 10

Conclusion

The research work in this thesis focuses on building detectable data structures for persistent memory. In this thesis, I provide the first definition of detectability through a specific data type *DetectableT*, which is generic with few system assumptions. Based on *DetectableT*, Detectable Queue, a detectable MS queue implemented using private variables, is presented. To help implementing detectable data structures, I design a primitive called CASWithEffect, which can execute a CAS opepratoin and store the effect of this CAS operation into persistent memory atomically. By leveraging CASwithEffect, CASWithEffect Queue provides the same detectability as Detectable Queue but with a much simpler design. During experiments, Detectable Queue and CasWithEffect Queue are compared with another queue algorithm – Log Queue. As expected, the experiment results show that the detectable operations in Detectable Queue cost more overhead than non-detectable operations with additional information being stored in persistent memory. Also, CASWithEffect queue has lower performance because of the primitive's generality. Detectable Queue has the best performance. It outperforms Log Queue by 1.5x on average. One reason of the improvement is that Detectable Queue uses pre-allocated private variables from persistent memory instead of dynamically allocated log variables. Another reason is that using private variables avoids the contention caused by modifying the same shared log variables among threads.

References

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

[2] James H Anderson and Mark Moir. Universal constructions for multi-object operations. In *PODC*, volume 95, pages 184–193. Citeseer, 1995.

[3] Maya Arbel-Raviv and Trevor Brown. Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. *arXiv preprint arXiv:1708.01797*, 2017.

[4] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 7–16, 2018.

[5] Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable object. *arXiv preprint arXiv:2002.11378*, 2020.

[6] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM, 2019.

[7] Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 20:1–20:17, 2015.

[8] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. Makalu: fast recoverable allocation of non-volatile memory. In *Proc. of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 677–694, 2016.

[9] Trevor Brown and Hillel Avni. Phytm: Persistent hybrid transactional memory. *Proc. VLDB Endow.*, 10(4):409–420, 2016.

[10] 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 Sigplan Notices*, 47(4):105–118, 2012.

[11] 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*, SPAA '18, pages 259–269, 2018. ISBN 978-1-4503-5799-9.

[12] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 373–385, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL http://dl.acm.org/citation.cfm?id=3277355.3277392.

[13] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 28–40, 2018.

[14] Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 211–220. ACM, 2017.

[15] Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 17–26, 2018.

[16] Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 65–74, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3964-3. doi: 10.1145/2933057.2933087. URL http://doi.acm.org/10.1145/2933057.2933087.

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

[18] Rachid Guerraoui and Eric Ruppert. Linearizability is not always a safety property. In *International Conference on Networked Systems*, pages 57–69. Springer, 2014.

[19] 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.

[20] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[21] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2011.

[22] 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.

[23] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, 2016. ISBN 978-1-4503-4091-5.

[24] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 313–327, 2016.

[25] Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 30:1–30:15, 2017.

[26] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *ACM SIGPLAN Notices*, 51(4): 385–398, 2016.

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

[28] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1995.

[29] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 689–694, 2015.

[30] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*, pages 37:1–37:16, 2017.

[31] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 265–276, 2014.

[32] Aditya Ramaraju. Rglock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015.

[33] PMDK team at Intel Corporation. PMDK Homepage, 2020. URL https://pmem.io/pmdk/. [last accessed 10-December-2020].

[34] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[35] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104. ACM, 2011.

[36] 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.