Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Model-checking the libcds Library

Bachelor Thesis

Benedek Mezei

August 12, 2025

Supervisor: Prof. Dr. M. Kokologiannakis

Department of Computer Science, ETH Zürich

Abstract

The C++ concurrent data structures library libcds has many implementations of thread-safe data structures based on different published papers, as well as multiple safe memory reclamation schemes. Writing such concurrent code is notoriously difficult, even more so in the presence of the weak memory consistency models employed by many platforms, allowing the reordering of memory access instructions by the compiler and processor.

In this thesis, we will use the stateless model checker GenMC to formally reason about the correctness of the library's data structure implementations. GenMC explores all possible states of a program up to a bound to find states where some invariant is violated. To combat the typical state space explosion, GenMC reduces the workload by avoiding exploring executions that only differ in the order of independent instructions, as well as other optimizations.

Contents

Chapter 1

Introduction

Writing concurrent code is well-known to be difficult, with many bugs occuring due to improper synchronization. Adding insult to the injury, modern platforms employ weak memory consistency models, i.e., the instructions of each thread might be reordered by the compiler and the CPU.

To make writing concurrent code less error-prone, developers often use concurrent libraries. These libraries implement standard concurrent data structures in a thread-safe manner, and aim to achieve optimized performance by incorporating fine-grained synchronization, elaborate memory reclamation algorithms, and contention reduction mechanisms. However, many of these libraries are not formally checked for correctness, so it's possible that there are executions that lead to undesired behavior.

To combat this, we aim to formally reason about the correctness of one such library, namely libcds, by using the GenMC model checker. GenMC explores all possible states of a given program up to a bound to find states where some invariant is violated. To reduce the workload, GenMC avoids exploring states that only differ in the order in which independent instructions are executed, and is also equipped with various optimizations designed to aid in the verification of concurrent data structures.

We will first introduce libcds, the library we are aiming to model-check. We will go through its components, show how it can be used by a client program as well as its inner workings. After a short theoretical interlude giving us an understanding of the C++ weak memory consistency model, we will continue with the modeling of the library functionality. This will in particular include a new model of hazard pointers for the model checker, as well as other improvements of the tool, and ensuring an environment for the library that suits our verification effort.

Finally, we will apply our knowledge and changes to verifying some libcds data structures. This will include using the relaxed lineariziation checker

component of GenMC, giving us strong guarantees about the correctness of the verified data structures. Additionally we will be able to uncover bugs in various implementations, propose some fixes for them and then verify the corrected versions.

Chapter 2

The libcds Library

This is a library providing concurrent data structures. It is supposed to give correct, thread-safe implementations of different data structures, both lock-free and lock-based, and additionally different safe memory reclamation (SMR) mechanisms. The implemented data structures include stacks, queues, trees, sets, and more. Each of these has different usable implementations based on different published papers. The SMR algorithms included are two implementations based on Hazard Pointers [13, 14, 1], and different versions of a reclamation mechanism relying on user-space read-copy-update (URCU) synchronization [4, 6, 5]. These algorithms often rely on having their own thread-private storage that the garbage collector needs to access, so support for different threading models is also included in the library. These three parts make up the majority of the code and will be presented in the following. For any of the following content, refer to the libcds documentation [12] and the library code [11] for more details as needed.

2.1 Threading model

Since the library needs support from the threading model to provide thread-local storage (TLS) for the garbage collectors it uses, it provides interfaces for some of the most used models depending on the operating system and compiler used. These include:

- Windows TLS API
- Microsoft Visual C++ __declspec(thread)
- pthread_key_t with getspecific and setspecific
- GCC __thread
- C++ thread_local

These models have a ready-for-use manager class that implements the API required by the library using the specific model that has been chosen. This is done either through an autodetect system that checks for the existence of specific macros defined by compilers and operating systems to determine the suitable manager, or simply defined by the user. This manager class is mainly responsible for holding and managing URCU data structures that ensure correct access control. In addition, hazard pointer data like a hazard pointer pool or the records for keeping track of retired pointers also need to be accessed and have their own TLS manager in the garbage collector class.

2.2 Safe memory reclamation

In languages that have manual memory management, it is already non-trivial to ensure that a program stays inside the bounds of defined behaviour, avoiding double frees, use after frees, accessing non-allocated memory, etc. If there are multiple threads that might access the same memory, it is usually possible to prevent such behaviour with the help of locks. Since libcds implements some lock-free data structures, it is essential that there is some way of ensuring that any memory region that a thread accesses is only deallocated once every thread has finished its operation on that region. This is needed because even if a node is not part of a data structure anymore, some other thread can still hold a reference to it.

As an example, a thread could save the pointer to the top of a stack but before it completes its operation, another thread changes the data structure by advancing the top, reading its value and then deallocating it. After all this, the first read tries to continue by trying to deference its pointer which leads to a use-after-free bug. This is illustrated in the following figure. As we can see, if Thread 2 now deallocates Node 2, after which Thread 1 dereferences its local pointer, Thread 1 would access freed memory.

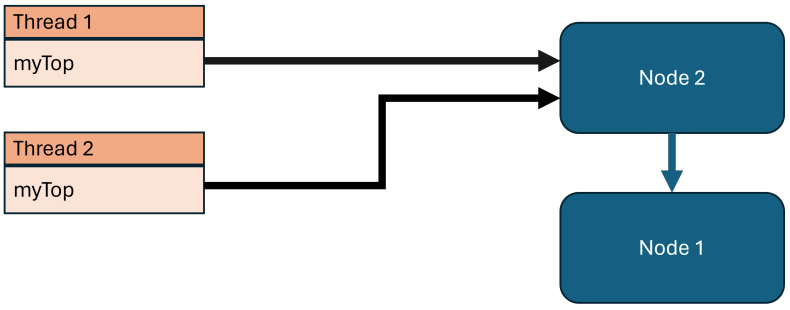

Figure 2.1: A simple stack with two concurrent accesses

An algorithm that ensures that any memory reclamation will not lead to

safety violations is called a safe memory reclamation scheme. Libcds has two different such schemes, namely read-copy-update and hazard pointers, each with a few variants with minor differences.

2.2.1 Read-copy-update

The underlying idea of this scheme is fairly simple. If a thread wants to free a memory region, for example a node in a data structure, it first logically removes it from the data structure. This means that after the removal, no new threads will be able to get a reference to the node. After the removal comes the actual deallocation, which can happen after all threads that were holding a reference to the region in a critical section of their code at the time of removal have exited said critical section. This way, every individual thread sees a valid state of the data structure and no memory is freed that could still be accessed.

The actual implementation relies on a theoretical foundation that ensures the consistency and correctness of the algorithm. Any thread that needs to ensure that a reference it is holding stays valid has to access that reference in a critical section, marked by the special functions `access_lock` and `access_unlock`. When a thread is not in a critical section, it is in a so-called *quiescent state*. When there is a period where every thread that uses RCU has been in a quiscent state, we call that a grace period. This means that any critical section that existed at the start of the grace period must come to an end before the end of that grace period. This in turn means that if a thread has removed a node from the data structure, meaning that no new threads will enter a critical section on that node, it can simply wait for a grace period, ensuring that every critical section accessing that node has finished. Then, it can deallocate the node in a sound way.

Libcds gives four variants of an RCU scheme:

Immediate reclamation This is the simplest reclamation scheme. When it wants to free a pointer, it simply waits for the end of the grace period right away. This means it blocks for any retire call until every other thread has ended the critical section it was in.

Deferred reclamation This scheme defers reclamation by putting any pointers it wants to retire into a buffer. When the buffer is full, the thread that put the last pointer in will initiate synchronization, meaning that it waits for the end of the grace period. After that, all of the accumulated pointers are freed at the same time.

Reclamation thread This scheme works similarly to the deferred reclamation scheme, but instead of the thread that filled up the queue needing to dispose

of every pointer, a special disposer thread is called to take care of freeing up the buffer.

Signal-handled, deferred reclamation This version also uses a buffer to take care of the pointers that are waiting to be freed. In addition to that, it uses the signal sending mechanism of the operating system to handle synchronization between threads.

2.2.2 Hazard pointers

The other way libcds has to ensure that memory is only reclaimed safely is to use a garbage collector utilizing hazard pointers. This scheme allocates private structures to each thread that can hold pointers, these are called *hazard pointers*. If a thread needs to rely on a pointer not being freed for some time, it can assign that pointer to its own hazard pointer structure, an operation called *protect*. Freeing a pointer is replaced by the *retire* operation. Instead of freeing the pointer straight away, the scheme scans through the hazard pointers of every thread to determine whether that pointer is currently being protected. This means that objects are only freed after no other thread is protecting references to them. In general, the hazard pointer garbage collection works with atomic pointers to enable synchronization and allow the garbage collector to see a valid and consistent state of the hazard pointers structures across threads.

The libcds hazard pointer infrastructure has two versions. The `cds::gc::HP` has a more static nature, in that it assigns a fixed number of hazard pointers to each thread. Based on the maximum number of threads and the maximum number of hazard pointers per thread, it also allocates a fixed size buffer to hold retired pointers waiting to be freed. These parameters are evaluated during the construction of the garbage collection singleton instance. The other version, `cds::gc::DHP` is a dynamic version of this. It can allocate an unlimited number of hazard pointers per thread, for an unbounded nunber of threads.

Standard hazard pointer operations

We will now look at the operations that hazard pointers use to fulfill their purpose.

Protect The `protect` operation sets the hazard pointer to the pointer we want to keep in memory to avoid another thread freeing it while we might still access it. In libcds, this looks like the following:

```
1 template <typename T, class Func>
2 T protect( atomics::atomic<T> const& toGuard, Func f )
3 {
```

```
4       assert( guard_ != nullptr );
5
6       T pCur = toGuard.load(atomics::memory_order_relaxed);
7       T pRet;
8       do {
9           pRet = pCur;
10          assign( f( pCur ));
11          pCur = toGuard.load(atomics::memory_order_acquire);
12      } while ( pRet != pCur );
13      return pCur;
14  }
```

Listing 2.1: Protect operation

Since we are operating on atomic pointers, first we need to load the value of the pointer (for additional details, see chapter 3). Next, we enter a loop that we repeat until the pointer has been assigned without changing between these operations. The `assign` is where the pointer is actually saved, and we need to make sure that it did not become invalid between the load and the assignment. Libcds also has an additional functionality to the standard hazard pointers. The protect operation has a parameter `Func f`, which is applied to the pointer before it is assigned to the hazard pointer. This allows the library to protect the internal node, even if it is at a different address as the node that holds the data (see 2.3.1).

Retire The retire operation in libcds looks like the following:

```
1  template <typename T>
2  static void retire( T * p, void( *func )( void * ))
3  {
4      hp::details::thread_data* rec = hp_implementation::tls();
5      if (!rec->retired_.push( hp::details::retired_ptr(p, func)))
6          hp_implementation::instance().scan( rec );
7  }
```

Listing 2.2: Retire operation

First, we retrieve the TLS of the current thread, this is where all the pointers that have been retired but not yet freed are saved. Next, we construct a `retired_ptr`, which holds a pointer and a function that is called when the pointer is actually freed. This custom free makes it possible to for example call the destructor of the struct being pointed to. This `retired_ptr` is then inserted in the array of retired pointers that is saved in the TLS. If this array is full after the insertion, a scan is triggered which goes through the whole array and checks whether each entry is protected by any thread. This scan has two available implementations, `classic_scan` and `inplace_scan`. The classic scan allocates extra memory to store a private copy of the protected pointers, meanwhile the inplace scan uses the least significant bit of the retired pointers to mark them as protected or not. This requires them to be

at least 2-byte aligned. Should this invariant not hold, the classic scan is executed as a fall back.

Clear This function simply removes the protection of the pointer that is currently held in the hazard pointer. This is achieved by simply assigning the null pointer to the struct holding the protected pointer.

Allocate and free Finally, the structs themselves need to be managed as well. On the abstract level, we simply allocate and free a struct that we use as a hazard pointer. In the implementation, there are a few optimizations that avoid constantly (de)allocating these structs and instead maintain a pool that stays allocated and gets linked and unlinked to be used.

Get This operation is not standard and not described in the original hazard pointer description. In libcds however, it is used in multiple data structures. Conceptually, it is very simple, namely it returns the pointer that is currently being protected by the hazard pointer.

The libcds implementation

The classes that implement the thread-local hazard pointer functionality in libcds are called `Guard` and `GuardArray`. We will focus on `Guard` first. This class implements the functions `protect` and `clear`, a helper function `assign`, as well as the non-standard functionality, the `get` operation. To achieve all this, it has an underlying private member of type `guard`. This `guard` is what holds the actual pointer that is being protected. Additionally, it has a `next_` member, which is used to keep track of a linked list of currently non-used `guard` structs to allow for fast assignment to a `Guard`. When a `Guard` is initialized, all we do is remove the head of the `guard` free list and link it to the `Guard` by assigning it to the private member `guard_`.

A `GuardArray` is the natural extension of this concept. Whenever we do not want to keep track of a bunch of `Guard` instances by individual names, we can instead instantiate a `GuardArray` that is basically the same as a standard array with the added benefit of a constructor that allocates all the entries and a destructor that frees them. It implements the same methods as `Guard` with the addition of having to specify the index of the `Guard` that we want to execute our operation. A `GuardArray` takes the size of the array as a template argument.

Here is a small example that illustrates the difference between the classes and shows how to use them in a program.

```
1 cds::gc::HP::Guard guard1;
2 cds::gc::HP::Guard guard2;
3 guard1.protect(somePointer);
```

```
4 guard2.protect(somePointer);
5
6 cds::gc::HP::GuardArray<2> guards;
7 guards.protect(0, somePointer);
8 guards.protect(1, somePointer);
```

Listing 2.3: Simple usage of `Guard` and `GuardArray`

2.3 Data structures

2.3.1 Stored data types

Libcds has two kinds of data structures available for use. The first are non-intrusive structures. These make a copy of the object being stored by allocating a new node with the required members of the data structure and copying the value the user wants to store into the node. In contrast, intrusive data structures store the objects themselves which thus must satisfy some requirements. In libcds, there are three different ways to achieve this:

Base hook This is the most straightforward option. The data type that will be stored inherits from a base node type defined by the data structure and simply adds fields to store additional data.

Member hook In this version, the node type of the data structure is a member of the type stored in the structure. This hook type additionally requires the user to specify the offset of the node member in the struct to allow the data structure to access it.

Traits hook A traits hook is slightly more involved. To use this option, the user needs to provide a struct that has methods to convert between the type that is inserted into the data structure and the node type that holds the information that the data structure needs to manage its nodes.

Since the non-intrusive data structures internally rely on their intrusive counterparts with a layer that allocates and manages nodes with the base hook method, for the rest of the thesis, we will focus on the intrusive containers.

2.3.2 Traits

Since this is a general purpose library, the user can specify some additional features of each data structure. While these features vary with the different types of structures and implementations, here is an overview of some that most have in common, with the usual default noted as well.

Hook

This is the hook from the previous section. It specifies the type of hook used. This hook type need fit the data type that we want to store. If we use the traits hook, we also need to make sure that we provide the node traits that tell the library how to access the data it needs. **Default:** Base hook

In the following example, we can see how we take the default traits of a stack implementation by inheriting from the default traits (1) and then redefining the hook trait as the member hook explained in the previous section (2). We can also see how the type we want to insert into the stack (`myData`) has the required node type as one of its members (3). We then use the `offsetof` macro to tell libcds where to find that node in our custom type (4).

```
1  struct myData
2  {
3      int value;
4      cds::intrusive::single_link::node<cds::gc::HP> node; //3
5      myData(int v) : value(v) {}
6  };
7
8  struct myTraits : cds::intrusive::treiber_stack::traits //1
9  {
10     typedef cds::intrusive::treiber_stack::member_hook<
11         offsetof(myData, node), //4
12         cds::opt::gc<gc>
13     > hook; //2
14 };
```

Listing 2.4: Example usage of a member hook

Allocator/Disposer

While intrusive data structures usually do not deallocate the objects that are stored, since the user that inserted it can still hold a reference to them, it is sometimes still needed. A good example is the *clear* operation that most data structures have. Since it removes the nodes from the data structure but does not return a reference to them, there is no thread that could dispose them. This means that the data structure itself needs to call the disposer to make sure there are no memory leaks. Non-intrusive data structures additionally use the allocator to allocate new nodes to insert into the data structure and to (with the help of the SMR scheme, safely) dispose them when they are removed. This is safe, since the client threads do not have access to the internal nodes. **Default:** empty disposer, which does nothing

In the following example, we can see how a custom disposer can be implemented. It needs to define `operator()` that is applied when a node should be disposed of. This specific implementation simply prints the address of the node and then deletes it.

```
struct printDisposer {
    void operator()(myData* p) const {
        if (p)
            printf("deleting %p\n", p);
        delete p;
    }
};
```

Listing 2.5: Custom disposer

Item counter

This option is used to enable counting the number of the items currently stored. The options are `empty_counter` (always returns zero), `item_counter` (real counter), and `cache_friendly_item_counter` (real counter with additional padding to prevent false sharing). **Default:** empty counter

All of the counters need to support pre- and postincrement, returning the current value, and de- and increasing the current value by a specific amount. This is part of the empty item counter class, which simply always returns zero.

```
class empty_item_counter {
public:
    typedef size_t counter_type;
public:
    /// Returns current value of the counter
    static counter_type value()
    {
        return 0;
    }

    counter_type operator ++() const
    {
        return 0;
    }

    ...
}
```

Listing 2.6: Empty item counter

In contrast, here is the standard item counter class, which uses an underlying atomic variable to keep track of the current number. In the real implementation, the user can also specify the memory order of the atomic load and read-modify-write operations (for additional details, see chapter 3).

```
class item_counter
{
private:
    std::atomic<size_t>     m_Counter;
```

```
5
6  public:
7      /// Returns current value of the counter
8      size_t value()
9      {
10         return m_Counter.load();
11     }
12
13     /// Increments the counter. Semantics: postincrement
14     size_t inc()
15     {
16         return m_Counter.fetch_add(1);
17     }
18
19     ...
20 }
```

Listing 2.7: Standard item counter

Memory model

Allows the user to select between having every atomic operation constrained to the sequentially consistent C++ memory ordering `memory_order_seq_cst` and having them relaxed as much as intended by the implementation. Again, see 3 for more details. **Default:** relaxed ordering

Stat

This allows the user to get statistics about the data structure. This is very dependent on the specific structure and implementation but might count the number of times items were inserted, removed, how often some operations failed or collided with each other, etc. The options here are `empty_stat` (every update is a no-op), `stat` (standard statistics about the structure), or even a custom implementation that implements the interface used by the data structure. **Default:** empty stat

Any custom statistics implementation needs to implement the same methods as the default empty stat struct. Here is the stat struct for a queue implementation referred to as Michael & Scott's queue [15]. The template argument in this example allows the user to define how strictly the count events are ordered, it can also be as simple as `int`.

```
1 template <typename Counter = cds::atomicity::event_counter >
2 struct stat
3 {
4     typedef Counter     counter_type;
5
6     counter_type m_EnqueueCount;
7     counter_type m_DequeueCount;
8     counter_type m_EnqueueRace;
```

```
9       counter_type m_DequeueRace;
10      counter_type m_AdvanceTailError;
11      counter_type m_BadTail;
12      counter_type m_EmptyDequeue;
13
14      void onEnqueue()                { ++m_EnqueueCount; }
15      void onDequeue()                { ++m_DequeueCount; }
16      void onEnqueueRace()            { ++m_EnqueueRace; }
17      void onDequeueRace()            { ++m_DequeueRace; }
18      void onAdvanceTailFailed()      { ++m_AdvanceTailError; }
19      void onBadTail()                { ++m_BadTail; }
20      void onEmptyDequeue()           { ++m_EmptyDequeue; }
21
22      void reset()
23      {
24          m_EnqueueCount.reset();
25          m_DequeueCount.reset();
26          m_EnqueueRace.reset();
27          m_DequeueRace.reset();
28          m_AdvanceTailError.reset();
29          m_BadTail.reset();
30          m_EmptyDequeue.reset();
31      }
32  }
```

Listing 2.8: Default stat struct of Michael & Scott's queue datastructure

Backoff

When a thread detects that there is contention on a resource, it can optionally wait some time to allow other threads to make progress without interfering with each others operation as much. There are several options, like calling `yield`, constant backoff, exponential backoff, or calling processor-specific backoff or pause hints. The backoff can also be interrupted early if some predicate returns true, allowing the implementation of elimination (see *Elimination*, 2.3.2). **Default:** depends on data structure

A simple example of this is the yield backoff. It simply tells the thread implementation to schedule a different thread. While this is stateless, note that there can be stateful backoff strategies. An example of that is the exponential backoff, which waits a different span of time depending on how many times it has been called already. This is why we need the `reset` method, so the state does not accumulate when we do not want it to.

```
1   struct yield
2   {
3       void operator ()()
4       {
5           std::this_thread::yield();
6       }
7
```

```
8       template <typename Predicate>
9       bool operator()(Predicate pr)
10      {
11          if ( pr())
12              return true;
13          operator()();
14          return false;
15      }
16
17      static void reset(){}
18  }
```

Listing 2.9: Yield backoff strategy

Elimination

This feature is not found in many data structures, only in stacks where inserting and removing elements happens at the same place and an implementation of a queue where multiple requests are combined by a single thread (flat combining queue). This mechanism tries to eliminate pairs of operations that together do not change the state of the data structure. In a stack, this would mean a pair of a `push` and a `pop`, since we can simply give the item of the thread that is inserting to the one that is popping. This requires additional infrastructure where the threads can meet each other and synchronize the elimination itself. This is done in an array of structs with atomic members that coordinate an elimination. When a thread detects contention on the data structure and is still waiting for its backoff period to be over, it selects a slot in the array non-deterministically. If another thread with an opposing operation selects the same slot at the same time, they can exchange values and do not have to change the state of the stack at all. **Default:** disabled

How is this used?

To bring all these aspects together, here is a real definition with all of the traits we defined above being used in our custom variant of a Treiber stack [16]. The template arguments of the final type are the garbage collector, the data type we want the stack to operate on, and the stack traits we defined.

```
1   typedef cds::gc::HP hpgc;
2
3   struct myData : public cds::intrusive::single_link::node<hpgc>
4   {
5       int value;
6       myData(int v) : value(v) {}
7   };
8
9   struct printDisposer {
10      void operator()(myData* p) const {
11          if (p)
```

```
12              printf("deleting %p\n", p);
13          delete p;
14      }
15  };
16
17  struct myTraits : public cds::intrusive::treiber_stack::traits
18  {
19      typedef cds::intrusive::treiber_stack::base_hook< cds::opt::
        gc<hpgc> > hook;
20      typedef printDisposer disposer;
21      typedef cds::atomicity::item_counter item_counter;
22      typedef cds::opt::v::relaxed_ordering memory_model;
23      typedef cds::intrusive::treiber_stack::stat<int> stat;
24      typedef cds::backoff::exponential<> elimination_backoff;
25  };
26
27  typedef struct cds::intrusive::TreiberStack <
28      cds::gc::HP,
29      myData,
30      myTraits
31  > myStack;
```

Listing 2.10: Our very own Treiber stack

2.4 A libcds program from start to finish

A program that wants to use libcds data structures and garbage collection schemes needs to follow a specific pattern.

```
1  int main(int argc, char** argv)
2  {
3      cds::Initialize();
4      {
5          cds::gc::HP hpGC;
6          cds::threading::Manager::attachThread();
7          // use libcds data structures
8      }
9      cds::Terminate();
10 }
```

Listing 2.11: Minimal libcds program

First, the library infrastructure must be initialized, which needs to happen before any SMR scheme or data structure is initialized and is done by the `cds::Initialize` function. This function first tries to find out about the topology of the system, namely how many logical cores there are available to use. Depending on the OS, some other data structures are initialized to keep track of the topology, with Linux, this is a no-op. After that, it initializes the manager (see 2.1). This depends on the threading model, the pthread-based model for example creates a `pthread_key_t` key that the threads then can use to retrieve their thread-local data.

Next, the garbage collection object is constructed. This follows the singleton pattern, there is a single static instance of the SMR object, which is allocated and initialized in the constructor of the `cds::gc::HP` class. This constructor sets some parameters that new threads can use to construct their own thread-local data for the garbage collection algorithms. These parameters are the number of hazard pointers a thread can have (default 8), the maximum number of threads that the program will use (default 100), the maximum number of pointers that can be retired at the same time (default $2 \cdot 100 \cdot 8 = 1600$) and the specific algorithm used to scan through the retired pointers to decide which can be freed. Finally it initializes a list where every thread inserts their thread-local data, so that the retired pointer scan (see 2.2.2) can access them.

With the core infrastucture and the garbage collection having been initialized, we also need to tell libcds about our thread. This is done by calling the `cds::threading::Manager::attachThread()` function. This, again, differs manager to manager, if pthread is used, it allocates a new `ThreadData` and saves it with the pthread key it created earlier. With the allocation of the `ThreadData` object, the thread will check for the existence of each type of SMR garbage collector and allocate a `thread_record` object, which is needed for each type that is in use. This allocation can mean the reuse of an object that has been allocated before but is currently unused, or the creation of a brand new object. For hazard pointers, this `thread_record` sets the owner of the record and inserts it into the linked list that the garbage collector singleton maintains. In addition, the record contains the data structures that the hazard pointer scheme requires, namely a linked list of hazard pointers (see 2.2.2) that the thread can use later, an array for the retired pointers, and two pointers to the location of these two data structures.

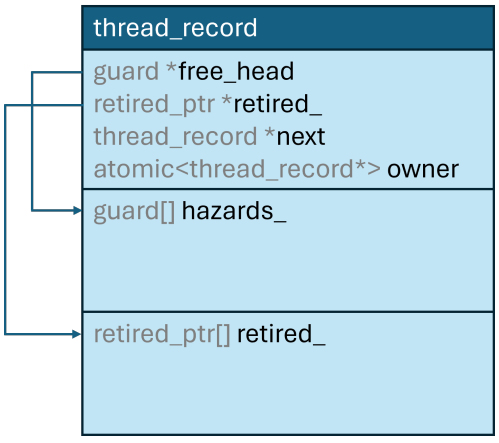

Figure 2.2: Thread record structure

During the initialization of this record, we not only allocate the memory for the hazard pointers, but actually initialize them all as well. While this may take some extra time in the beginning, we can later simply link these guards when a thread needs them. This means that when we need to "allocate" a hazard pointer, we simply set a pointer to the head of the free list and advance the head. This makes this process faster and keeps all the guards together. To see how such a guard is used, refer to 2.2.2.

With these steps taken care of, we can use the data structures and garbage collection provided by libcds, as long as we attach every thread that wants to do this. After we are done, we need to reverse the steps we have taken to clean up the library data structures properly.

When any thread that is not the main thread has finished its work, we also need to detach it. This clears all the hazard pointers of the thread to avoid any protection persisting when it is no longer needed. After that, the owner of the `thread_record` is set to `nullptr`. Note that the record stays in the linked list, but it is ignored by operations that use it. Finally, the manager is reset. With the pthread model, the TLS associated with the thread is set to `nullptr` and the thread data that was allocated is deleted.

When, in the main thread, the garbage collection singleton goes out of scope, its destructor is called. This first detaches all threads by going through each `thread_record` where the owner is set, clearing all hazard pointers, and resetting the owners to `nullptr`. Next, it destructs the SMR singleton itself by going through the entire linked list of `thread_record` objects and freeing any remaining retired pointers, as well as deallocating the records themselves. In the end, the memory occupied by the singleton itself is freed.

At the end, all that is left is to call `cds::Terminate`, which terminates the manager used. With the pthread model, this means deleting the key that was created in the beginning, as well as detaching any thread that is still attached and thus has data associated with the key.

Chapter 3

C++ Weak Memory Consistency Model

While not every detail of the C++ weak memory consistency model is relevant for libcds, we need to understand its usage and consequences for the data structures and memory reclamation schemes, more specifically, the usage of atomic variables and the different memory orderings used to access them.

In the C++ weak memory consistency model, stores and loads to atomic variables can have different degrees of strictness, which are governing the guarantees for which memory access operations become visible to other threads at what point. For our purposes, the difference boils down to the following: If the effect of an atomic operation with some ordering is visible to another, how can the memory accesses around the pair be reordered? Here are the possible memory orders that we can specify for an atomic memory access instruction:

- `memory_order_relaxed`
- `memory_order_consume`
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel`
- `memory_order_sqt_cst`

We will go through each to understand what they allow and what they do not. In general, these orderings in the list go from weaker to stronger guarantees that the compiler and the hardware have to respect. Having weak guarantees means that the program can be optimized more but sometimes we need stronger guarantees to ensure the correctness of the program. This can cost performance, so we always want the weakest but still correct ordering.

memory_order_relaxed This ordering can be applied to loads, stores and read-modify-write operations. A relaxed memory order does not impose any restrictions on memory accesses around the atomic instruction. Memory accesses from before the atomic instruction can be moved after it and the other way around. The only guarantee we have is that the operation is atomic, meaning that other threads can not see intermediate states, either they see the state of the variable before or after the operation. Of course, this goes for all orderings.

memory_order_consume A consume memory order is valid for atomic loads and the load part of read-modify-write operations. It guarantees that any later operations which are dependent on the value being loaded stay after the load operation. Note that this ordering is under review and will be deprecated in C++26 [2]. It is not used anywhere in libcds.

memory_order_acquire An acquire memory order applies to atomic loads and the load part of read-modify-write operations as well. It is stronger than the consume ordering in that any memory access after the acquire will stay after it, not only the ones dependent on the result of the load.

memory_order_release A release memory order applies to atomic stores and the store part of read-modify-write operations. It guarantees that any memory accesses before the atomic store stay before it. This implies that if another thread sees the result of the store, it will also see everything that happened before the store as well.

memory_order_acq_rel An acquire-release ordering applies to atomic read-modify-write operations. The load part of the operation will have acquire, while the store will have release semantics. This means that memory accesses can not cross this instrucion in any direction.

memory_order_seq_cst A sequential consistency ordering applies to loads, stores and read-modify-write operations. It gives loads an acquire, stores a release, and read-modify-writes an acquire-release ordering. In addition to this, it ensures a single global ordering of the visibility of sequential consistency atomic accesses with this ordering, meaning that every thread is guaranteed to see the same order of these operations.

While these orderings are defined for single operations, the effect they have is useful for pairs of them where one sees the result of the other. A load needs to see the result of a store for these orderings to be applied, otherwise the threads have no synchronization at all. While this is implicitly contained in the definitions above, we need to highlight the concept of a release-acquire

pair. If an acquire load reads the result of a release store, any operations around them can not cross this fence-like construct.

We will show the relevant effects with the help of a small code example. The following program shows two threads accessing two global variables, attempting to snychronize with each other. This pattern is ubiquitous in lock-free programming. We can imagine `thread1` calculating something and, when it is done, signaling to `thread2` that it can now use the results of the calculation. While in this example, we simply use an if statement to model `thread2` seeing the flag being set, in general it might be spinning in a loop or even waiting for an interrupt.

```
1  int x = 0;
2  std::atomic<int> flag = 0;
3
4  void *thread1(void *param)
5  {
6      x = 42;
7      flag.store(1, std::memory_order_relaxed);
8      return NULL;
9  }
10
11 void *thread2(void *param)
12 {
13     if (flag.load(std::memory_order_relaxed) == 1) {
14         assert(x == 42);
15     }
16     return NULL;
17 }
```

Listing 3.1: Relaxed memory order pair

In listing 3.1, both the store and the load operation are relaxed. When we are inside of the if statement in `thread2`, we have seen the result of the store in `thread1`. Still, the assertion can be violated. The read of `x` in the assert statement can be moved to happen before the write in `thread1` or the other way around. Verifiying this program with GenMC would rightefully tell us that there is a safety violation.

```
1  int x = 0;
2  std::atomic<int> flag = 0;
3
4  void *thread1(void *param)
5  {
6      x = 42;
7      flag.store(1, std::memory_order_release);
8      return NULL;
9  }
10
11 void *thread2(void *param)
12 {
13     if (flag.load(std::memory_order_acquire) == 1) {
```

```
14          assert(x == 42);
15      }
16      return NULL;
17  }
```

Listing 3.2: Release-acquire pairing

In listing 3.2, the situation is different. Since `thread1` stores to the `flag` variable with release ordering and the result is visible to `thread2`, which loads from it with acquire ordering, we have a valid synchronizing pair of instructions. Since we can neither move the read in the assert before the load of the flag, nor move the write of `x` in `thread1` after the store of the flag, it is guaranteed that the read will see the result of the write and the assertion will hold. Note that these are the weakest possible orderings that ensure that the assertion holds. Changing any to a weaker one would break the correctness of the program. In turn, we could strengthen either of them to the sequentially consistent ordering and preserve correctness but possibly lose out on performance.

Chapter 4

Modeling libcds for Verification with GenMC

4.1 Overview

First, we will give an overview of the verification process of a program using GenMC so that we can contextualize the changes and improvements that we carried out. GenMC, which stands for Generic Model Checker, is a stateless model checker, enumerating all executions of a program without explicitly storing the visited states. Since this quickly leads to a state space explosion, it employs a range of optimizations to reduce the number of executions explored in a sound way. The checker can verify bounded client programs that use concurrency. The input we can give it can be a C or C++ program, where concurrency is achieved with the `pthread` standard library. It can check for assertion violations, data races, illegal memory accesses and other such errors. It is generic with respect to the memory model that the checked program complies with.

The verification of a program starts with the compilation to the LLVM IR, on which the actual verification is carried out. After that, a range of transformations and passes are carried out. These include things like unrolling loops, inserting internal function calls or carrying out annotations. Most of these serve the purpose of making the verification more efficient or collecting data about the program that can be used later.

Then, comes the actual verification. This requires three parts that all coordinate during the process. These are the *Interpreter*, the *Driver*, and the *Execution graph*.

Execution graph First, we have the execution graph into which events can be inserted to record the current execution being explored. These events hold information specific to the instruction that generated them, a *write* event may

for example store the value written, the address written to, the dependencies of the instruction, etc.

Interpreter The interpreter executes the program instruction by instruction. For every LLVM instruction or internal function call, it has implemented a specific visitor. For every instruction type, it reads out the information from the instruction and, if the event is visible, notifies the driver. To achieve this, it creates an event and invokes a handler in the driver that can modify and insert it into the execution graph.

Driver Based on the information that is provided by the interpreter or stored in the execution graph, the driver can calculate return values, schedule revisits to instructions dependent on the current one or block entire executions that are not needed for a correct result. Additionally it is the task of the driver to populate the execution graph with the events that it received from the interpreter.

Based on this process, the verifier will either output a success message or give a trace corresponding to the error it found. This trace contains every event that is required to happen for the error to occur. It takes the form of an graph, where we can see the trace of the instructions explored through the program order and the reads-from relation. Note that this trace does not show a single total order on all instructions but rather a partial order based on the interleaving of dependent instructions. In addition, the verifier will look for things like memory leaks that are not necessarily errors and warn the user with a trace showing a full execution of the program where some allocated memory was left unfreed.

A small example verification We will include a small example that shows us how to interpret the results of a verification. The program itself is trivial, so the main thing we focus on is how to read the results. For more complex examples, refer to 4.15, 5.2 and A.1.

```
1 std::atomic<int> x;
2 int y;
3
4 int main()
5 {
6     x.store(42);
7     y = 42;
8     assert(x.load(std::memory_order_acquire) == 0);
9 }
```

Listing 4.1: Simple program

```
1 *** Compilation complete.
2 *** Transformation complete.
```

```
3  Error: Safety violation!
4  Event (0, 3) in graph:
5  <-1, 0> main:
6          (0, 1): Wsc (x..__a_, 42) atomic:949
7          (0, 2): Wna (y, 42) L.10
8          (0, 3): Racq (x..__a_, 42) [(0, 1)] atomic:957
9
10 Assertion violation: x.load(std::memory_order_acquire) == 0
11 *** Verification unsuccessful.
12 Number of complete executions explored: 1
13 Total wall-clock time: 0.05s
```

Listing 4.2: The output of verifying listing 4.1

As we can see, the verifier first outputs some status messages confirming that the compilation and code transformation are done. After that, it tries to verify our program and, of course, finds the assertion violation at line 8. Then, it shows the trace that leads to the error by printing the trace of the execution it was currently exploring when it found the error. A few things to note in the graph:

Memory ordering First, for each write and read event, marked with W/R, we can see the memory ordering of the access. Event `(0,1): Wsc`, was a sequential consistency store (the default in the <atomic> library), then comes a non-atomic store (`(0,2): Wna`) and finally an acquire read (`(0,3): Racq`).

Memory accessed Then, in the parentheses, we can see what memory was accessed and what the value written/read was. In the accesses of the atomic variable, we see the artifacts of the implementation of an atomic variable, namely that `x` is actually a struct containing another anonymous struct, containing another anonymous struct, containing the actual value of 42 in the member `__a_`.

Reads-from The final thing to note can be seen for every read event, namely the write event that they read from, supplied in the brackets. This helps establish the partial order that leads to the error. In our case, we can see that the read event (0,3) read its value from (0,1). For each read event, the possible options are either some other event like above, the special `[INIT]` event containing the initial values of each variable, or `[BOTTOM]`, denoting a read that accessed uninitialized or freed memory.

Error location While the error for an assertion violation does not refer to the graph, others like accessing freed or uninitialized memory, or data races will tell the user where the events corresponding to the error lie in the graph.

For example, a data race error will show the two instructions that access the same variable at the same time.

4.2 Hazard pointers in GenMC

Now that we have a basic understanding of how GenMC works, we will turn our attention to verifying programs that use the libcds library by going through the modeling of the library as well as the changes we made to GenMC to accomodate the model. We will start with the hazard pointer safe reclamation scheme, as described in 2.2.2. We choose the hazard pointer safe reclamation scheme, as it already has an implementation in GenMC. In contrast, read-copy-update does not and adding a new SMR model from scratch is out of scope for this thesis.

As we have seen, hazard pointers are heavily used in libcds data structures and are the garbage collection mechanism we will focus on for this thesis. Since GenMC already had an implementation of hazard pointers, we did not have to start from scratch. This implementation worked purely on the execution graph that GenMC builds as the interpreter executes the program [9]. Here is a brief overview of how each hazard pointer operation worked:

Allocate The allocation of a hazard pointer corresponded to a simple malloc event in the graph that allocated a struct of type `__VERIFIER_hp_t`, which had a single unused dummy member. As with all malloc events, data such as the address of the memory region returned, its size, alignment, etc. were saved.

```
1 typedef struct { void * __dummy; } __VERIFIER_hp_t;
```

Listing 4.3: The hazard pointer struct type definition

Free The deallocation of the hazard pointer was exactly the opposite. It freed the `__VERIFIER_hp_t` struct and with that added a free event to the graph, saving the freed address and the corresponding allocation label.

Protect This operation first loaded the actual pointer that the user wanted to protect from the atomic variable that held it and then added a label called `HpProtectLabel` to the graph, saving the address of the hazard pointer that was utilized, as well as the pointer that was being protected.

Clear The clear operation just adds a `WriteLabel` to the graph, which contains the address of the hazard pointer being cleared, with the value stored being zero. Having this operation correspond to an actual write event was unnecessary and only a remnant of a prior implementation.

Retire This again is modelled as its own event `HpRetireLabel`, which inherits from `FreeLabel`, saving the same information. The different consistency checkers can then use these events, together with the `HpProtectLabel` to determine whether any later accesses to the memory location should be marked as an access to freed memory in case there was no valid protection at that time or let it go though as valid if there is still a valid hazard pointer protection.

What is missing ? With a quick comparison of the GenMC hazard pointers to the libcds hazard pointer API (see 2.2.2), we can clearly see the missing operation that the GenMC model did not implement, namely the `get`, which retrieves the current pointer being protected.

4.3 Changing Hazard Pointer Model

While the information about the currently protected pointer of a given hazard pointer is saved in the execution graph, it would be cumbersome to retrieve it for the get operation, since these events are not explicitly tied together. To circumvent this program, we will instead save the protected pointer in the actual program being verified. This resembles the usage of hazard pointers in the program and allows us to retrieve the value directly by the reference that the program itself uses. Since we already allocate a struct with a dummy member, we can repurpose that member to save the pointer being protected. However, this changes the way the interpreter tells the driver about the `protect` that is happening. Instead of the internal function that was used before, we have turned the `protect` into a normal write. To achieve the same functionality as before, we need to tell the interpreter that this write is special and requires different handling. We do this by annotating the write with a special value that the interpreter can check for when it comes across the instruction. This way, it can recognize the write as belonging to a hazard pointer protect operation and add the special `HpProtectLabel` to the graph that it was using before.

4.3.1 GenMC's annotations

Since some operations need to carry more information than what is usually contained in an instruction, GenMC has the option to add annotations to these instructions. This is exactly the case with the hazard pointer protect operation. While the only thing happening in the code is a simple store to some location, the interpreter needs the additional context of it being a hazard pointer protect operation so that it is able to notify the driver that a hazard pointer protect has taken place. This system is built on top of the LLVM metadata that each instruction carries with itself. When the

LLVM code undergoes the different transformations that GenMC carries out, the `EliminateAnnotationsPass` finds out which intructions need additional metadata (see 4.9) and adds it to these instructions under the type "genmc_attr". This metadata consist of a 64-bit bit vector, where each bit corresponds to some pre-defined property. In general, the *Kind* annotations convey a unique identity and are as such exclusive to each other, while the *Attr* ones represent properties that can all be set at the same time, as well as mixed with a *Kind*. Note that these values are not supposed to be used directly by a GenMC user during verification.

```
#define GENMC_ATTR_LOCAL 0x00000001
#define GENMC_ATTR_FINAL 0x00000002

#define GENMC_KIND_NONVR    0x00010000
#define GENMC_KIND_HELPED   0x00020000
#define GENMC_KIND_HELPING  0x00040000
#define GENMC_KIND_SPECUL   0x00080000
#define GENMC_KIND_CONFIRM  0x00100000
#define GENMC_KIND_PLOCK    0x00200000
#define GENMC_KIND_BARRIER  0x00400000
#define GENMC_KIND_HPSTORE  0x00800000
```

Listing 4.4: GenMC's current annotation values

These annotation values are bit-packed together to allow an instruction to have different annotations at the same time. Below is the function that adds the annotations to an instruction. It takes as input an instruction to annotate, the type of annotation (for this purpose "genmc_attr"), and the annotation value we want to add. It first gets the metadata saved under the type (1). If there is already some data, it combines it with the new value with a bitwise `OR` (2). Then it saves the resulting bit vector in an `llvm::MDNode` (3), which is the actual type being stored, and finally sets the metadata of the instruction to this new value (4).

```
void annotateInstruction(llvm::Instruction *i, const std::string
    &type, uint64_t value)
{
  auto &ctx = i->getContext();
  auto *md = i->getMetadata(type); /*(1)*/

  uint64_t mValue = value;
  if (md) {
    auto old = dyn_cast<ConstantInt>(
            dyn_cast<ConstantAsMetadata>(md->getOperand(0))->
    getValue())
            ->getZExtValue();
    mValue |= old; /*(2)*/
  }

  auto *node =
```

```
15      MDNode::get(ctx, ConstantAsMetadata::get(ConstantInt::get(ctx
        , APInt(64, mValue)))); /*(3)*/
16    i->setMetadata(type, node); /*(4)*/
17    return;
18  }
```

Listing 4.5: The instruction annotation function

This is how we add the annotations we want to the instructions that need it. The next step is in the interpreter, since it needs to do different things based on the annotations that the instruction which it is currently visiting has. Extracting the annotation of an instruction is very similar to adding it. Below is the function that returns the correct annotation for store instructions. First, we again get the metadata (1). If there is none, the store being examined is a normal write, so we return that (2). Next we access the value of the metadata and cast it back to a `uint64_t` with the help of LLVM helper functions (3). Since we only care about the *Kind* of instruction and not the *Attr* (see 4.3.1), we mask the corresponding bits with the `GENMC_KIND` macro (4). Since all the kinds are exclusive, we can just use a switch statement to return the right result (5).

```
1  EventLabel::EventLabelKind getWriteKind(llvm::Instruction &I)
2  {
3    using Kind = EventLabel::EventLabelKind;
4    auto *md = I.getMetadata("genmc.attr"); /*(1)*/
5
6    if (!md)
7      return Kind::Write; /*(2)*/
8
9    auto *op = llvm::dyn_cast<llvm::ConstantAsMetadata>(md->
       getOperand(0));
10   BUG_ON(!op);
11
12   auto flags = llvm::dyn_cast<llvm::ConstantInt>(op->getValue())
       ->getZExtValue(); /*(3)*/
13
14   switch (GENMC_KIND(flags) /*(4)*/){
15     case (GENMC_KIND_HPSTORE):
16       return Kind::HpProtect; /*(5)*/
17     case (0x0):
18       return Kind::Write; /*(5)*/
19     default:
20       BUG();
21   }
22 }
```

Listing 4.6: Extracting the annotation of an instruction

To round it out, we use the function we defined above to create the correct label for our annotated instruction and then invoke the store handler in the driver:

```
switch (getWriteKind(I)){
    case Kind::Write:
      CALL_DRIVER(handleStore, WriteLabel::create(...));
    case Kind::HpProtect:
      CALL_DRIVER(handleStore, HpProtectLabel::create(...));
}
```

Listing 4.7: Interpreter code calling the driver

4.3.2 Changes to the protect operation

Below, you can see a comparison of the old (4.8) and the new (4.9) implementation of the hazard pointer protect operation. Before, the implementation used an internal function call, from which the hazard pointer's address and the protected pointer were extracted by the interpreter. Now, the protect uses a store into a real hazard pointer struct member to save the pointer being protected and uses the GenMC annotation functionality to mark it as a hazard pointer protect for the interpreter.

```
#define __VERIFIER_hp_protect(hp, p)                                  \
({                                                                    \
  void *_p_ = __atomic_load_n((void **) p, __ATOMIC_ACQUIRE);         \
  __VERIFIER_hazptr_protect(hp, _p_);                                 \
  _p_;                                                                \
})
```

Listing 4.8: Old hazard pointer protect

```
#define __VERIFIER_hp_protect(hp, p)                                  \
({                                                                    \
  void *_p_ = __atomic_load_n((void **) p, __ATOMIC_ACQUIRE);         \
  __VERIFIER_annotate_begin(GENMC_KIND_HPSTORE);                      \
  __atomic_store_n(&(hp->__private), _p_);  \
  __VERIFIER_annotate_end(GENMC_KIND_HPSTORE);                        \
  _p_;                                                                \
})
```

Listing 4.9: New hazard pointer protect

A last detail An attentive reader might have realised that the hazard pointer protect functionality of libcds (see 2.2.2) is not fully covered by this new implementation. Namely, in libcds, the user can specify a function that is applied to the pointer that has been loaded, before the result is actually assigned to the hazard pointer itself. In our version, the load of the pointer is followed directly by the store to the hazard pointer struct. To remedy this, we split the functionality of our protect as well, replacing the store of the pointer by another macro, which we call `__VERIFIER_hp_assign`. For existing users of

the __VERIFIER_hp_protect macro, nothing changes. However, we can adjust the call in the library to execute the load and apply the function before calling our new assign operation. This changes the library fucntion to the code seen in listing 4.10, where one can see the split of the pointer load (1) and how the result of the function application of the parameter `f` to the pointer is assigned to the hazard pointer (2). The definition of __VERIFIER_hp_assign can be seen in listing 4.11.

```
template <typename T, class Func>
T protect( atomics::atomic<T> const& toGuard, Func f )
{
    assert( guard_ != nullptr );
    T pCur = toGuard.load(atomics::memory_order_relaxed); /*(1)*/
    T pRet;
    do {
        pRet = pCur;
        __VERIFIER_hp_assign(guard_, f(pCur)); /*(2)*/
        pCur = toGuard.load(atomics::memory_order_acquire);
    } while ( pRet != pCur );
    return pCur;
                }
```

Listing 4.10: New libcds hazard pointer protect

```
#define __VERIFIER_hp_assign(hp, _p_)                          \
  __VERIFIER_annotate_begin(GENMC_KIND_HPSTORE);               \
    __VERIFIER_annotate_begin(GENMC_ATTR_LOCAL);               \
  __atomic_store_n(&(hp->__private), _p_, __ATOMIC_RELAXED);   \
  __VERIFIER_annotate_end(GENMC_ATTR_LOCAL);                   \
  __VERIFIER_annotate_end(GENMC_KIND_HPSTORE);
```

Listing 4.11: __VERIFIER_hp_assign definition

Note that we have added an additional annotation and changed the memory ordering to relaxed. These changes are sound, since hazard pointers in the model are thread-private, meaning that other threads do not need to see their state. It is important to note that this only applies to the model, since the protection information is evaluated in the execution graph based on the events, not the values of the hazard pointer struct members. In a real hazard pointer implementation, other threads can access the hazard pointers so that they can find out which pointers are safe to be freed. With these changes, the verifier is able to optimize away the exploration of some unneeded executions. Using the new model leads to the exact same number of executions explored, executions blocked and hints checked as before, exactly modeling the old system.

4.3.3 Changes to the clear operation

As mentioned in section 4.2, in the old model, the interpreter simply created a write event with the written value set to zero. Since we now model the

hazard pointer as actually having a value saved in the allocated struct, we need to also change the clear operation, since it also changes that value. Just like with the hazard pointer write operation, we will insert a store instruction that writes the value zero to the hazard pointer. To avoid a duplicate event in the execution graph, we will delete the internal function call that was there before, since the interpreter will insert the write event upon seeing the newly inserted instruction and does not need to be told to do that explicitly anymore.

4.3.4 Implementing `get`

Now that we have changed the implementation of the model of hazard pointers in GenMC, we can easily add the hazard pointer get operation that was incompatible with the previous implementation. All we need to do is retrieve the pointer from the hazard pointer struct and return it, as can be seen in listing 4.12. Note once again that the relaxed ordering of the atomic store is sound as discussed in section 4.3.2.

```
1 #define __VERIFIER_hp_get(hp)                                          \
2 ({                                                                     \
3  void *_p_ =__atomic_load_n(&(hp->__private), __ATOMIC_RELAXED);\
4  _p_;                                                                 \
5 })
```

Listing 4.12: New hazard pointer get macro

4.4 Adjusting libcds to use verifier model

Now that we have covered the GenMC side of hazard pointers, it is time to return to libcds. As we have seen in listing 4.10, we can not simply replace every protect operation in libcds with a GenMC protect and be done. As it turns out, splitting the functionality of protect into a separate load and assign has another benefit. The assignment of a pointer to the hazard pointer as used in the protect operation is not just some internal method, but indeed an operation exposed to the rest of the library and extensively used in different data structures.

Going through the operations of the libcds hazard pointer implementation, we see that allocating, freeing, assigning to, and clearing a hazard pointer translates cleanly into simply calling the GenMC implementation with the same arguments and using the results in exactly the same way as before. The two operations that we changed are the protect and the retire.

As we have seen, using the separate load and assign models the implementation of the libcds protect accurately. However, there is a downside to keeping the rest of the code unchanged. The libcds version needs to assign the pointer

in a loop until it succeeds without another thread interfering. When we model such an execution, there are two possibilities. Either the assignment succeeds before we get to the bound of our loop unrolling in the model checking and we continue, or it does not and the thread is blocked and the execution is disregarded. This makes it evident that in any verified execution, the assignment succeeds. Because of this, we can skip the loop entirely and simply assign the pointer to the hazard pointer as follows:

```
1 template <typename T, class Func>
2 T protect( atomics::atomic<T> const& toGuard, Func f )
3 {
4     assert( guard_ != nullptr );
5     T pCur = toGuard.load(atomics::memory_order_acquire);
6     __VERIFIER_hp_assign(guard_, f(pCur));
7     return pCur;
8 }
```

Listing 4.13: Final version of hazard pointer protect

Finally, we come to the retire operation. In libcds, a retired pointer that is not protected anymore is not simply freed. Instead, when calling `retire`, the user specifies a `Disposer`. This disposer is then called instead of the standard library free, enabling the user to specify custom behaviour. Since GenMC looks back at the history of a memory location when it is accessed to determine whether it is a legal access, there never is an actual free that happens at some specific point in time, but rather the verifier checks whether there have been any free events before an access to a location and whether the location is protected at the time of access. If it is not protected, the model assumes that the garbage collector has done its job at some unspecified time and regards the memory location access as invalid. This saves a lot of verification resources, as the hazard pointer garbage collection does not have to be tracked explicitely and is reasonable, since the exact scheduling of the garbage collector is usually not transparent to the user anyway. Modeling this behaviour would require an extensive rebuild of the hazard pointer garbage collection model of GenMC. A simple example to show why is a disposer accessing a global variable. If this is the case, the verifier needs to assign a specific time to the access to see the value read or inform other events of the value written.

For our model, this means that using a custom disposer on a retired pointer is replaced by a simple free operation. Modeling the custom dispose functionality of the library is left as future work as the extensive rebuild of this aspect of GenMC is out of scope for this thesis.

4.5 <thread>

The C++ standard library header `<thread>` is used in two different ways in libcds, for executing backoff strategies and to explore the proccessor topology. Since currently, GenMC has no model for the C++ thread header, we have to modify the usage of it in the library or model it in GenMC. Since the usage in the library is not extensive and not relevant for the verification, we will go with the former option.

4.5.1 Yield and sleep

Firstly, it is used in the implementations of some backoff functionality. As a reminder, most data structures offer a backoff upon conflict functionality, which allows the threads executing the conflicting operations to pull back before retrying to increase the chances of success. The exact details depend on the chosen implementation. The library uses the `<thread>` header to implement some standard strategies. In those strategies, the `<thread>` functionality used is `std::this_thread::yield()` and `std::this_thread::sleep_for`. As the model checker both does not care about the time intervals between events, only their interleavings and also checks all interleavings anyway, these commands would not lead to different executions being explored and we can simply leave them out from the model.

4.5.2 Processor topology

The code that is executed during the initialization of libcds (see 2.4), when the library has recognized the operating system it is running on as Linux. It tries to find out the number of concurrent threads supported by the implementation by invoking the function `std::thread::hardware_concurrency` [3]. Since to verify the library functionality, we do not want to restrict ourselves to specific operating systems but rather want to verify the logic of the library abstracted away from these details, we do not want to model this anyway. Helpfully, the library itself provides a "fake topology" that stubs these functions to return dummy values or not execute at all, which is exactly what we would have done if we had to do it ourselves.

With this, we have provided stubs for these OS directives from the `<thread>` header that the library relies on, making it once again compatible with GenMC.

4.6 <mutex>

Since currently, there is no model for the C++ standard library <mutex>, we are once again faced with the choice of changing the library model to accomodate it, and changing GenMC to model it. While it would be nice to

have support for this standard library in GenMC, we did not have the time to add this feature, so we had to adjust the programs that we verify. Since there are some lock-based data structures in libcds that use <mutex>, we can not verify these. Additionally, some features like *elimination* (see 2.3.2) uses these locks as well. This means that we have to turn this feature off when we verify the data structures. However, this is not the main reason that keeps us from modeling elimination, so we do not actually lose too much there, as we will see next.

4.7 Randomness

Currently, GenMC is not equipped to deal with functions returning random values. Since at the time of writing this thesis, work has already begun on integrating this feature, we will not attempt to do so ourselves. For now, we will turn off the only feature that uses this, which is *elimination* (see 2.3.2).

4.8 Variable name collection

Since GenMC operates on the LLVM IR instead of the C or C++ source code and not all source language features correspond one to one with the LLVM IR, some difficulties can arise when dealing with the LLVM IR code. This is even more pronounced with C++, as GenMC has been largely untested on C++ code, which has generally more abstraction than plain C. In these cases, we might have to adjust the parts of GenMC dealing directly with the LLVM IR. Such features are `static const` or `static constexpr` class members. These members do not get compiled into the corresponding struct type in the LLVM IR, since it does not make sense to carry that information in every instance of the class. A simple example is the following program, where we create an instance of a `TestClass` using the default constructor. After this, a failing assertion is reached simply to make the verifier find a safety violation and print the error trace.

```
1  class TestClass {
2      public:
3          static constexpr int ctor_does_not_write_this = 0;
4          int ctor_writes_this = 1;
5  };
6
7  int main() {
8      TestClass testClass;
9      assert(0);
10     return 0;
11 }
```

Listing 4.14: A test case using `constexpr` from C++

When trying to verify this program, we come across a curious trace (see listing 4.15). We can see that the program was verified correctly, since the value 1 was written right after the allocation of the `testClass` object. Since the verification operates on memory locations and not variable names, this often does not lead to a correctness issue at all, only confusing the user a bit when trying to debug their program, as the wrong variable name `ctor_does_not_write_this` is printed in the trace. Of course we know this to be impossible, since static class variables are not initialized when an instance of the class is created, but in a more convoluted scenario, this may not immediately become obvious. In addition to this, entering wrong information into the GenMC data structures may violate invariants that are expected to hold. In our case, this can end up in an out of bounds access to a vector, leading to reading from and dereferencing unallocated or otherwise illegal to access memory, crashing the verification as a result.

```
1 Error: Safety violation!
2 Event (0, 2) in graph:
3 <-1, 0> main:
4         (0, 1): MALLOC testClass
5         (0, 2): Wna (t.ctor_does_not_write_this, 1) L.7
6
7 Assertion violation: 0
```

Listing 4.15: A faulty trace printout

To uncover the reason for the occurrence of this bug, we need to look at one of the passes that GenMC performs, namely `MDataCollectionPass`. This is where the class member is associated with the wrong name.

4.8.1 How it works

When GenMC executes its `MDataCollectionPass`, it goes through global and local variables and their types in the LLVM IR and tries to get their names through the LLVM IR debug data combined with the LLVM IR types. The part we will focus on is how the names of struct members are collected. For this, we will give a small example of a code snippet that works as intended:

```
1 class TestClass {
2     public:
3         int var1;
4         long var2;
5         char var3;
6 };
7
8
9 int main() {
10     TestClass testClass;
11     assert(0);
12     return 0;
```

```
13 }
```

Listing 4.16: Example program for correct variable name collection

Collecting the names of struct members consists of two parts. The first is the anonymous LLVM IR type that is associated with a variable. In our example in listing 4.16, this is simply the following:

```
type {
    i32,
    i64,
    i8
}
```

As we can see, the LLVM IR does not retain variable names, so we need the second part, which is the debug information that the LLVM IR retained during compilation. We retrieve the elements of the struct and, with some simplification for conciseness, arrive at the following:

```
TestClass {
!DIDerivedType(tag: DW_TAG_member, name: "var1"),
!DIDerivedType(tag: DW_TAG_member, name: "var2"),
!DIDerivedType(tag: DW_TAG_member, name: "var3")
}
```

This makes it easy to retrieve the names of the members. We simply iterate through the type information and the debug type information at the same time and associate each offset with the right name, which we save to be accessed later. However, this approach leads to problems when there is some mismatch between the type and debug type information, which is exactly the case with our example from listing 4.14. When we look at the normal LLVM IR type of our variable, we only see a singular member, even though the original code had two. This happened because the value of the `constexpr` has been evaluated at compile time and optimized out from the type, since it is a waste of memory to have it included in every instance of the class. Thus, the type we are left with is:

```
type {
    i32
}
```

However, when looking at the debug type information of the same class, we find something different, as both variables show up in the debug data:

```
TestClass{
    !DIDerivedType(tag: DW_TAG_variable, name: "ctor_does_not_
    write_this")
    !DIDerivedType(tag: DW_TAG_member, name: "ctor_writes_this")
}
```

Now, the root of the bug is obvious. When we try to assign a name to the member in our LLVM IR type, we iterate through both the type and debug type information, associating the first entrance in the type (corresponding to `ctor_writes_this`) with the name of the first entrance in the debug type, `"ctor_does_not_write_this"`.

4.8.2 The fix

Luckily for us, despite not having a direct correspondence in LLVM IR, the debug type of these special members still has some differences that we can look out for. With some testing, we can find out that if a class member has

- the tag `DW_TAG_variable`,
- a base type with tag `DW_TAG_const_type`, and
- the flag `llvm::DINode::FlagStaticMember` set,

it is likely a remnant of a class member that has been optimized out from the LLVM IR type and we can skip it during the iteration through the names of the variables. We need to note that these factors are only a heuristic as compilers are not required to translate these members in a predefined way.

4.9 annotationElimination

In 4.3.1, we have talked about GenMC's annotation system, specifically how it is used throughout GenMC. However, the mechanism of finding the right instructions that need annotations can be tricky as well, hiding a couple of bugs. In this section, we will go through the process and highlight the changes we made to it.

4.9.1 Finding instructions to annotate

GenMC's internal annotation functionality is implemented using two internal functions, `__VERIFIER_annotate_begin` and `__VERIFIER_annotate_end`. These functions need to be placed around any instructions that we want to annotate. During the compilation of the source code to LLVM IR, these get translated as simple function calls. Next, the `annotationEliminationPass` pass is carried out. It simply searches through the entire function that is being looked at currently and saves references to all the *begin* and *end* instructions separately. Then, it tries to match them up with each other. This is done with the help of some LLVM analyses, namely dominator and post-dominator trees, as well as the annotation values themselves.

Our strategy is as follows. We will go through each *begin*, and try to find its pair. To do that, we can define some properties for an *end* that matches. Namely, such a candidate needs to:

1. have the same annotation value,
2. be dominated by the begin,
3. post-dominate the begin, and
4. have no interfering *end* instruction.

An interfering *end* is one that is distinct from the candidate and has properties 2. and 3., with the addition of also dominating the candidate. In basic terms, it is an *end* that sits between the candidate and the *begin* that we are trying to match up. This is actually where the first bug was situated. We actually need to extend this definition of an interfering instruction to also encompass property 1., otherwise a completely unrelated *end* instruction with a different annotation value could keep us from finding the true match of the *begin* that we are looking for.

The second bug was also hiding here. The LLVM (post)dominator tree analysis is only valid for reachable instructions. This means that it is possible that some *ends* interfere with each other even though they should not, simply because the information that we are provided with is invalid. We can avoid this by filtering out the instructions that are not reachable from the entry point of the function before trying to match them up.

After we have found a pair, we can go through each instruction on each path between them and apply our annotations as described in section 4.3.1.

Chapter 5

Results

We will first provide a summary of the changes in the library and GenMC that we implemented to enable the verification of the data structures. Next, we will give an overview of the data structures that are compatible with the current model and constraints. Finally, we will go through the verification results of the data structures, discussing their correctness, the scope of our guarantees, performance evaluations, as well as fixes for the bugs that we find.

5.1 Summary of changes

5.1.1 Libcds

We have replaced the libcds hazard pointer implementation with our own model, rerouting calls to allocating, freeing, and clearing hazard pointers, as well as protecting, retiring, and retrieving pointers, to the GenMC hazard pointer API. This includes removing the thread-local data manager infrastructure, since we are keeping track of any hazard pointer related information in the execution graph. This also made the initialization and termination of the library redundant. As for the data structures, we were mostly able to keep them as they were. Data structure features such as elimination that use `<mutex>` need to be turned off, since we currently are not able to model-check them.

5.1.2 GenMC

The main work here was changing the model of hazard pointers to store the protected pointer in a real data structure, so that we can easily access it later. We took care to emulate the behaviour of the old model exactly, in regard to executions explored and blocked, while adding new features on top. This new model required changing the hazard pointer protect operation

from an internal function call to a store to a data structure. Reintegrating this with the old verification process called for notifying the interpreter that this is a store with special semantics, which was achieved by adding a new annotation and adapting the interpreter to check for it.

Apart from the hazard pointer model, some bugs occurring around the annotation and class member name collection system have been ironed out. In addition, some features supporting user-friendliness of the tool have been added, such as the ability to insert custom events into the execution graph to aid establishing the connection from the trace back to the source code.

5.2 Overview of data structures

With these changes, we have the ability to check programs that use the libcds concurrent data structures library. We will stick to verifying intrusive data structures (see 2.3.1), since in general, the non-intrusive ones are built on top of the intrusive ones, with some extra functionality to allocate the nodes so that the user does not have to. The actual operational logic stays in the intrusive implementation, so that is a good place to start the model checking.

For the different data structures, we will have different methods for verification, depending on what is available in GenMC, what we want to achieve, what our model supports, and what our constraints are. This means that we will only be able to verify a subset of all data structures. The ones that are compatible with our constraints are:

1. Stacks
    - Treiber stack [16]
2. Queues
    - Michael & Scott's queue [15]
    - Moir queue [7]
    - Optimistic queue [10]
    - Vyukov's MPMC bounded queue [17]
3. Lists
    - Iterable list[1]

5.2.1 Checking linearizibility

GenMC has a component called RELINCHE [8], which can check the implementation of a data structure in general, instead of only specific clients. It

[1]This data structure does not have a specific paper it is based on.

achieves this by constructing a ”Most Parallel Client”, and checking it for a (for the weak memory consistency model relaxed) linearizibility property up to some bounded invocations of its operations. This check includes the usual freedom from safety violations and memory safety checks, but also checks the returned values against a specification, ensuring functional correctness as well. Then, the correctness of this special client is generalized to all possible clients, effectively covering any possible application using the data structure up to the checked bound. Currently, this tool is applicable to stacks and queues.

5.2.2 An obstacle

During the verification of some data structures, we came across a curious trace when investigating an error, supposedly a use-after-free. The trace, in its essence, boiled down to the following events:

```
1 (0, 26): MALLOC  [0x2000000000000120]
2 (0, 45): FREE    [0x2000000000000120]
3 (0, 51): MALLOC  [0x2000000000000120]
4 (0, 52): Rrlx    [0x2000000000000120]
```

This means first allocating a specific memory region, then freeing it and then reallocating the same address and finally accessing it. While this is perfectly fine in a normal program, GenMC models it a bit differently. Namely, every new memory allocation is supposed to return a fresh address, since the model checker does not have to worry about the memory efficiency of the program, only its correctness. For this reason, it only checks whether an address is in a freed memory region when it is accessed, and not if it has been re-allocated again, since that is never supposed to occur. This only occurs with hazard pointer allocations, so to make sure it is not a problem, we needed to revert to the old model and try to check the same programs. Unfortunately, the problem persisted, indicating that this was not a consequence of using the new model. We can circumvent this by never freeing the hazard pointer data structures at all. While this obviously leads to memory leaks (that GenMC will detect), we can still use the new model to verify data structures that use the hazard pointer get operation. This will give us weaker guarantees, since we stop modeling part of the program, but the deallocation of garbage collector objects should not influence the logic of the data structure implementations. This issue affects every data structure that uses multiple hazard pointers at the same time, so all except the Treiber stack.

5.3 Let's verify!

Finally, we have done all the work recquired to start using the tool. We will go through the data structures in order and discuss the results yielded.

5.3.1 Treiber stack

We will use this data structure to demonstrate the process. Since this data structure only uses a singe hazard pointer per thread, we can use the full hazard pointer functionality here, as well as the RELINCHE linearizibility checker. The only caviat is that elimination (see 2.3.2) has to be turned off, since it uses both locks (see 4.6) and randomness (see 4.7). To use RELINCHE, we need to satisfy the interface it requires, which looks like the following:

```
1 void init_stack(mystack_t *s, int num_threads){};
2 void clear_stack(mystack_t *s, int num_threads){};
3 void push(mystack_t *s, unsigned int val){};
4 unsigned int pop(mystack_t *s){};
```

Listing 5.1: The RELINCHE stack interface

We can simply define our stack type as seen in listing 2.10, then just call the stack methods directly. Since the interface works with the `int` type, we need to wrap it into the type that we want to put into the stack and unwrap it to check the return values. The final change we have to make is remove the destructor of the stack from the library code, since RELINCHE constructs a global instance of our stack, for which the compiler handles the destructor in a way that is currently not compatible with GenMC. This issue arises because the destructor is dynamically assigned to the object and GenMC can not link these objects based on only the LLVM IR. Luckily, the destructor would only call `clear`, which is something that will be done in the client anyway. With that done, this will be our final code:

```
1  void init_stack(mystack_t *s, int num_threads){}
2
3  void clear_stack(mystack_t *s, int num_threads)
4  {
5      s->clear();
6  }
7
8  void push(mystack_t *s, unsigned int val)
9  {
10     myData *newData = new myData(val);
11     s->push(*newData);
12 }
13
14 unsigned int pop(mystack_t *s)
15 {
16     myData *poppedData = s->pop();
17
```

```
18      unsigned int retval = 0;
19      if (poppedData)
20          retval = poppedData->value;
21
22      return retval;
23  }
```

Listing 5.2: Treiber stack RELINCHE client

Result Invoking RELINCHE with the provided "Most Parallel Client" with the code above and the type definition included confirms that this data structure is linearizable, meaning that every possible client using the data structure implementation is guaranteed to be correct up to the checked number of method invocations. In figure 5.1, we can see the time taken for verification based on the number of reader (R) and writer (W) threads. The figures are based on the average duration of five runs (except for 3R - 3W, which was a single measurement). Note the logarithmic scale of the time axis.

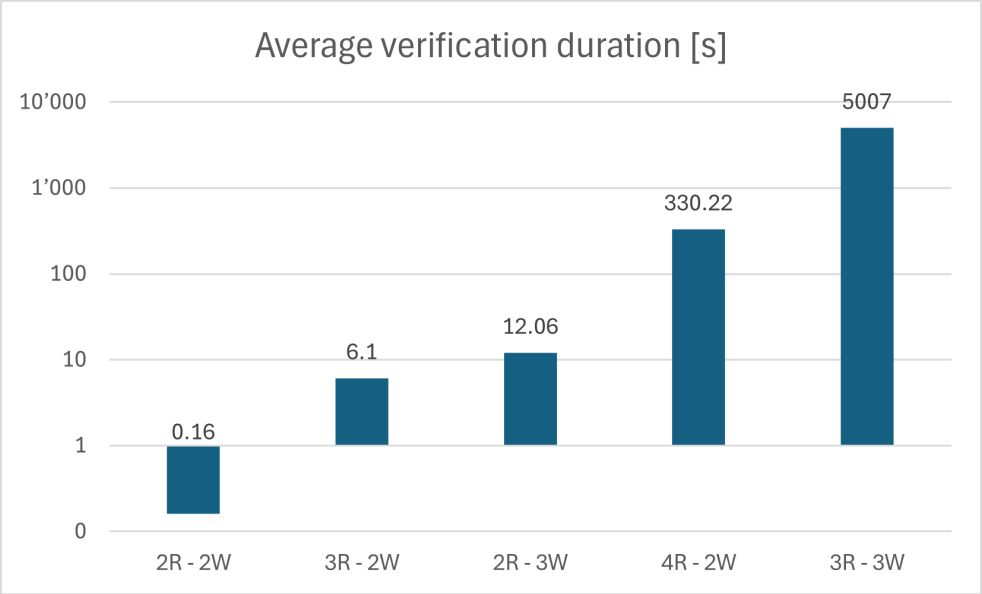

Figure 5.1: Duration of verifying linearizibility of the Treiber stack implementation

On its own, we do not learn a lot from this graph. We can see that more threads scale the time needed for verification exponentially, which is to be expected. We can see that having more reader threads is faster to verify than having more writer threads. This can be explained by the fact that reader threads can fail to pop if the stack is empty and thus exit early and not

change the data structure, resulting in the verifier having to examine fewer interleavings of dependent instructions. We refer back to this graph in a later section (5.3.3).

5.3.2 Michael & Scott's queue

From this point on, we will need to use the reduced hazard pointer model where the hazard pointer data structure is never freed, in order to work around the bug described in 5.2.2. Verifying a queue is very similar to a stack, the interface that we have to satisfy is slightly different, as `push` is replaced by `enqueue` and `pop` is replaced by `dequeue`. Additionally, the value dequeued is written to a location given as a pointer parameter to the function and the return value is a simple `bool` indicating success.

Result

As it turns out, the linearization of the Michael & Scott's queue fails. This can be traced back to some missing synchronization and can be recreated with a single enqueue thread and two threads that dequeue. Since the full execution graph is a bit unwieldy, we will only include it in the Appendix (A.1). However, most of these events are irrelevant, so I will show a concise version of the parts relevant to the unwanted behaviour. Of course, there are some more operations that we need, like advancing the tail, but we only kept the parts that are explicitely relevant to the missing synchronization. Note that CW refers to the store part of a compare-and-swap instruction. Additionally, we have indicated the reads-from relation with arrows.

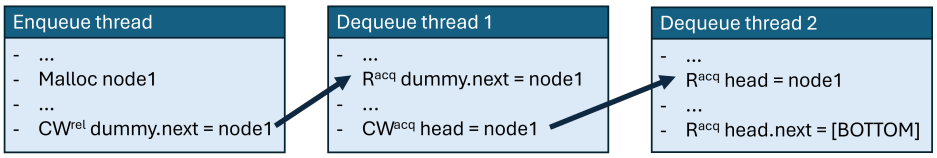

Figure 5.2: The events leading to the bug

We start the trace with the queue initialized and holding a single dummy node. As we can see, the enqueue thread allocates a node and successfully inserts it at the tail of the queue, setting the next pointer of the dummy node to the new node with a CAS. This happens with `release` semantics, so the following read in the dequeue thread, which has `acquire` ordering and reads from that write is able to synchronize these threads around this pair. However, when the second dequeue thread uses a CAS to advance the head to dequeue, this only happens with `acquire` semantics. This means that the third thread, also reading the head with `acquire` ordering, can not synchronize. Since there has been no synchronization, the memory

accesses, particularly to `head.next` can be reordered around this read. As is visible in the larger trace (A.1), the third thread has no other synchronizing reads or writes. This means that the read can be reordered to all the way before the node was allocated in the first thread and thus access unallocated memory. While this is not a problem by itself, the next instruction would have dereferenced this access, which is of course undefined behaviour and a memory safety violation.

The fix

Looking at this sequence of events, it becomes clear that we can fix this bug by changing the memory ordering of advancing the head with a CAS operation during dequeuing to `release` or stronger. Since we want to remain as efficient as possible, we will choose the least restricting option of `release`. With this change, we can now run RELINCHE again and confirm that it worked, as the verification succeeds. We will look at the performance together with the next data structure.

5.3.3 Moir queue

This data structure is based on the Michael & Scott's queue from the previous section. Internally, it even uses the same file to reduce code duplication. The crucial difference is that the dequeue operation that contained the bug in the Michael & Scott's queue is different, since the authors saw a way to optimize the number of accesses to global variables leading to better performance during high contention periods. Additionally, they formally verified the correctness of their queue [7].

Result

With this knowledge, it does not surprise us that the design is correct and in addition, the libcds implementation is as well. The Moir queue is linearizable without any changes required. A bit more interesting is the performance of the verification itself. In figure 5.3, we can see the average duration of the verification using RELINCHE across five runs. Included are the numbers for the Michael & Scott's (MS) queue and the Moir queue. Again, the number of reader and writer threads is indicated by R and W respectively.

In this figure, we can see two things. First, even though the verification process does not care about the efficiency of the checked program at all, the verification data structure that deals better with high load completed much faster when there were more threads involved. This may be attributable to the effiency improvement that the Moir queue has introduced. When there are fewer accesses to global variables, there are fewer dependent instructions whose different interleavings GenMC has to explore, leading to a faster

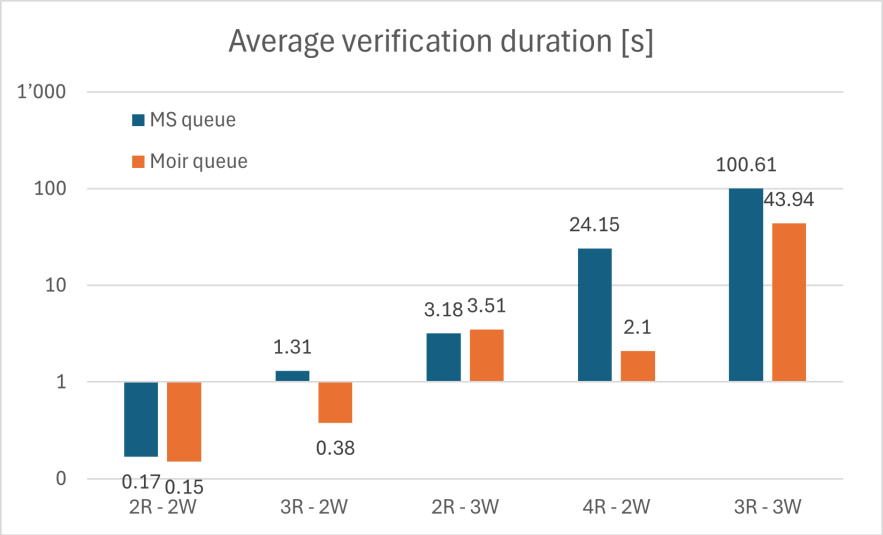

Figure 5.3: Verification times of MS and Moir queue

verification. While this does not have any implications beyond this example, it is a neat fact to think about.

The second thing we can read off of this graph concerns the verification process a bit more. To see it, we need to compare this with the results of the Treiber stack verification (see figure 5.1). We can notice that the verification of the stack took orders of magnitude longer than of the stacks. We can easily find out how this happened, namely the stack verification could not find any executions that it could consider blocked. Usually, this happens due to the verifier recognizing a situation where it can abort the exploration of a specific execution in a sound way. Finding out why it could not find any such opportunities for optimization is more difficult and beyond the scope of this thesis.

5.3.4 Optimistic queue

This queue implementation tries to apply an optimistic approach to concurrent queues. It replaces the singly-linked list structure of the other queue implementations with a doubly-linked list, and replaces the relatively costly compare-and-swap pointer updates with simple stores. Whenever a bad ordering of events causes the list to become inconsistent, it can be fixed again with the help of the double links. This means that under lower contention when the optimistic low-cost stores do not become inconsistent too often, the queue performs better than a Michael & Scott's queue. Since it is a normal queue implementation, we can once again use RELINCHE to check linearizibility.

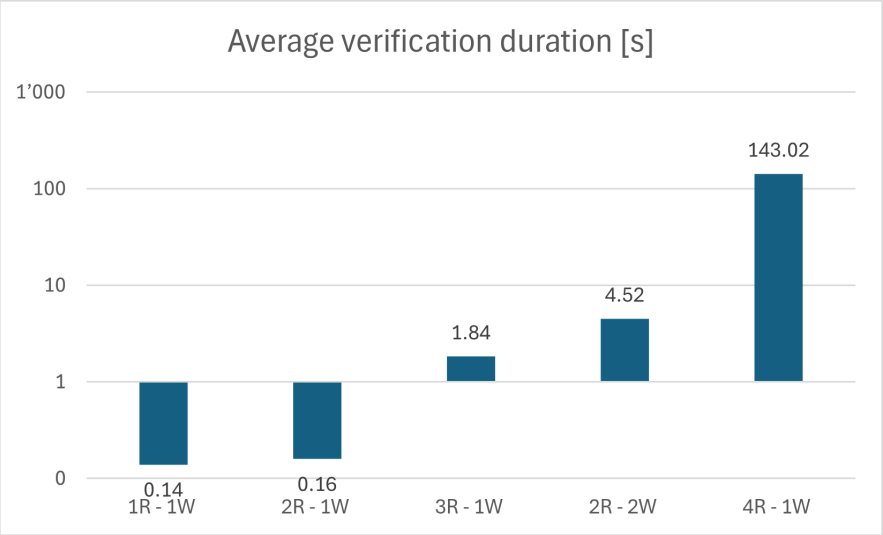

Figure 5.4: Verification times of Optimistic queue

Result

With the appropriate client satisfying the "Most Parallel Client" interface, we can run RELINCHE. It tells us two things. First, there is an execution where non-allocated memory is accessed, this is something we can fix. Second, it tells us that there are unordered writes. This is only a warning, since it does not directly lead to errors but is often a symptom of a bad design. In our specific case, it seems to be the result of the optimistic approach. When we use the relatively low-cost stores, the read that synchronizes the operations does not happen atomically with the store, as would be the case with a RMW operation. Since the queue has another mechanism, the consistency check, that can correct the pointers if something goes wrong, this is not an issue for this implementation.

The fix

This case is very similar to the Michael & Scott's queue issue that we looked at before. We can trace back the synchronization issue to a compare-and-swap instruction that has acquire ordering. Once again, we can change it to release, which allows us to rerun RELINCHE and prove that the data structure is linearizable. We should note that for this data structure, we can only verify up to two reader and writer threads, as more would take an unfeasible amount of time, so the guarantees we get for this data structure are weaker than those for the Michael & Scott's and Moir queues and the Treiber stack.

Figure 5.4 shows us the time, averaged over five runs, that it takes to verify

the linearizibility of the optimistic queue with the noted number of reader (R) and writer (W) threads, plotted with a logarithmic time scale. We can clearly see that more threads very rapidly increase the time taken, which is to be expected. We can even see that the increase seems to be more than linear on the logarithmic scale. With this data, it is not surprising that the tool would take a long time to verify larger scenarios. However, what is a bit unexpected is that GenMC estimates the executions it needs to explore as about 800 billion, taking almost 18 years to finish. While these numbers are extremely rough estimates and should not be taken too seriously, it does tell us that the verification is not feasible. Indeed, the process takes longer then 15 minutes, after which it was aborted.

5.3.5 Vyukov's MPMC bounded queue

Once again, we are verifying a concurrent implementation of a queue. This queue is array-based, fails on overflow and actually does not require garbage collection. This means that we the results we get for it are valid for the whole implementation and do not have to ignore any hazard pointer memory leaks. While not technically lock-free as progress can be blocked, the locks are implemented by atomic RMW operations instead of mutexes, so we can still verify the data structure. We will again create a client for RELINCHE so that we can check the linearizibility.

Result

This data structure turns out to not be linearizable out of the box, as it is possible that threads return unexpected values because we can not establish a total order on the methods that are invoked on the queue. While the operations are synchronized enough that there are no accesses to freed or uninitialized memory, we can not linearize the execution.

The fix

We need to introduce additional snychronization to be able to linearize the execution. To achieve this, we first try to only strengthen the memory orders of the different operations as little as possible, usually this means stores to release and loads to acquire semantics. However, in this case this results in too many executions that need to be checked. However, when we strengthen the orderings to be sequentially consistent, the verification time goes down to manageble levels again. This does not mean that the weaker orderings are not enough for linearizibility, just that we can not check that option.

5.3.6 Iterable list

This list implementation verification has to be a bit different from the rest. Namely, since there is currently no specification and no "Most Parallel Client" for list data structures in RELINCHE, we are not able to determine whether this implemention is linearizable. Note that this does not stem from a fundamental incompatibility with the tool, just that the relevant parts have not been implemented yet. It also does not mean that the data structure is not linearizable, only that we can not examine that property. To still get some results, we implement our own client that concurrently inserts and erases nodes from the data structure. Additionally, the execution graphs that are built during verification get very large, with millions of events. Since that would take an extremely long time to verify, we need to make use of another feature of GenMC, namely loop unrolling. With this, we can get down the number of executions we need to explore to a managable level.

Result

When trying to verify our custom client, we quickly find another bug. Similarly to the Michael & Scott's queue implementation, we see that there are executions where uninitialized memory can be accessed. This, we can once again trace back to insufficient synchronization between the threads due to memory accesses with orderings that are not strong enough.

The fix

The fix in this case is not that simple. Changing the orderings of some of the memory accesses temporarily allows the verifications to succeed, however, introducing more threads uncovers even more bugs. This goes on to the point where the verification will take an infeasibly long time. Since we can not make sure, that introducing one more thread will not uncover another bug during verification, to remain on the safe side, we need to change all the accesses to have a sequentially consistent memory ordering. While this may cost us performance when using the library, at least we know that the operations will be synchronized properly. This issue of the verification time increasing exponantially with the size of the program that we are trying to examine is well-known and even the heavy optimizations that GenMC employs can not keep up with it indefinitely.

Chapter 6

Conclusion

We have shown the full process of verifying data structures in the libcds concurrent data structures library for C++. This included explaining how libcds works, some theoretical background for understanding the results, as well as the changes necessary both in the library and in GenMC to allow the verification of some client programs using the library. We implemented a new hazard pointer model in GenMC, keeping the explored executions for programs using it the same as before the changes with the help of GenMC's annotation system. For most data structures, we were able to apply RELINCHE, GenMC's relaxed linearizibility checker, to get even stronger guarantees about the data structures. During this verification process, we found multiple bugs in the implementations and were able to fix them. The maintainers of the library have been notified of the issues found and the proposed changes. Additionally, we were able to find some errors in GenMC itself and fix most of these as well.

6.1 Improving scalability

While we have not been able to include a complete analysis in the thesis, we have found some opportunities to enhance the scalability of the tool. Specifically, we saw that the linearisation of the Treiber stack implementation failed to recognize any opportunities to block executions during exploration. Contrasting this with GenMC's own Treiber stack implementation, which does find opportunities to optimize in this way, we can see that the problem is not inherent to the data structure. Thus, we have a program that can be used as a case study to find opportunities in the verifier, perhaps as annotations added to the program, by enhancing some pre-exploration transforms or passes, or even by changing the exploration driver.

6.2 Future work

The results from this thesis can be used for further verification work. To achieve this, more work is required in some areas. Specifically, GenMC would need to be extended to support more C++ header libraries.

Support for <mutex> Adding support for this new C++ library would directly allow the verification of more libcds data structures. Currently, we can only verify those that in their core are entirely lock-free. This would also be a good starting place for someone trying to familiarize themselves with the GenMC workflow and inner workings, as well as provide some insight into the libcds library.

Support for randomness Adding the ability to deal with randomness requires more familiarity with GenMC. Adding this capability would result in the verification of some randomness-based data structures as well as of some features that require randomness, like the backoff elimination in the Treiber stack (together with <mutex>).

GenMC Additionally, there still remains the bug in GenMC that prevents us from freeing hazard pointers when verifying a library. This means that any verification results can not account for memory leaks, as there will be some anyway as soon as hazard pointers are used. Luckily, we have a few different programs that exhibit this behaviour, so there is a starting point for working on this issue.

Finally, as is the case with many automatic verification tools, working on scalability issues will always open the possibility of finding contrived bugs that require a larger number of executions explored.

Appendix A

Appendix

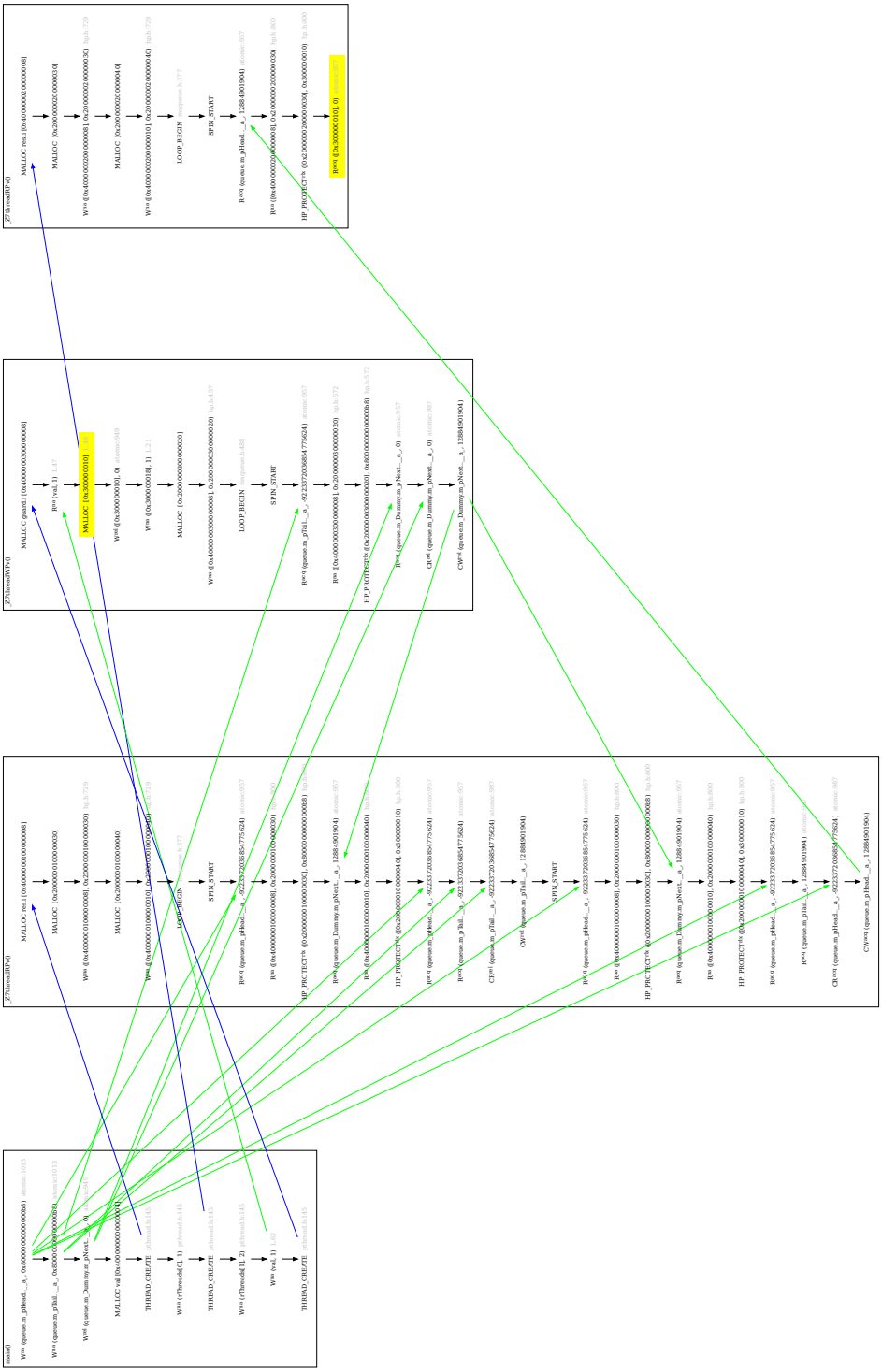

Figure A.1: The full trace of the Michael & Scott's queue access to uninitialized memory

Bibliography

[1] Andrei Alexandrescu and Maged M. Michael. Lock-free data structures with hazard pointers. *C++ User Journal*, pages 17–20, 2004.

[2] Hans Boehm. Defang and deprecate memory_order::consume. Paper P3475R1, JTC1/SC22/WG21, 2025.

[3] Online C++ reference page. https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency.html. Accessed: 14.07.2025.

[4] Mathieu Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009. Chapter 6.

[5] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. Supplementary material for user-level implementations of read-copy update. *sleep*, 5(40):41, 2010.

[6] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2011.

[7] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 97–114. Springer, 2004.

[8] Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. Relinche: Automatically checking linearizability under relaxed memory consistency. *Proceedings of the ACM on Programming Languages*, 9(POPL):2090–2117, 2025.

[9] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.

[10] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20(5):323–341, 2008.

[11] libcds source code. https://github.com/khizmax/libcds. Accessed: 12.08.2025.

[12] libcds documentation. https://libcds.sourceforge.net/doc/cds-api/index.html. Accessed: 03.08.2025.

[13] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC '02, page 21–30, New York, NY, USA, 2002. Association for Computing Machinery.

[14] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[15] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[16] R. Kent Treiber. Systems programming: Coping with parallelism. *International Business Machines Incorporated*, April 1986.

[17] Bounded MPMC queue. https://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue. Accessed: 22.07.2025.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☒ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

Model-checking the libcds library

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Mezei	Balázs Benedek

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Baden, 12.08.2025	Benedek Mezei

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).