TDT4900: Masters Thesis

CMR: A concurrent memory management system for Rust

Martin Hafskjold Thoresen

supervised by
Magnus Lie Hetland

June 4, 2018

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Sammendrag

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Contents

List of Figures

Resolve These Things

Chapter 1

Introduction

This is a thesis blabla
Something about memory management and parallel systems
bing bong about CMR
Motivation
RQ ?

Chapter 2

Background

2.1 Garbage Collectors

A *Garbage Collector* usually refers to an automatic subsystem that handles memory management without requiring programmer assistence. Many widespread language implementations, including Java, Python, and Go, use a garbage collector, although the internal details of each system varies greatly.

31/05 10:58
What types of GC are there?

The job of the garbage collector is to identify memory segments that are no longer used by the program. One way of doing this is to represent the program memory as a graph $G = (V, E)$ where V is all allocated memory segments and $(u, v) \in E$ if the region u contains an address that is inside the segment v. One consequence of this model is that memory addresses cannot be computed from other values.

31/05 10:58
More here

31/05 10:58
Discussion about GC in general?

2.2 Operating Systems

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.2.1 Virtual Memory

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Memory Maps

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.2.2 Threads and Processes

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.2.3 Signals

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.3 Programming Languages

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.4 Concurrency

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.4.1 Common Patterns in Concurrent Programming

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.4.2 The ABA-Problem

2.5 Memory Reclamation

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.5.1 Hazard Pointers

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.5.2 Forkscan

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.6 Related Works

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.6.1 Crossbeam

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit

blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 3
Rust

Rust is a new programming language focusing on safety, performance, and concurrency[14]. The official first stable release, Rust 1.0, was released in May 2015, and a new version of the language as well as the official compiler, `rustc`, is released every 6th week. The language is developed as an open source project on the version control platform GitHub[8] by over 2000 contributors as of May 2018[15]. The Rust project is organized into *teams*, such as the Core Team, the Compiler Team, and the Documentation Team. Many of the members of the Rust teams are Mozilla employees, and Mozilla officially sponsors the Rust project. The language has no formal specification, although all language changes are developed and documented through an Request For Comments (RFC) process. For a thorough introduction to Rust, see [17].

3.1 Introduction

Rust is a compiled language with a minimal runtime, similar to C and C++. `rustc` uses LLVM[2] as a compiler backend for code optimization and code generation. The performance of Rust code is very similar to that of C and C++[1]; variations are often due to the lack of stable features like SIMD support, or from different compile time information given to LLVM by either language.

Rust has many features from the ML family of programming languages, such as pattern matching and tagged enums, and a rich type system with type inference. Most notably, and unlike most other modern programming languages, Rust does not have a garbage collector. Despite this, Rust programs does not handle memory management manually; memory management is typically done statically at compile time by utilizing language features covered in the upcomming sections.

Rust uses `struct`s similar to C and C++ which can have *methods*, but it does not have inheritance. `Trait`s are similar to interfaces: they define methods and optionally an implementation, and `struct`s *implement* the `Trait`. Traits can even be implemented for types that we have not defined ourselves, as long as we have defined the `Trait`. This is useful, since it means we can extend types from the standard library, or from other third party crates[1]. Important `Trait`s include `Deref` (the `*` operator), `Clone` (values that are clonable), and `Drop` (ran when a value is destroyed).

[1] A *crate* is a project unit, similar to a library

When an owned value leaves its scope, it is destroyed and its `Drop` method is ran. Primitive types, such as `char` or `u32` does not have a `Drop` implementation, but types which holds a resource, like allocated memory, often has. `String` and `Vec<T>` are common examples. `String` has a pointer to an internal buffer, which needs to be freed upon destruction in order not to leak memory. This `free` call is done inside `String::Drop`.

3.2 The Borrow Checker

A central concept in Rust is that of ownership. At any moment, an object has exactly one binding which *owns* the object. Ownership may be transfered ("*moved*", which is the default behaviour), or it may be *lended out*. Then the receiver is *borrowing* the binding. There are two types or borrows: immutable and mutable borrows. One of the reasons to differentiate between mutable and immutable borrows, is references in Rust can be either aliased, or mutable, but never both. That is, if there is a mutable reference to some object, then that reference has to be the *only* reference. This ensures that immutable references are never changed, which makes it simpler for the programmer to reason about the code since we get referential transparency, in addition to that it enables more compiler optimizations.

Borrowed objects are in effect *references* to some data, similiar to pointers or references in other programming languages. While Rust does have raw pointers (see Section 3.4), it is rarely used, and passing values by reference is prefered. The three types of ownership handling is shown in Fig. 3.1. In Fig. 3.1a we move `x`, so `x` is no longer usable after the last line, and an attempt to use it is caught as a compile time error: `error[E0382]: use of moved value: ‘x‘`. Since the caller of `foo` has "sent" the `Foo` to the function, it does no longer have to do any cleanup: this is now `foo`s responsibility.

Fig. 3.1b shows immutable borrow of `x`; the function `foo` may use the `Foo`, but it cannot mutate it. Fig. 3.1c shows a mutable borrow; now `foo` may mutate the `Foo`. Note that the binding `x` also needs to be mutable in order to borrow mutably.

(a) Ownership transfer

```
fn foo(f: Foo);
let x = ...
foo(x);
```

(b) Immutable Borrow

```
fn foo(f: &Foo);
let x = ...
foo(&x);
```

(c) Mutable Borrow

```
fn foo(f: &mut Foo);
let mut x = ...
foo(&mut x);
```

Figure 3.1: The three types of ownership handling.

Understanding the borrow checker is often a pain point for new programmers, and the period in which new Rust programmers learns an intuition about how to structure programs within these rules is often refered to as "fighting with the borrow checker".

3.3 Lifetimes

Lifetimes is the second important concept in Rust. The idea of lifetimes is to reason about the duration of the program execution in which some object is valid — its lifetime. By tracking the lifetime of all variables at compile time the Rust compiler is able to catch errors such as returning function local variable addresses. Section 3.3 shows an example function attempting to do this.

```
fn foo(_a: &i32) -> &i32 {
  let num: i32 = 420;
  let r: &i32 = &num;
  r }
```

Since Rust tracks the lifetime of all variables, it knows that the lifetime of `num` is the same as that of the function body. The lifetime of `r` is the same, as it is a reference to `num`. So when we try to return `r` in the last line of the function, Rust realizes that the lifetime of the reference we return ends its life at the end of the function; this is clearly not what we wanted, since it would make the returned reference dead on arrival. Compilation fails with the following error: `error[E0597]: ‘num‘ does not live long enough`.

Although Rust programmers may have to think about the lifetime of the variables, they seldom have to write lifetime annotated functions, due to *lifetime elison* — the compiler can ususally figure out the most general lifetime that fits the function. Functions may be annotated with explicit lifetimes, for instance if it takes multiple references in which the relative difference of the lifetimes of the references is important. `struct`s can also be annotated with lifetimes, and in fact is required to be so if any of its members are references. This is because the lifetime of the struct is bounded by the lifetime of its member variables.

```
struct Person<'a> {
  age: i32,
  name: &'a str }
```

Should we have a function that crates a new `Person` we might want to annotate it explicitly, if the function takes multiple references, but only one of these references is the `name` field:

```
fn make<'x, 'y>(f: &'y File, n: &'x str) -> Person<'x> { ... }
```

This way we can convey the information that the resulting `Peron` should live as long as `n`, but may outlive the file `f`.

3.4 Unsafe Rust

When talking about the Rust progrmaming language, one usually talks about a subset of Rust, called *Safe Rust*. In Safe Rust, there are no race conditions, mutable memory locations are never aliased, and all pointer accesses are valid. The real world, on the other hand, rarely offers these guarantees, and the unfortunate truth which Rust programmers must deal with is that in order to implement some of these safe abstractions we want (like `Vec`, `Mutex`, and `Box`), some unsafety is required. For this reason, Rust offers an escape hatch for some of its rules: *Unsafe Rust*.

The difference between Safe and Unsafe Rust is only four things. In Unsafe Rust one may: 1) dereference raw pointers 2) mutate statics 3) call `unsafe` functions 4) implement `unsafe` traits. One way of thinking about the unsafety of ones codebase is that there should be no undefined behaviour in safe code, no matter how the code looks like. In other words, it should be impossible to mess up so badly as to invoke undefiend behaviour without typing `unsafe`.

Dereferencing raw pointers is naturally `unsafe`, as it is not possible to statically guarantee that the address of the pointer is valid memory, or that the objects it points to is still alive, nor that mutation of that memory does not change an immutable reference some other place in the program. Mutation of `static` variables is also unsafe due to mutability of aliased references, and due to the lack of thread synchronization.

`unsafe` functions and traits are just a marker added to the function or trait, signaling that not all uses of this is guaranteed to be safe. As an example, the trait `Send` is a marker trait and types implementing `Send` may be sent across thread boundaries. While this is fine for most types, there are types which does not allow this. The reference counted pointer `Rc<T>` is an example, which is a pointer to a tuple[2] `(count, data)`. The `count` is incremented each time `.clone()` is called, and decremented when a variable is `Drop`ped. To understand why this cannot be send across thread boundaries safely, consider what happens if T_1 `.clone()` at the same time as T_2 `Drop`s it: the `count` field is written to twice without any synchronizationor atomic operations[3] — a race condition!

3.5 Concurrency

bing bong

3.5.1 Concurrency and Aliasing

One observation to make from the reference rules as presented in Section 3.2 is that since references are either aliased or mutable, then there can be no writes shared data between threads, in Safe Rust, even using atomics. While this is *technically* true, the Rust standard library uses `&T` and `&mut T` slightly different than "immutable" vs "mutable" in this context: `&T` means that the type may be shared between threads.

Take `AtomicUsize` as an example, a `usize` exposing atomic operations like `store`, `load`, and `compare_and_swap`, which signatures are shown in Listing 3.1.

Listing 3.1: Signatures for selected operations on `AtomicUsize`

```
fn load(&self, order: Ordering) -> usize;
fn store(&self, val: usize, order: Ordering);
fn swap(&self, val: usize, order: Ordering) -> usize;
fn compare_and_swap(&self, current: usize, new: usize, order: Ordering) -> usize;
```

Clearly, `AtomicUsize::store` modifies memory of the `usize`; despite this the function is `&self` and not `&mut self`, since the operation is allowed on variables which are shared between threads. This is a useful distinction, since we can have methods on `AtomicUsize`

[2] Not really, but for our purposes here we can pretend that it is.
[3] `Rc` does not use atomics for performance reasons, but `Arc` does, and it does implement `Send`.

that *is* `&mut self`, which then is only possible to invoke should the variable not have been shared between threads yet; we know this since this means that we have aliased mutable references, which is not allowed. For instance, `AtomicUsize::get_mut(&mut self) -> &mut usize` allows the underlying `usize` to be changed without any synchronization overhead.

3.5.2 Common Patterns

The standard librarys synchronization module `std::sync` contains primitives that most concurrent programs require, such as `Mutex`, `Channels`, `Condvar`, and `Atomic`s. A common pattern in Rust is the Resource Allocation Is Initialization (RAII) pattern. The idea is that resources should be managed automatically when constructing and destructing an object. `Mutex` uses these ideas: `Mutex::lock` returns an `Result<MutexGuard>`, where the `MutexGuard` wrapps a mutable reference to the data that is protected by the `Mutex`. When the `MutexGuard` goes out of scope, its `Drop` implementation is ran, and the `Mutex` is unlocked.

It is common among Rust programmers to build abstractions over lower level primitives. For instance, a common pattern in parallel and concurrent programming is to have a *thread pool*, which is given work, and internally handles the thread synchronization and work division. Example usage of such an abstraction could be `let tp = ThreadPool::new(); tp.execute(|| ... );`. Since this can be implemented without any special compiler support, such crates are usually made as third party libraries.

Another example is data parallelism: given some collection of data we want to iterate over the elements and perform some operation on each element. The Rust library `rayon` offers exactly this: parallel iterators. Instead of writing `vec.iter()` to iterate over a `Vec` and then performing some operation on each element sequentially, with `rayon` we can write `vec.par_iter()`, and get data parallelism for free. The operation is then ran in parallel with any number of threads. Internally `rayon` uses a thread pool and work stealing to handle the division of labour among the threads.

09/05 12:32 Look at stuff from last semester. Move into std sec?

3.5.3 Memory Orderings

3.6 Nightly Rust

The Rust language and compiler follows a fixed release schedule, where a new stable version is released every six weeks. In addition to this there is the beta branch, which is the upcomming version, and the nightly version which is the most recent version, build daily from the `master` branch of the source tree.

The nightly version of the compiler allows users to opt in on *untsable* features: features that are partially or fully implemented, but which details are not yet committed to. These featuers includes new APIs in the standard library, new syntax, and new language features all together. As we have used multiple unstable features in CMR, we look at some of them in deatil.

3.6.1 Non-Lexical Lifetimes

The current implementation of lifetime checking in the compiler is *lexical*, meaning variables are live until they go out of scope, despite not being used. This is a limitation that one may want to get rid off. The feature Non-Lexical Lifetimes (NLL) lifts this requirement, and lets the lifetime of a variable last only until its last usage. Having this it is possible to seemingly break some of Rust rules, like aliased mutable references:

```
let mut v = vec![1,2,3];
let r1 = &mut v;
let r2 = &mut v;
```

This clearly violates one of the Rust rules, namely that we cannot have mutable aliased referenes. Yet, in this example we have two mutable references, `r1` and `r2`, to the same data. With NLL this will compile, as we do not use `r1` after having made `r2`, so its lifetime is implicitly ended right after its declaration. If we write `r1.push(1);` after `let r2`, we get the same error as without using NLL, since the lifetime `r1` overlaps with the lifetime of `r2`.

3.6.2 Trait Objects

When using traits in function signatures or structs we can either make the struct generic over some type that implements the trait, or we can use dynamic dispatch. As generics usually are implemented by copying the source code for the type for each invocation of a new type, it increases code size and compilation time. In addition, collections and similar structures cannot mix different types: a `Vec<SomeTrait>` cannot both contain elements of type `A` and `B`, even if both implements `SomeTrait`.

Dynamic dispatch is the other option. Now variables are *fat pointers*, containing both the pointer to the data type, and a pointer to a `vtable`[4], which contains information about the function addresses for that type, as shown in Fig. 3.2. The entry in the `vtable` is all functions for some trait. With this we can take any concrete type, and follow its vtable pointer, in order to find the implementation of some trait function for that type. In Fig. 3.2, both `Foo` and `Bar` implements some trait which have a function named `fnc`. By following the pointers from the stack, we get the data (left) and the function pointer (right). This way of implementing Trait Objects are usually not mandated by any standard, but it is popular across different language implementations nevertheless.

While trait objects offers greater flexibility in the usage of traits, the pointer jumping leads to worse cache behaviour which may have a large impact on performance, and important compiler optimizations like inlining is impossible.

3.6.3 Specialization

Specialization is a feature which allows multiple implementations of a trait for the same type, where the implementations are ordered by their specificity.

Assume we want to implement the trait `Debug` for a struct that is generic over some type `T`: `Struct<T>`. We might want to have different implementations of `Debug`

[4] the name `vtable` comes from the C++ world, where function on abstract types are called `virtual` functions

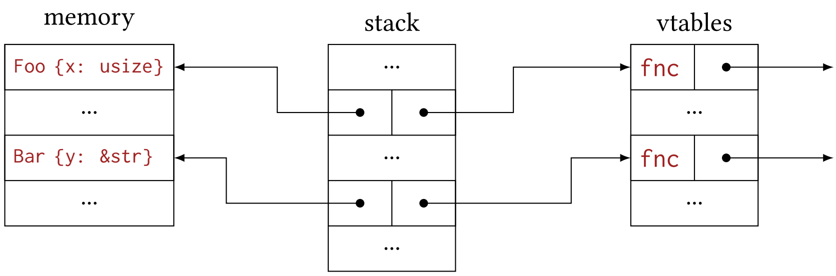


Figure 3.2: Illustration of memory when using Trait Objects.

depending on whether the generic parameter `T` implements `Debug` or not. Specialization makes this possible.

Listing 3.2: Using specialization to implement a trait twice.

```
impl<T> Trait for Struct<T> {
  default fn fmt(&self);
}
impl<T: Debug> Trait for Struct<T> {
  fn fmt(&self);
}
```

With only these two implementation it is clear which of the two we want for any type: if <T> implements `Debug` we want the second, and if it does not, we want the first. However, if we mix in yet another trait, `Clone`, such that we have a third implementation

```
impl<T: Clone> Trait for Struct<T> { ... }
```

it is no longer clear which implementation to use if `T` implements *both* `Clone` and `Debug`. The current implementation forbids such implemementations.

3.6.4 Allocators

The final nightly feature that we look at is *allocators*. It is not yet possible to change the default allocator in stable Rust, but a suggested API for creating new allocators and specifying the default system wide allocator for Rust programs is avaiable by opting in on the allocator feature. The feature defines a trait `GlobalAlloc` that defines functions analagous to `malloc` and `free` from libc, and a attribute `#[global_allocator]` to select which allocator we want to use.

The default allocator for Rust is jemalloc[10]. By using other external crates we can use either the default system allocator, or jemalloc wrapped in our own allocator. This can be useful if we want to do bookkeeping, gather statistics, or do any thread synchronization outside of the actual allocator we are using.

Listing 3.3: Custom allocators wrapping jemalloc and the system allocator

```
pub struct WrapJemalloc;
unsafe impl GlobalAlloc for WrapJemalloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut Opaque {
        ⟨Do something before calling alloc⟩
        Jemalloc.alloc(layout) }
    unsafe fn dealloc(&self, ptr: *mut Opaque, layout: Layout) {
        ⟨Do something before calling free⟩
        Jemalloc.dealloc(ptr, layouer); } }

pub struct WrapSystem;
unsafe impl GlobalAlloc for WrapSystem {
    unsafe fn alloc(&self, layout: Layout) -> *mut Opaque {
        ⟨Do something before calling alloc⟩
        System.alloc(layout) }
    unsafe fn dealloc(&self, ptr: *mut Opaque, layout: Layout) {
        ⟨Do something before calling free⟩
        System.dealloc(ptr, layout); } }
```

CHAPTER 4

CMR

In this chapter we present a concurrent memory reclamation scheme called *CMR*. We define the problem of memory management carefully in Section 4.1, in order to get a complete understanding of which problem we set out to solve. In Section 4.2 we present an abstract overview of CMR, in order to get a high level understanding of the system as a whole, without having to think about technical or implementation details. Section 4.3 discusses the primitives and operations of CMR, and how they are used. Finally in Section 4.4 we argue for the correctness of the system as presented in this chapter. By reasoning about CMR without an implementation we later aim to show that the implementation (Chapter 5) fits the description of the system as we define it in this chapter.

4.1 Problem Definition

We start by defining some central concepts. Memory M is the set of all addresses in the address space of the machine. A *block* is a tuple (a, n) and represents the memory segment $[a, a+n)$. M is a disjoint set $M = A \cap F$ where A is the set of allocated blocks, and F is the remaining of the memory space. F needs not, and is almost never, a consecutive segments, but simply all memory that is outside any allocated block. We call such memory *invalid*, and all memory in an allocated block *valid*. We model the program memory as a graph $G = (A, E)$ where $(u, v) \in E$ iff there is a pointer in u pointing to an address in the segment v. That is, memory blocks are the vertices, and pointers in the program are the edges. See Fig. 4.1 for a possible memory layout with a graph.

As most programs need memory blocks of dynamic size, allocation and deallocation, *"freeing"*, is commonplace. The problem of memory management is to know when it is *safe* to free a memory block. We want to avoid the following memory hazards:

Definition 4.1 (*use-after-free*)**.** Memory that was allocated and then freed is read.

Definition 4.2 (*invalid-read*)**.** Memory that has never been allocated is read.

Definition 4.3 (*double-free*)**.** A block is freed twice without being allocated in between.

```
a = Node { value = 4, next = null }
b = Node { value = 8, next = a }
list = [a, b, 3]
```

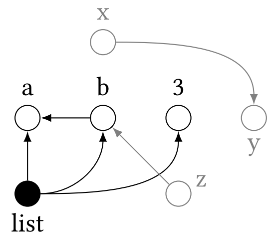


Figure 4.1: Code sample (left) with possible heap layout (right). If the black filled node is the only root, the black nodes are reachable, and the grey nodes are not.

use-after-free is the most hazardous of the three, as program behaviour is often undefined when freed values are read; in many language implementations undefined behaviour means that the entire program is illegal, and one cannot assume anything about its behaviour (see Section 2.3). The consequence of *use-after-free* usually ranges from reading values that are unchanged from the time the block was freed, to mutation of memory that has been reused.

invalid-access is the least frequent of the three, as it requires the programmer to conjure a pointer out of thin air, since it has never been allocated in the system. As with *use-after-free*, this is too is usually undefined, with similar consequences. Despite their similarities we choose to have *invalid-access* as a separate category, as pointer arithmetic may lead to these hazards.

double-free is technically not a memory hazard, as the operating system can check for the validity of pointers that are freed, although this is often not done in practice. It is not clear whether this is due to performance penalties of checking, or if it is primarily a legacy behaviour; POSIXs definition of `free` states that it is undefined behaviour to pass a non-allocated pointer to `free`[13].

We aim to show that CMR guarantees that neither of the three memory hazards are possible.

4.1.1 Shared Memory

Newer languages like modern C++ and Rust aim to avoid having the programmer manage memory manually, due to a long history of the consequences of memory hazards. For single threaded application, this may be considered a problem with suggested solutions. Rusts ownership model and lifetime tracking (Chapter 3), and similar methods from the C++ standard library, are proposed solutions. However, the ownership model does not handle shared memory functionally, as objects in shared memory might not have an owner responsible for its management. Despite not being a complete solution, having “solved” single threaded memory management turns out to be of great help.

We divide up $\mathcal{A}$ into two disjoint parts $\mathcal{O} \cup \mathcal{S}$: *owned* and *shared* memory. Owned memory is all memory which management is already handled by some system, like Rusts ownership model or the smart pointers of C++. Shared memory is the memory in which the structures that is not modelled well by other constructs live, like the nodes in a linked list.

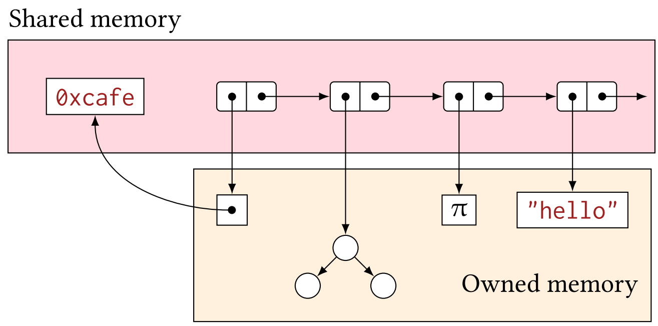


Figure 4.2: Example of memory layout showing owned memory (beige) and shared memory (red). Types in shared memory may contain pointers to owned memory, and vice versa.

A key idea to recognise is that despite data being in Shared memory, they might themselves own data that is in Owned memory, like the binary tree in Fig. 4.2. The destruction of a list node containing the binary tree will utilise the system for owned memory, and make sure that the binary tree is cleaned up properly. It does not matter if the list node itself resided in owned or shared memory. With this distinction we can reduce our problem space significantly, as we only have to worry about the small subset of A that is shared memory. Note that it is also possible to have the data types that are referenced from shared memory but stored in owned memory, like the pointer pointing to `0xcafe` in Fig. 4.2. This includes pointers on a stack frame, but might also include a entry in a hash map. It is these pointers that CMR aims to control.

4.2 Overview

We call a pointer from owned memory to shared memory for a *root*. CMR is based on the idea that if we have access to all roots in the system at an instant, finding the set of all reachable blocks R from the set of roots R_0 is simple: R is the transitive closure of "there is a pointer from x to y" on R_0. We call identifying R *reachability analysis*. By then tracking all allocated blocks A, we can identify the set of unreachable block G by taking the relative complement of R in A: $G = A \setminus R$.

CMR tracks all roots for each thread by restricting where the roots may be stored in memory. This way we know at any time where all roots in the process resides, so they can be collected by any other thread with relatively low effort.

When performing the reachability analysis in a concurrent systems, simply following pointers while maintaining a frontier of unvisited blocks is not sufficient. Since there are multiple threads in the system, some other thread T' may come along and change pointers, causing reachable blocks to be observed as unreachable by the reclaiming thread, as shown in Fig. 4.3. After having read the left child of some node with two children, the two pointers can be swapped by the other thread, causing us to visit one of the nodes twice, as if the two child pointers point to the same node. CMR handles this problem by obtaining a snapshot of the process memory, and performs the reachability analysis on the snapshot.

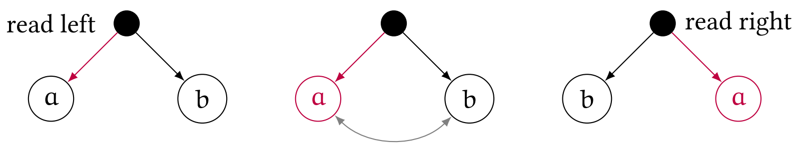


Figure 4.3: Mutation in the memory graph may lead to reachable blocks being observed as unreachable.

The Reclaim procedure shows how we reclaim memory in CMR. The input is the set of allocated address A and information on all threads T. In Get-Roots we collect all roots for all threads. Find-Reachable runs the reachability analysis, and returns the set of all reachable blocks R. We can then find G.

Reclaim(A, T)

Snapshot$()$
$R_0 = $ Get-Roots(T)
$R = $ Find-Reachable(R_0)
$G = A \setminus R$
Free(G)
$A = R$

Find-Reachable(R_0)

Frontier $= R_0$
Seen $= R_0$
while $m = $ Pop$($*Frontier*$)$
    for *ptr* $=$ Pointers(m)
        if *ptr* $\notin$ *Seen*
            Insert$($*Seen*, *ptr*$)$
            Push$($*Frontier*, *ptr*$)$
return *Seen*

4.3 Primitives of CMR

In this section we look at the four data types in CMR, and operations that act on them. All types are generic over some type T, which is omitted for brevity. The operations on these types do not change the generic type.

Definition 4.4 (*Guard*)**.** *Guard* is an object that either contains a root or $\bot$. The *Guard* is non-movable in memory. All roots are stored in *Guards*.

01/06 13:44 “other systems”

The *Guard* is the only type that CMR defines that are different from other systems, and we only use *Guards* to limit the storage of roots.

Definition 4.5 (*Atomic*)**.** *Atomic* is a pointer type that provides safe concurrent access to its users.

Atomic is similar to regular atomic pointers from any programming language; it offers safe reads and writes for concurrent systems.

Definition 4.6 (*NullablePtr*)**.** *NullablePtr* is an immutable pointer that may be $\perp$. It is obtained through a Guard. When a NullablePtr p is obtained from a Guard g, g is immutable throughout the lifetime[1] of p.

The definition of *NullablePtr* is important: it shows that we scope the immutability of a *Guard* to the lifetime of the *NullablePtr*; this allows us to have certain invariants that hold in between changes to a *Guard* to hold for the lifetime of an *NullablePtr*.

Definition 4.7 (*Ptr*)**.** *Ptr* is an immutable pointer that may not be $\perp$. All accesses to shared memory is through a Ptr.

The semantics of *Ptr* are similar to that of *NullablePtr*, but the two are distinct types for simplification of the $\perp$-case.

4.3.1 Operations

A Guard can be constructed with the initial value of $\perp$ with *make-guard*

$$\textsc{Make-Guard} :: () \to \mathit{Guard} \tag{4.1}$$

It can also copy the value of another Guard with *copy-guard*.

$$\textsc{Copy-Guard} :: (\mathit{Guard}, \mathit{Guard}) \to () \tag{4.2}$$

The pointer a *Guard* holds can also be read:

$$\textsc{Read-Guard} :: (\mathit{Guard}) \to \mathit{NullablePtr} \tag{4.3}$$

General usage of *Guard* is to construct the number of *Guard*s one needs for some operation. These *Guard*s are then used to load *Atomic*s into.

Atomic is a regular atomic pointer variable, supporting operations such as *store*, and *compare-and-swap*.

$$\textsc{Store} :: (\mathit{Atomic}, \mathit{NullablePtr}) \to () \tag{4.4}$$

$$\textsc{Compare-And-Swap} :: (\mathit{Atomic}, \mathit{NullablePtr}, \mathit{NullablePtr}) \to \mathit{NullablePtr} \tag{4.5}$$

It is not safe to *load* an atomic, as there is no guarantee that the pointer read is protected by a guard. Instead, CMR defines *load-atomic*, which loads an Atomic into a Guard, and returns the value read as a NullablePtr:

$$\textsc{Load-Atomic} :: (\mathit{Guard}, \mathit{Atomic}) \to \mathit{NullablePtr} \tag{4.6}$$

The NullablePtr is just a convenience type in order to not have to handle the $\perp$ case of all pointers. Whether the pointer is null or not can be checked:

$$\textsc{Is-Null} :: (\mathit{NullablePtr}) \to \mathit{bool} \tag{4.7}$$

Ptr may be used in the place of NullablePtr, since is it just a special case of it. All functions that take a NullablePtr can also take a Ptr.

[1] We use the same meaning of lifetime as Rust (Section 3.3)

4.3.2 Pointer Tagging

CMR also supports using the lower bits of a pointer to store extra information (a *tag*). This is useful for implementing deletion in linked lists, among other things. The tag is read with *tag*,

$$\textsc{Tag} :: (\mathit{NullablePtr}) \rightarrow \mathit{int} \tag{4.8}$$

and a new NullablePtr can be constructed with a given tag using *with-tag*.

$$\textsc{With-Tag} :: (\mathit{NullablePtr}, \mathit{int}) \rightarrow \mathit{NullablePtr} \tag{4.9}$$

The actual address of the pointer is obtained through *addr*

$$\textsc{Addr} :: (\mathit{NullablePtr}) \rightarrow \mathit{int} \tag{4.10}$$

4.4 Correctness

Having defined the types and operations that CMR provides we prove important properties of the system. In this section we may assume that no reclamation pass is happening within the procedure LOAD-ATOMIC:

Claim 4.8. *No reclamation happens while the procedure* LOAD-ATOMIC *is running.*

Lemma 4.9. *If a Guard is valid, then any Ptr read from it is valid.*

Proof. The *Ptr* p is read from a *Guard* g and g is immutable throughout the lifetime of p so they have the same value: $g = p \neq \perp$. □

Theorem 4.10 (*Guard* is valid)**.** *If a Guard is not* $\perp$*, it points to valid memory.*

Proof. The *Guard* got its pointer from an *Atomic* a using LOAD-ATOMIC. We start by showing that a is valid.

If $a \in O$ then a is valid. Else then $a \in S$, so it is accessed through a *Ptr* p, which is read from a *Guard* g'. Since LOAD-ATOMIC mutates g and g' is immutable throughout the lifetime of p (Definition 4.6), $g \neq g'$. Thus a is valid by induction.

Next, since a is valid, it is reachable, and any address reachable from it is also reachable. Since the *Guard* g is protecting the pointer read from a, and since no reclamation may happen during LOAD-ATOMIC, g points to valid memory. □

Lemma 4.11 (*Ptr* is valid)**.** *The Ptr points to valid memory.*

Proof. This follows from Theorem 4.10 and the fact that a *Ptr* cannot be constructed from a *Guard* that is $\perp$. □

Lemma 4.12 and Lemma 4.13 follows, which guarantees that neither of the three memory hazards defined in Section 4.3 are possible in CMR.

Lemma 4.12. *CMR has no use-after-free or invalid-read*

Proof. This follows from Lemma 4.11 as all accesses to shared memory are through a *Ptr* (Definition 4.7). □

Lemma 4.13. *CMR has no double-free*

Proof. $G = A \setminus R$ so only allocated addresses are freed. $A_{i+1} = R$, so freed addresses are discarded from A in each call to RECLAIM. □

Chapter 5

Implementation

In this chapter we look at the Rust implementation of CMR. The source code is openly available on GitHub under some license[6]. license

We start this chapter by implementing the primitive types and operations from Section 4.3.

5.1 Data

In order to better understand how CMR is laid out, we start out by looking at the data. flowcharts Fred brooks

Allocated addresses are stored in a global `HashSet`, `ALLOCS`, which uses a `Mutex` for thread synchronisation. Only addresses in `ALLOCS` are subject for reclamation.

Thread also stores data in Thread-Local Storage (TLS). Each thread maintains a `Vec` of pointers to their `Guard`s, such that collecting all guards is just a matter of iterating through the `Vec`, and following the pointer. Since the data is thread local, no synchronisation is needed when operating on the `Vec`, which makes updates cheap.

One caveat of CMR is that new threads needs to register themselves before using the system. This is done through `cmr::register_thread()`. This initialises thread local data, and pushes a thread handle used in Section 5.3.

5.2 Primitives

Blabla

Guard

The `Guard` is implemented as a single word, in addition to an empty type (the `PhantomData`) as Rust *requires* generic types to be used. `Guard`s aren't normally constructed directly (), but rather declared with the `guard!` macro, which constructs it and calls `Guard::register`. An excerpt of the definitions of `Guard` is shown in Listing 5.1. non movable types somewhere

`Guard::register` gets a mutable reference to the thread local `Vec` of `Guard`s, and inserts a pointer to itself into it. `Guard::drop` (omitted) does the opposite. `Guard::new` is marked `unsafe` since the caller must guarantee to `register` the guard before using

Listing 5.1: Excerpt of Guards definitions

```
struct Guard<T> { ptr: usize, _marker: PhantomData<T> }
impl<T: Trace> Guard<T> {
    pub unsafe fn new() -> Self { Guard { inner: 0, _marker: PhantomData, } }
    pub fn copy_guard(&mut self, other: &Self) { self.inner = other.inner; }
    pub fn register(&mut self) {
        ROOTS.with(|r| { let mut v = r.borrow_mut();
                         v.push(GuardPointer::from_guard(self)) }); }
    ⟨...Remaining methods⟩ }
macro_rules! guard {
    ($var:ident) => { let $var = unsafe { &mut $crate::guard::Guard::new() };
                      $var.register(); } }
```

it. This is normally handled by the `guard!` macro, but there are use cases for calling `new` directly. Usage of the guard is normally as follows:

```
{
  guard!(g);
  let my_num = cmr::alloc(123, g);
} // `g` is dropped here
```

Note that by using `guard!`, the caller only obtains a `&mut Guard<T>`, and not the `Guard<T>` itself; this makes it impossible to move the `Guard` in memory.

Atomic

`Atomic` is mainly a wrapper around Rusts `AtomicPtr`, although the internals differ slightly. CMR defines its own type so that we can control the return types of certain functions. Listing 5.2 shows the definition of the `struct`, as well as `cas`, the compare-and-swap operation, in which we utilise some Traits from the Rust standard library to convert between types. Implementation of remaining methods are straight forward.

Listing 5.2: Excerpt of Atomics definitions (Trait bounds omitted for brevity)

```
pub struct Atomic<T> { data: AtomicUsize, _marker: PhantomData<T>, }
impl<T> Atomic<T> {
    pub fn cas<'a, A, B>(&self, a: A, b: B, ordering: Ordering)
     -> Result<A, NullablePtr<'a, T>> {
        let (old, new) = (raw(a), raw(b));
        let ret = self.data.compare_and_swap(old, new, ordering);
        if ret == old { Ok(A::try_from(NullablePtr::new(ret)).unwrap()) }
        else { Err(NullablePtr::new(ret)) } }
    ⟨...Remaining methods⟩ }
```

NullablePtr

`NullablePtr` is used as the canonical pointer type in CMR, and all pointer like types are converted to `NullablePtr` using the `From` and `Into` traits from the Rust standard library, which handles conversion between types. For instance, we implement `From<*const T> for NullablePtr<T>`. This way we can write functions that are generic over all types of pointers, so that the user of CMR does not have to handle these conversions themselves.

The definition of `NullablePtr` is shown in Listing 5.3, with the `new` and `ptr` methods. Note that we cannot get a reference to the `T` that `NullablePtr` points to; this is because we don't know if the pointer is `null` or not. `ptr` promotes the `NullablePtr` to a `Ptr`, should it not be `null`, by using the `Option` type which Rust provides.

Listing 5.3: Definition of `NullablePtr`

```
pub struct NullablePtr<'a, T: 'a>(usize, PhantomData<&'a T>);
impl<'a, T> NullablePtr<'a, T> {
    pub fn new(u: usize) -> Self { NullablePtr(u, PhantomData) }
    pub fn ptr(self) -> Option<Ptr<'a, T>> {
        if addr(self) == 0 { None }
        else { unsafe { Some(Ptr::new(raw(self))) } }
    ⟨... Remaining methods⟩ }
```

Ptr

`Ptr` provides access to the type it points to, as it is guaranteed to be non-`null`. This is done through the `Deref` trait, which handles the `*` in Rust. Due to auto-deref, we can now use `&Ptr<T>` in place of a `&T`. The definition of `Ptr`, a new of its methods, and its `Deref` implementation is shown in Listing 5.4. Node that both `new` and `get_mut` are `unsafe` methods; `new` because we can not guarantee that the address passed is valid, and `get_mut` because the data may be aliased.

Listing 5.4: Definition of `Ptr`

```
pub struct Ptr<'a, T: 'a> { data: usize, _marker: PhantomData<&'a T> }
impl<'a, T> Ptr<'a, T> {
    pub(crate) unsafe fn new(u: usize) -> Self {
        Self { data: u, _marker: PhantomData, } }
    pub unsafe fn get_mut(&mut self) -> &mut T { &mut *self.as_raw() }
    fn as_raw(&self) -> *mut T { with_tag(*self, 0).data as *mut T }
    ⟨... Remaining methods⟩ }
impl<'a, T> Deref for Ptr<'a, T> {
    type Target = T;
    fn deref(&self) -> &T { unsafe { &*(self.as_raw()) } } }
```

Tagging

By having one canonical pointer type, we can define functions that are generic over all types that supports conversion from and/or to `NullablePtr`. This is used in the functions for pointer tagging, as well as the `cas` in Listing 5.2 (the types `A` and `B`). Listing 5.5 shows some of the free functions for pointer tags that are generic over different pointer types.

Listing 5.5: Implementation of pointer tagging functions

```
TA1 pub fn tag<'a, P, T: 'a>(p: P) -> usize where P: Into<NullablePtr<'a, T>> {
TA2     let n: NullablePtr<T> = p.into();
TA3     n.0 & ones(TAG_BITS) }
TA4
TA5 pub fn with_tag<'a, P, T: 'a>(p: P, tag: usize) -> P
TA6 where P: Into<NullablePtr<'a, T>> + TryFrom<NullablePtr<'a, T>> {
TA7     let p = p.into();
TA8     let n = (p.0 & !(ones(TAG_BITS))) | tag;
TA9     P::try_from(NullablePtr::new(n)).unwrap_or_else(|_e| panic!("failed conversion")) }
```

`ones(k)` returns the bit mask with the `k` lower bits set, and `TAG_BITS` is a predefined number of bits allowed to use for tagging for any pointer. We convert from `P` to `NullablePtr` with `.into()` (TA2). In `with_tag` (TA5) we need to use `TryFrom`, which is a conversion trait that may fail. In CMR `Ptr<T>` implements `TryFrom<NullablePtr>`, where the conversion fails if the `NullablePtr` is `null`. We assert that this failure should never happen (TA9) with the rationale that if we converted some type `P` into a `NullablePtr` and changed its tag, we should be able to convert back to `P`, even though the conversion is not always possible in general.

5.2.1 Free Functions

Having looked at the types and their member functions we now look at the implemetations of important free functions.

We first look at the higher order function `without_reclamation`:

```
pub fn without_reclamation<R, F: FnOnce() -> R>(f: F) -> R {
    let lock = ALLOC_LOCK.lock();
    compiler_fence(SeqCst);
    let ret = f();
    compiler_fence(SeqCst);
    drop(lock);
    ret }
```

The function runs the given closure without having a reclamation pass happening in between. The function simply grabs the reclamation lock before executing; is it however important that the overhead here is as low as possible, as this is used in other important functions. For this reason CMR also has the `without_reclamation_repeat` function, with attempts to run the closure without any synchronization; if a reclamation pass happend while running, we rerun the function.

With the ability to run arbitrary code without a reclamation pass happening in between we can implement `guard`, which is Load-Atomic (Eq. (4.6)).

```
pub fn guard<'a, T>(guard: &'a mut Guard<T>, a: &Atomic<T>) -> NullablePtr<'a, T> {
    without_reclamation(|| { let p = unsafe { a.load(SeqCst) };
                             guard.inner = ptr::raw(p);
                             p }) }
```

Since we guarantee that no pass happend in between reading the `Atomic` and protecting the data it pointed to in the `Guard`, we know that the data is still valid.

Another important function is `alloc`, which allocates memory:

```
pub fn alloc<T: Trace>(guard: &mut Guard<T>, t: T) -> Ptr<T> {
    let ptr = alloc::alloc(t);
    guard.inner = ptr::addr(ptr);
    alloc::register(ptr);
    ptr }
```

Note that we do not need to use `without_reclamation` here, since the newly address is protected by the `Guard` before being registered; recall from Section 5.1 that only registered allocations are subject for reclamation.

5.2.2 Correctness

We argue for the correctness of the primitives as presented with respect to the definitions from Chapter 4.

Claim 5.1. *Claim 4.8 is attainable.*

Proof. `guard` implements Load-Atomic with the wanted semantics. □

Claim 5.2. `Guard` *satisfies Definition 4.4.*

Proof. The `Guard` is constructed with `null`, and gets values from `Atomic`s using `cmr::guard`; the values read are roots. Using the `guard!` macro it is impossible to move the `Guard`. □

Claim 5.3. `NullablePtr` *satisfies Definition 4.6.*

Proof. The type does not expose any mutating methods, so it is immutable. Looking at the function `guard` we see that the `&mut Guard` is mutably borrowed, and since the lifetime of the `NullablePtr` returned has the same lifetime, the `Guard` is borrowed for the lifetime of the `NullablePtr` □

Claim 5.4. `Ptr` *satisfies Definition 4.7.*

Proof. The type does not expose any mutating methods, so it is immutable. Since `Ptr` is the only type implementing `Deref` and no function return a `&T`, all accesses to `T`s must be though the `Ptr`. □

We argue that since the primitives defined in Chapter 4 are implemented with the defined semantics the results from Section 4.4 holds for the Rust implementation of CMR.

5.3 Snapshot

For obtaining a snapshot of the process memory CMR utilises a operating system features offered by POSIX compliant systems: *forking*. Calling `fork()` makes a copy of the current process, called the *child process*. The return value of `fork()` determined whether we are in the child or parent process. In the child process, only the thread that called `fork()` continues its execution. For this reason, we need to perform some work before forking. Most importantly, the threads needs to tell the reclaiming thread where to find their `Guard`s. To do this we use a second POSIX feature: *signals* (see Section 2.2.3).

Listing 5.6: Thread signaling

```
fn signal_threads_except_self() -> usize {
    let mut count = 0;
    let me = thread_id();
    let mut th = THREAD_HANDLERS.lock().unwrap();
    th.retain(|&th|
      if th == me { true }
      else { unsafe {
            let val = libc::sigval { sival_ptr: std::ptr::null_mut() };
            let r = libc::pthread_sigqueue(th as u64, libc::SIGUSR1, val);
            if r == 0 { count += 1; true }
            else { false } } });
    count }
```

Using POSIX threads (`pthreads`) signals we register a signal handler for the `SIGUSR1` signal with the `sigaction` call, and the reclaiming thread signals all threads with `pthread_sigqueue`. This is done through the Rust library `libc`, which provides Rust bindings to the C standard library. The procedure for signalling all registered threads is shown in **??**. Here we actually do two things at once: in addition to signalling the threads, we remove the thread handlers that we fail to signal. The procedure returns the number of threads we successfully signalled, so the caller knows how many threads to expect being in the signal handler. Pseudo code for the signal handler is shown in Listing 5.7.

Listing 5.7: Pseudocode for the signal handler used by CMR

```
SH1 id = sh_enter_counter.fetch_add(1)
SH2 write_out_data_to(thread_datas[id])
SH3 sh_done_counter.fetch_add(1)
SH4 while sh_frozen.load():
SH5     wait()
SH6 sh_enter_counter.fetch_sub(1)
```

We use `sh_enter_counter` to keep track of how many threads are present in the signal handler; the reclaiming thread knows how many threads it successfully signalled, so it knows how many threads to expect. (SH1) registers a threads presence, in addition to giving each thread a unique index in the range $[0, n)$, where n is the number of threads signalled. This is used in (SH2), where each thread writes out their guards and allocations into the global vector `thread_datas`. We then register that we have written our data (SH3), and wait for the reclaiming thread to unfreeze us (SH5). At last we register that we have seen that we are done (SH6), so that no thread risk being stuck in the next iteration of the reclaiming procedure, waiting again on the `sh_frozen` flag.

5.4 Reachability

Reachability analysis is a straight forward implementation of the FIND-REACHABLE procedure from Section 4.2, and is shown in Listing 5.8. We maintain one `HashMap` (`FR2`) for all blocks we have seen, and a `VecDeque` (`FR3`) for the queue of blocks we want to visit. For efficiency reasons we write out the reachable set when we find a new block, instead of collecting up the blocks and writing it in one iteration (`FR10`). This implementation has capped the number of pointers a single type can write out to be 32 (`FR6`); while this is not sufficient in the general case, most types only require one or two pointers.

Listing 5.8: Rust implementation of FIND-REACHABLE

```
FR1  fn mark_and_sweep(mut cursor: Cursor<&mut [u8]>, roots: Vec<TraitObject>) -> usize {
FR2      let mut seen = HashMap::new();
FR3      let mut queue = VecDeque::new();
FR4      ⟨insert roots into seen and queue⟩
FR5      let mut num_ptr = 0;
FR6      let mut ptr_buffer: [TraitObject; 32] = unsafe { std::mem::zeroed() };
FR7      while let Some(to) = queue.pop_front() {
FR8          let addr = to.data as usize;
FR9          let t: &ptr::Trace = unsafe { ::std::mem::transmute(to) };
FR10         ⟨write out addr to the cursor⟩
FR11         num_ptr += 1;
FR12         let n = t.write(&mut ptr_buffer);
FR13         for i in 0..n {
FR14             let (to, addr, vtable) = ⟨destructure ptr_buffer[i]⟩
FR15             if seen.insert(addr, vtable).is_none() {
FR16                 queue.push_back(to); } } }
FR17     num_ptr }
```

5.4.1 Trace

Finding pointers in arbitrary data types might involve significant work since the size of the data types can be arbitrarily large. In addition, memory might not be initialised, and false positives might occur if we are not careful. Instead of scanning through the memory block linearly, CMR defines the Trait `Trace`, which all data types that is stored in shared memory must implement. A type implementing `Trace` knows a bound on how many shared memory pointers it contains, and can write these out to a buffer. For instance, a `Node` in a single linked list contains only one pointer, namely its next pointer, which is trivial to write out.

The implementation of this uses *Trait Objects* (Section 3.6.2), which involves dynamic dispatch. This solution is potentially expensive, as it may involve cache misses in the I-cache, although the number of misses is limited by the difference in data types in shared memory, which normally is smaller than in Rust memory. Listing 5.9 shows the `Trait` as well as a sample implementation for a node in a linked list. This implementation uses *specialization* (Section 3.6.3) as the implementation of `Node`s containing data that itself is Trace is different.

Listing 5.9: Definition of the Trace trait and a sample implementation for a linked list node.

```
T1  pub trait Trace { fn count(&self) -> usize { 0 }
T2                    fn write(&self, &mut [TraitObject]) -> usize { 0 } }
T3  pub struct Node<T> { data: ManuallyDrop<T>,
T4                       next: Atomic<Node<T>> }
T5  impl<T> cmr::Trace for Node<T> {
T6      default fn count(&self) -> usize { 1 }
T7      default fn write(&self, slice: &mut [TraitObject]) -> usize {
T8          let p = unsafe { self.next.load(SeqCst) };
T9          if !p.is_null() { slice[0] = ptr::trait_object(p);
T10                           1 }
T11         else { 0 } } }
```

Trace contains default implementations of the two functions, such that primitive types can easily implement it. write takes a buffer, writes all pointers to it as TraitObjects, and returns the number of objects written. count gives an upper bound on the number of pointers written. This is useful for collection types, like Vec or HashMap, which also may contain pointers to shared memory.

Node is a standard node from a linked list, containing data, and a next pointer. The implementation of write loads the next pointer (T8), which is an unsafe operation, as there is no Guard protecting the pointer. This is safe in the context of the reclaiming thread since the memory will be freed at earliest when we finish the reachability analysis, and at that point we no longer read the memory. The implementation only writes out the pointer if it is non-null; while this is not required for CMR to function, it simplifies the logic in the reachability analysis.

5.4.2 Destructors

Since Rust uses the RAII pattern extensively, we would like to run the destructors of type when we free memory of that type. However, due to constraints in the the type system, this has shown to be difficult to implement. We would like to have a single function foo<T> that, based on whether the generic type T implements Drop or not to run have two different implementations. There is no implemented solutions in the type system that allows this. See [3] for discussion on the topic. The main difference between this and Trace is that CMR requires all types to implement Trace, but implementing Drop is optional.

5.5 Complications

A number of implementation complications arised while deleveopting CMR. We look at a few of them here.

5.5.1 Locks in libc

In order to protect programmers from deadlocks, POSIX defines a subset of functions as *async-signal safe*, meaning they are safe to call from a signal handler. Functions that are async-signal safe includes `time()`, `open()`, and `mkdir()`, but it does *not* include `malloc()`. As such, allocation in signal handlers is not safe, and is a source of deadlock bugs. This itself was not a large problem for CMR, as its signal handler did not require any allocation. However, as threads are frozen by the reclaimer in a signal handler, it is also not safe for the *reclaimer* to call `malloc`, despite not being in a signal handler itself. This is because some thread may be in the process of allocating memory, and have aquired a lock internal to libc, right before being signaled. The thread is still holding the libc lock and is frozen in its signal handler by the consolidator, which prevents *all threads*, even those oblivious to CMR, from allocating.

This problem is not solved properly by CMR, but its effects are mitigated by wrapping the general allocator in Rust to go through yet another lock, the `alloc_lock`, which can be aquired by the reclamating thread (this is why we have (R3) in ??). This prevents most allocations of deadlocking, but not all. Rust uses `pthreads` internally for thread handling on Linux, which allocates internally, both in `spawn` and `join`. The former may be circumvented by aquiring the allocation lock before calling it, but this is no solution for the latter, since the thread may depend on allocating before exiting.

5.5.2 SignalVec

For performance reasons threads store their allocation in TLS in between reclamation passes in a `Vec`. The allocations are later collected in the threads signal handler when some thread wants to reclaim memory, as the reclaiming thread requires access to all allocations in the process. However, due to the asynchronous nature of signals we run into problems if a thread is in the middle of some operation on the `Vec` when it is signaled, since the vector is copied and `clear()`ed in the signal handler.

To handle this compilication we made `SignalVec`, a `Vec` that supports asynchronous `clear()`s. The implementation is a standard `Vec` implementation except that we use `cas` to increment the `length` field of the `SignalVec`, such that we can detect the case where `clear()` was called in the middle of `push`. Since we are writing to the last element, we do not risk overwriting any values. The `SignalVec` is specifically designed to only work in the exact use case that CMR requires.

5.5.3 Thread Registration

Thread startup and shutdown has shown to be a difficult problem to handle, especially when carefully managing when threads are allowed to allocate. Early attempts were made to automate this using lazy initialization of thread local variables, but controlling allocations in these, or guaranteeing the order of initialization was problematic. Testing with continously checking whether the thread was initialized in CMR methods showed that it imposed too much overhead for the stragety to be viable.

Chapter 6
Methodology

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

6.1 Development

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

6.2 Testing

Testing is an important part of sfotware development. While formal methods have not yet made its way into the software development insdustry, simpler and more heuristic methods, like unit testing and integration testing, are widespread. However, as Edsger Dijkstra famously said: “Testing shows the presence, not the absence of bugs”[5]; while testing is useful to improve the quality of software, it is far from sufficient.

In the world of concurrent programming this is even less so. Many bugs are manifested through unfortunate[1] thread execution interleavings done by the scheduler. We try to reveal these interleavings by repeatedly running tests until our confidence that no such interleavings exists is sufficiently high. In addition, we run tests with tools such as Valgrind[20] and our own sanitizer (Section 6.2.1). Tests were also ran with and without compiler optimizations, as these optimizations often reveal yet more bugs.

6.2.1 Sanitizer

To automate validation of pointer reads we made a compile time feature[2] called `sanitize` that tracked all allocations, frees, and pointer reads. Allocations and frees were tracked in two `HashMap`s, `ALLOCATIONS` and `FREES`. On each new allocation, we insert it into the `HashMap` while asserting that it was not there previously. We also remove it from the frees map, in case it had previously been allocated and freed. Since we are using a custom pointer type, `Ptr`, checking the validity on each pointer access is possible, as shown in Listing 6.1.

```
pub fn alloc<'a, T: Trace>(t: T) -> Ptr<'a, T> {
    let addr = B::into_raw(B::new(t)) as usize;
    #[cfg(feature = "sanitize")] {
        let mut a = ALLOCATIONS.lock().unwrap();
        assert!(a.insert(addr));
        let mut f = FREES.lock().unwrap();
        f.remove(&addr); }
    unsafe { Ptr::new(addr) } }
```

Listing 6.1: Verifying all pointer accesses with `sanitize`

```
impl<'a, T> Deref for Ptr<'a, T> {
    type Target = T;
    fn deref(&self) -> &T {
        #[cfg(feature = "sanitize")] {
            let a = ::alloc::ALLOCATIONS.lock().unwrap();
            if !a.contains(&addr(self)) {
                let was_freed = ::alloc::FREES.lock().unwrap().contains(&addr(self));
                panic!("{:x} is not valid. Was is freed? {}", self.data, was_freed); } }
        unsafe { &*(self.as_raw()) } } }
```

6.3 Benchmarking

The benchmarks are ran with 5 second trials, where a function is ran repeatedly for the duration with any specified number of threads. The number of executions is counted for each thread and the total operations per second is reported. All threads run the same code, but they may have different thread local data. This is useful when benchmarking `HashMap::insert`, so that the threads can inserts values with different keys.

[1] Some would call interleavings that reveal bugs fortunate
[2] *features* are similiar to `#ifdef`s in C and C++

There are a number of pitfalls when it comes to benchmarking code. We discuss a few of them; [4] is a good resource for experimental testing of data structures.

Initialization of data structures should not be done on a single core as this creates a strong skew of data locality for that core, and other cores will have reduced performance due to the data locality. This is especially important on systems with multiple CPU sockets.

6.3.1 Trench

In order to more effectively benchmark threaded applications, an open source benchmarking library called `trench`[19] was developed. The library handles thread management and state for the runs of the benchmark. Trench supports both mutable thread local state and immutable shared state between all threads. For CMR this is useful since we can put the data strucutres we want to benchmark in the immutable shared state, as neither of the operations we want to test are `&mut self` (see Section 3.5.1). The user specifies the function to be benchmarked, the number of threads, and the states, and the duration of the benchmark, and `trench` handles the rest. The number of runs of the function specified during the given duration is measured. Listing 6.2 shows the benchmark for `HashMap::insert`. `RandomSource` allows us to pregenerate random numbers that we can insert into the hashmap, such that the random number generation itself is thread local, and is not included in the benchmarking loop.

Listing 6.2: `Hashmap::insert` benchmark using `trench`

```
fn hashmap_insert(num_threads: usize) {
    fn func(state: &HmState, local: &mut RandomSource<u64>) {
        state.hashmap.insert(local.next(), 0); }
    let b = trench::TimedBench::<HmState, RandomSource<u64>>::with_threads(num_threads);
    b.with_local_state(|l| { cmr::thread_activate();
                             l.gen_n(10_000_000); });
    let res = b.run_for(duration(), func);
    b.with_local_state(|_| cmr::thread_deactivate());
    println!("cmr::HashMap\tinsert\t{} ops/sec", fmt_thousands_sep(res.ops_per_sec)); }
```

The `with_local_state` function runs the closure on each thread in parallel; this is used both for initializing the local state, and for thread local initialization and destruction. The global and local states, `HmState` and `RandomSource`, implements the trait `Default`, so that we do not have to initialize it ourselves.

Chapter 7

Usage of CMR

In this chapter we look at usage code for CMR. The goal of this chapter is twofold: we want to look closer at how the abstractions that CMR provides are used, and how difficult they are to use; we also believe that showing the implementations of the data structures we can further reason about the performance characteristics and pitfalls for the results obtained in Chapter 8. 29/05 08:36 Write more here

We have implemented four data structures: a stack, a queue, a list, and a hashmap, and the implementations will be considered in sequence.

7.1 Lock-free Stack

We begin by looking at an implementation of a conurrent stack, which is arguably the simplest concurrent data strucutre. The stack is the well known Treiber Stack from [18].

The definitions of the `Stack` and `Node` structs and the two most important operations on a stack, `push` and `pop`, is shown in Listing 7.1. We look at each one in turn. Construction of the stack is omitted for brevity, since an empty stack just has a `null` pointer as its top node.

Push

`push` allocates the stack node itself, so it takes the value we want to push onto the stack 06/05 19:14 After writing 2.4.1, revisit this

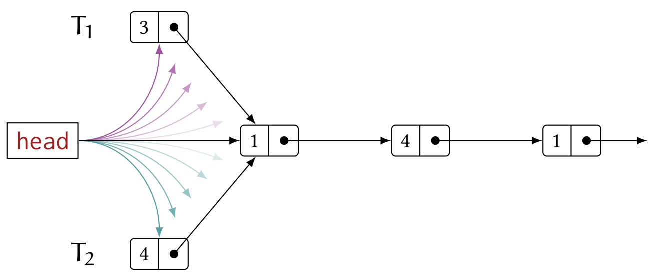


Figure 7.1: T_1 and T_2 both tries to swap the `head` pointer towards their node.

Listing 7.1: Stack::push and Stack::pop

```
ST1  struct Stack<T> { top: SharedGuard<Node<T>>, }
ST2  struct Node<T> { data: ManuallyDrop<T>, next: Atomic<Node<T>>, }
ST3
ST4  impl<T> Stack<T> {
ST5      pub fn push(&self, t: T) {
ST6          guards!(_new_top, _top);
ST7          let mut new_top = cmr::alloc(_new_top, Node::new(t));
ST8          loop { let top = cmr::guard(_top, &self.top);
ST9                 unsafe { new_top.get_mut().next = Atomic::new(top); }
ST10                if self.top.cas(top, new_top, SeqCst).is_ok() { break; } } }
ST11
ST12     pub fn pop(&self) -> Option<T> {
ST13         guards!(_top, _next);
ST14         loop { let top = cmr::guard(_top, &self.top).ptr()?;
ST15                let next = cmr::guard(_next, &top.next);
ST16                if self.top.cas(top, next, SeqCst).is_ok() {
ST17                    let node = unsafe { top.move_out() };
ST18                    return Some(node.data()); } } } }
```

(ST5). We start out by declaring two Guards (ST6): one for the new node we allocate, and one for the head of the stack. We must protect the head of the stack, since the node may be removed after we read its address, and we would have a dangling pointer. Next we allocate a new node (ST7), which is done outside the retry loop so that we only have to allocate one time per call to push. Now we enter the retry loop, which we repeat until we succeed in changing the top pointer of the stack to our new node. The top node is read (ST8), and the next pointer of the new node is set to the head (ST9). If we succeed of chaning the top pointer of the stack to our new node, we break out of the loop and return (ST10). If not, we retry until we do.

Pop

pop is similar to push. We declare two Guards (ST13), but this time they are for the first and second node in the stack. We read the top pointer (ST14), and return from the function if it is null using the ? Rust operator. We then read the next pointer of the node (ST15); here the null case is the same as the non-null case. We try to swap the head pointer from the first to the second node (ST16); if we fail we restart, and if we succeed we move out the Node from the Guard. This is an unsafe operation, as the type is copied out of its original place, effectively aliasing it. At last, the data is returned.

As an example of why reading and returning the node data is unsafe in the general case, consider two threads T_1 and T_2 using a Stack<Box<T>>. T_1 is looking at a node n, and T_2 is popping n from the stack, getting the Box<T> back from it. Now T_2 drops the Box, which frees the pointer. If T_1 decides to look at the data in n, it will dereference a freed pointer, which is a use-after-free (Definition 4.1). Despite being unsafe in the general case, it is safe for the implementation of the stack as presented, since no operation on the stack looks at the data of a node, except in (ST17), where only one thread may be for any given node, since we succeed the cas operation.

7.2 Lock-free Queue

The queue implemented is based on the well known Michael-Scott Queue from [12]. The idea behind the queue is to have a sentinel node as the first node of the queue in order to avoid difficult edge cases when the queue is empty. The sentinel node is the greyed out node in Fig. 7.2.

The `Node` and `MsQueue` struct definitions are omitted, as they are very similar to those of the `Stack`. The main difference is that in the `Queue` we maintain both the `head` and `tail`. The following invariants hold for the `Queue`: `head` is never `null`, at any instant `tail` is either the last, or second last node in the queue.

`push` is shown in Listing 7.2; `pop` is omitted due to its similarity with `Stack::pop`.

Listing 7.2: The `push` operation on a Michael-Scott Queue.

```
MS1  impl<T> MsQueue<T> {
MS2      pub fn push(&self, t: T) {
MS3          guards!(_new_node, _tail, _next);
MS4          let new_node = cmr::alloc(_new_node, Node::new(t));
MS5          loop { let tail = cmr::guard(_tail, &self.tail).ptr().unwrap();
MS6                 let next_ptr = &tail.next;
MS7                 let ptr = cmr::guard(_next, next_ptr);
MS8                 if ptr::addr(ptr) != 0 { let _ = self.tail.cas(tail, ptr, SeqCst); }
MS9                 else if next_ptr.cas(ptr::null(), new_node, SeqCst).is_ok() {
MS10                    let _ = self.tail.cas(tail, new_node, SeqCst);
MS11                    break; } } } }
```

We start out by declaring three `Guards` (MS3): one for the new node, one for the current tail, and one for the tails `next` node, which may be present. We load `tail` (MS5), and its `next` pointer (MS7). Since the Michael-Scott queue is always non-empty, we know that the `head` is non-`null`, and it is therefore safe to promote the `NullablePtr` to a `Ptr` using `.unwrap()`. If the next pointer is non-`null` the node we believed was the tail was not the tail after all. We try to swing `tail` from the node we read, to its next node (MS8) before restarting. If the tail was `null` we try to `cas` its next field from `null` to our new node (MS9). If we succeed, we `cas` the tail to our node and exit. If we fail, we restart. Note that we do not check the results of the the `cas` where we set the tail to the node we just inserted; if this operation fails, it just means that some other thread came along and noticed that `tail` was not the real tail, and `cas`ed it to the last node (MS8).

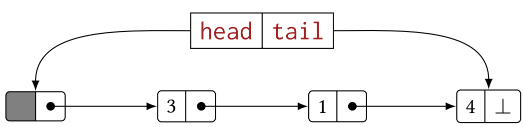


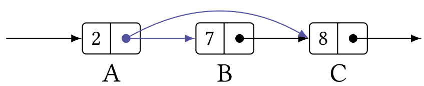
Figure 7.2: The Michael-Scott Queue. The first node in the queue is a sentinel node.

7.3 Lock-free List

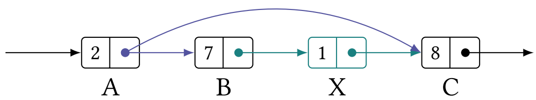
07/05 22:22 insert more stuf here?

Michael introduced a concurrent List in [9]. The list is similar to the Stack from Section 7.1, but we support more operations than `push` and `pop`, including queries and removals, and insertions into arbitrary points in the list. Definitions of `List` are very similar to that of the `Stack` and `Queue`, and is therefore omitted.

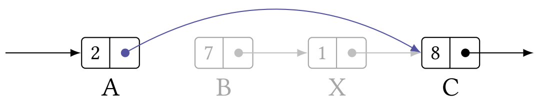
Having arbitrary `insert` and `remove` opens for a problem known as *double-remove*, shown in Fig. 7.3. Let there be two threads in the system T_1 and T_2, and let A, B, and C be three consecutive nodes in the list. If T_1 wants to remove the B node, there is a small window in which T_2 may insert a new node, X, between B and C. When T_1s `cas` operation succeeds — note that `A.next` was not touched by T_2 — it will accidently swing the pointer past the new node X without noticing. This is a variant of the ABA problem (Section 2.4.2).



(a) T_1 calls `cas(A.next, B, C)`.



(b) T_2 inserts X before the `cas` is ran.



(c) The `cas` succeeds. Both B and X are removed.

Figure 7.3: Double removal with `List::remove`.

A solution to this problem, as shown by Harris in [11], is to exploit memory alignment on modern CPU architectures: `struct`s are *aligned* in memory, meaning their address is a multiple of some power of two. This causes the least significant bits of their address to always be zero; bits that may be used for other purposes. We use the least significant bit in the `.next` field in a node for a *tag*[1], signaling that the node is logically deleted, and should not be acted upon. To see how this helps the problem as shown above, T_1 would start out the deletion process of B by calling `cas(B.next, C, with_tag(C, 1))`. Should this fail, T_1 can just read `B.next` again, and retry. When it succeeds, it may try to `cas` `A.next` over B to C. Now T_2 realizes that it should not insert X between B and C, since it reads the tag of B, realizing that it was deleted.

[1] Now we finally understand why CMR supports pointer tagging from Section 4.3.2

7.3.1 The Entry API

Many of the most interesting operations on the List involves iterating through the list. Due to the ownership and lifetime rules that Rust imposes, it may be tricky to implement typical iteration and juggle pointers around. For this reason, the API uses an indirection for iterating through the list: `Entry`.

Listing 7.3: Partial `Entry` API from the List implementation.

```
pub struct Entry<'a, T: 'a> {
    current: &'a mut cmr::guard::Guard<Node<T>>,
    previous: &'a mut cmr::guard::Guard<Node<T>> }
impl<'a, T> Entry<'a, T> {
    pub fn step(&mut self) -> Result<T>;
    pub fn current(&'a self) -> cmr::NullablePtr<'a, Node<T>>;
    pub fn previous(&'a self) -> cmr::NullablePtr<'a, Node<T>>;
    pub fn insert_between(&self, new_node: ptr::Ptr<Node<T>>) -> Result<T>;
    pub fn delete(&self) -> Result<T>;
    pub fn seek_with<F>(&mut self, f: F) -> Result<T> where F: Fn(&T) -> bool; }
```

An `Entry` is like a pointer into the list, which can `step()` to the next node, get a pointer to the `current()` node, `remove` the current node, insert a new node between (`insert_between`) two nodes, and find nodes which data satisfies arbitrary closures `Fn(T) -> bool`. Since there is some overhead in declaring a `Guard`, `Entry` contains two references to `Guard`s rather than the `Guard`s themselves. This makes construting a `Guard` nearly free. Another implication of this is that `Entry` is movable in memory (as `Guard` is not).

This indirection simplifies many operations, and we barely need to deal with lifetime and ownership issues, although it almost requires NLL (Section 3.6.1) to use.

Listing 7.4: Implementation of `List::for_each` using the `Entry` API.

```
pub fn for_each<F: Fn(&T)>(&self, f: F) {
    guards!(_a, _b);
    let mut entry = self.entry(_a, _b);
    while let Some(ptr) = entry.current().ptr() {
        f(ptr.data());
        if entry.step().is_err() { break; } } }
```

7.4 Lock-free Hash Table

The hash table is a versatile and popular data structure. It is widely used due to its constant time operations, including queries, insertions, and removals.

Lock-free hash tables are interesting for the same reasons. Despite the interest, designing a concurrent hash table turns out to be a difficult problem. Blabl, resize Most hash tables are split up in *buckets*, such that the hash of the elements within a bucket share some property (eg. a common prefix). Increasing the number of buckets is knows as *resizing*, which makes sure that the number of elements in each bucket is limited; many algorithms and hash functions give bounds on the number of elements in each bucket.

7.4.1 Split-Ordered List

We start by describing the *Split-Ordered List*. Split-Ordered Lists were introduced in [16]. The nodes in the list is ordered by the *reverse hash* of the value in the node. In addition, the list contains *sentinel nodes*, which are the beginnings of the buckets in the hash table. By making the number of buckets $b = 2^k$ we can double b when the load factor is too high, and insert one more sentinel node between each of the nodes already present, effectively differentiating between one new bit of the reverse hash.

7.4.2 Hash Table

07/05 17:00 Clean up these sections. Maybe have only one section?

Using the Split-Ordered List we can implement a concurrent hash table by having an array of pointers to sentinel nodes, and a "size" of the bucket array. If a sentinel pointer is `null`, then the node is not yet in the table. When inserting a new element into the table, we first find the sentinel node that precedes the node we want to insert (the *parent*); this is known, since we know the ordering of the nodes in the list — the reverse hash. However, due to the resizing method, the parent may not have been inserted yet. If not, we can simply recurse on the insertion method, and insert the parent first. Then, we simply iterate through the list, and find the place in which our new node should be. Assuming a small load factor, this is a fast operation.

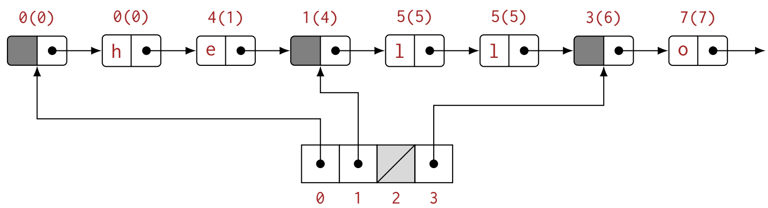


Figure 7.4: The Split-Ordered List. Node labels shows the `hash` and its reverse in parenthesis.

Fig. 7.4 shows the split-ordered list with a table size of 4. The nodes in the list are ordered by the reverse of their hash (shown in parenthesis). Given a node `n`, we find the sentinel node that should preceed it in the list by taking `hash(n) % table_size`. Note that this is not the reverse hash. For instance, inserting a node where `hash(n) == 7`, we look in bucket `7 % 4 == 3`, and iterate from sentinel node 3. Inserting a node where `hash(n) == 10`, we would get `bucket = 2`, which is `null`, so we need to insert the sentinel node first.

We look at n operations on the hash table: `contains`, `insert`, $and\ maybe\ more$.

7.4.3 Contains

Listing 7.5 shows the implementation of `HashMap::contains`. The implementation of utility functions such as `bucket_and_revhash` are omitted for brevity. We find the `parent` node (HC4), and use the `Entry` API from `List` (HC6) to look for the first node which hash and key is the same; if we encounter a node which hash is more than our node, we know that we have gone too far. `Entry::seek_with_opt` lets us break out of the search early by returning `None` (HC10). If we find a node with both the right hash and the right key, we return `Some(true)` from the closure (HC11), and `seek_with_opt` returns `Ok`. If we get back `Ok`, the search succeeded, so we return `true`, and `false` otherwise.

Listing 7.5: Implementation of `HashMap::contains`.

```
HC1  impl<K, V> HashMap<K, V> {
HC2      pub fn contains(&self, k: &K) -> bool {
HC3          let (bucket, rev_hash) = self.bucket_and_revhash(k);
HC4          let curr = self.get_or_insert_bucket(bucket);
HC5          guards!(_curr, _prev);
HC6          let mut entry = list::Entry::from_node_ptr(curr, _curr, _prev);
HC7          entry.seek_with_opt(|data|
HC8              Some(match data {
HC9                  &Entry::Value((h, ref key, _)) => {
HC10                     if h > rev_hash { return None; }
HC11                     else { h == rev_hash && k == key }
HC12                 }
HC13                 _ => false })
HC14         ).is_ok() } }
```

7.4.4 Insert

`HashMap::insert` is more complicated, as there are multiple things that can go wrong, and that some operations require cleanup. Listing 7.6 shows the implementation of `insert`.

31/05 15:43 fix HI numbers here

Listing 7.6: Implementation of `HashMap::insert`.

```
impl<K, V> HashMap<K, V> {
    pub fn insert(&self, k: K, v: V) {
        let (bucket, rev_hash) = self.bucket_and_revhash(&k);
        let curr: cmr::Ptr<_> = self.get_or_insert_bucket(bucket);
        guards!(_new_node, _curr, _prev, _r1, _r2);
        let node_data = Entry::Value((rev_hash, k, v));
        let mut new_node = cmr::alloc(_new_node, list::Node::new(node_data));
        'restart: loop {
            let mut entry = list::Entry::from_node_ptr(curr, _curr, _prev);
            let res = entry.seek_with(|e| match e {
                &Entry::Value((h, ref key, _)) => h >= rev_hash,
                &Entry::Sentinel(h) => h > rev_hash });
            if let Err(list::Error::Empty) = res {
                ⟨End of list case⟩ }
            if res.is_err() { continue 'restart; }
            if entry.insert_between(new_node).is_err() { continue 'restart; }
```

```
HI17             ⟨Remove other nodes with the same key⟩ }
HI18         self.increment_length(); } }
```

We start out by hashing the key, finding the reverse hash (HI3) and the bucket of the sentienl node, and a pointer to the node is acquired (HI4). We declare five (!) Guards (HI5), and alloc our new node (HI7). Next we make our entry from the sentinel (HI9), and find the correct place to put our new node (HI10). The new node is put before any other nodes with the same hash, but after the sentinel node, should their hashes be the same. We insert the new node in front of the old nodes so that other threads will see the most recently updated node first. The result of this operation has three cases: 1) we fail with Empty which means we got to the end of the list, and is handled by inserting the new node at the end of the list (HI14), 2) we fail with another failure case and restart (HI19), and 3) we succeed and actually insert our new node into the list (HI20), where we, again, restart upon failure.

After insertion we must check for other nodes with the same key, since there should only be one entry for any given key in the map (HI21). This is done by making a new Entry with the new node, stepping once, so that the current node is not our new node, and delete() any node that has the right key. When we hit a node which hash is more than our own, we are done (HI24). Since the map is generic over the key type K, comparing the type for equality might be expensive. Therefore, we check the hash before checking the key for equality (HI25).

HashMap::insert contains a single unsafe block: (HI14). This operation is unsafe since we are mutating new_node, which potentially could cause a race condition if the node was concurrently read by another thread. However, we know that it is not read by any other thread, since we have not been succesfull yet in inserting it into the list.

7.5 Crossbeam Integration

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 8

Results

In this chapter we look at the expermental results of the system as a whole. In Section 8.1 we look at the overhead of the operations done by CMR. In Section 8.2 we look at the performance of the data structuers implemented, with and without the overehead CMR, and compare them to alternatives in the Rust ecosystem, like Crossbeam[7] and data structures in the standard library wrapped in a `Mutex`.

8.1 Operations of CMR

The operations that CMR provides that are most interesting to look at is allocation (`cmr::alloc`) and guard initialization and destruction (`guard!`), as these operations are the only ones that have any significant overhead. Atomic loads pointer manipulations are mainly tricks of the type system to ensure the safey of the operations, and has no run-time overhead.

It is also interesting to look at the overhead of `fork()` calls, as this is an important part of the way CMR works.

The results are primarily from a . We have also ran the full benchmark suite on other machines. Interesting results are shown. The complete dataset of all results on all machines are listed in Appendix A

30/05 13:57 insert IST machine here

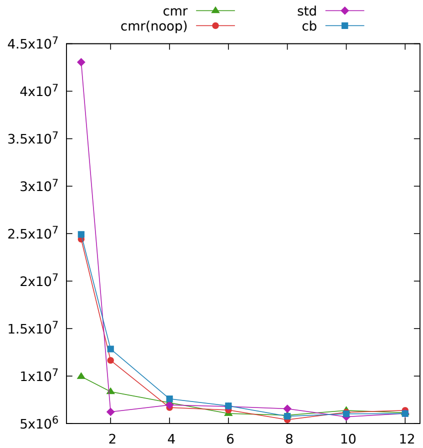
30/05 13:58 consider this after getting IST results

8.2 Data Structures

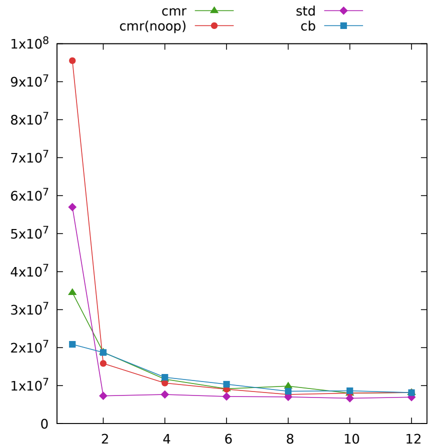
We look at all data structuers in the order that they were introduced in Chapter 7.

30/05 14:23 remove clearpages

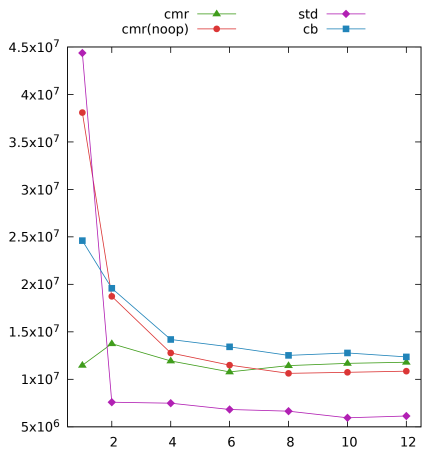
8.2.1 Stack



(a) `Queue::push`



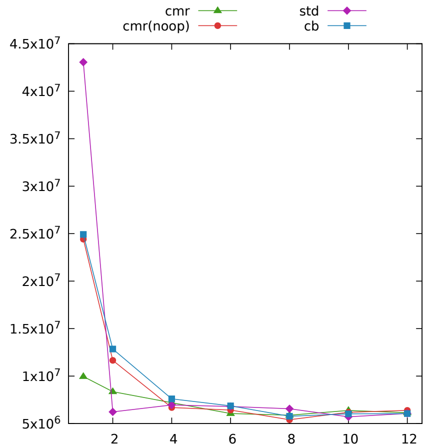
(b) `Queue::pop`



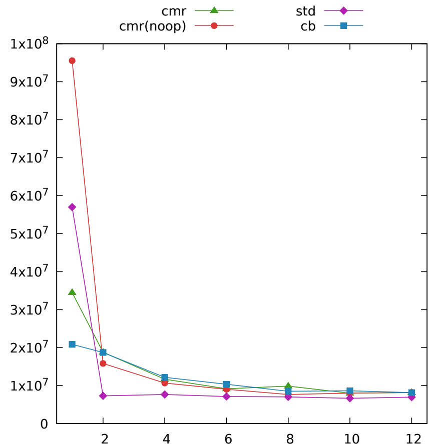
(c) `Queue` with 50% push, 50% pop

Figure 8.1: Queue performance on Gribb

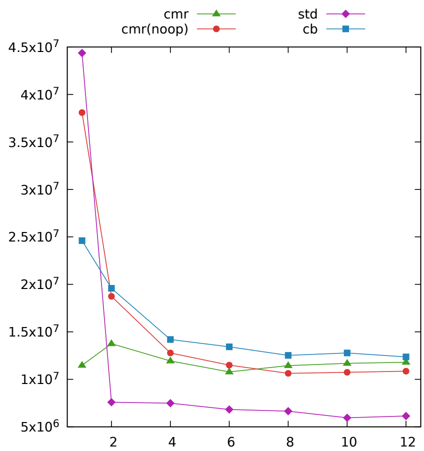
8.2.2 Queue



(a) Queue::push



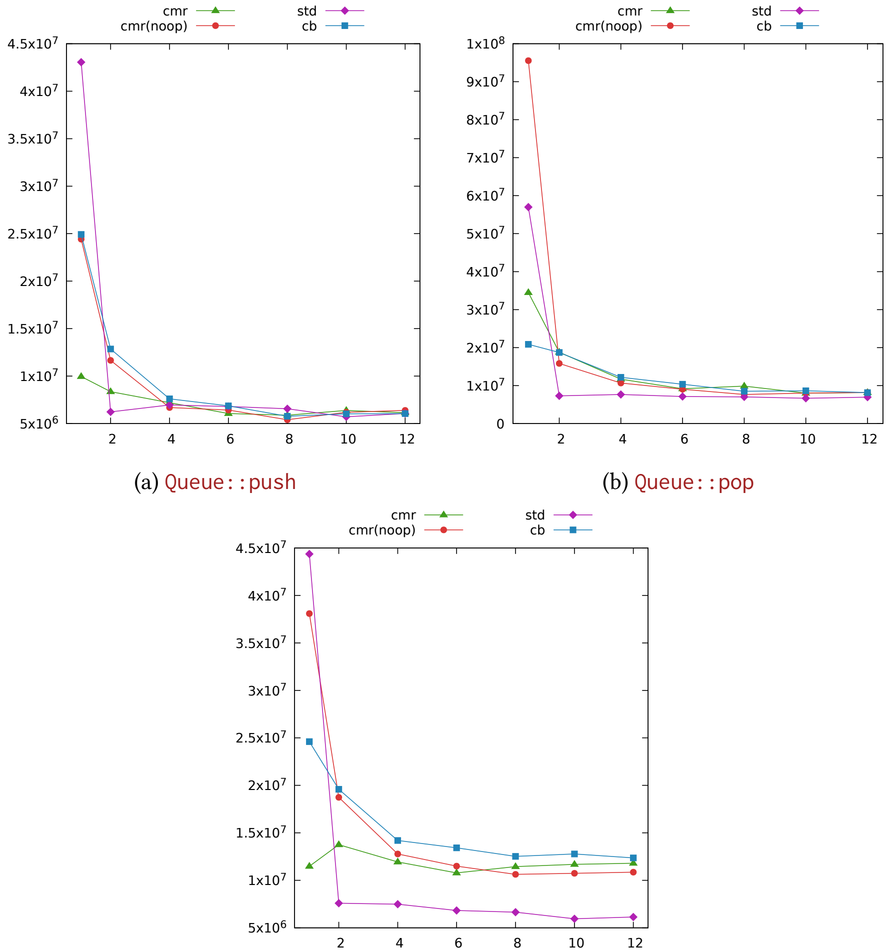
(b) Queue::pop



(c) Queue with 50% push, 50% pop

Figure 8.2: Queue performance on Gribb

8.2.3 List



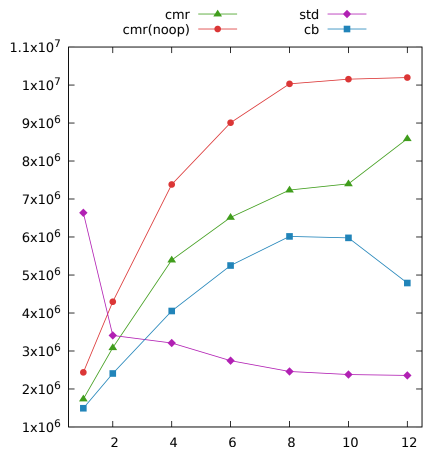
(a) `Queue::push`

(b) `Queue::pop`

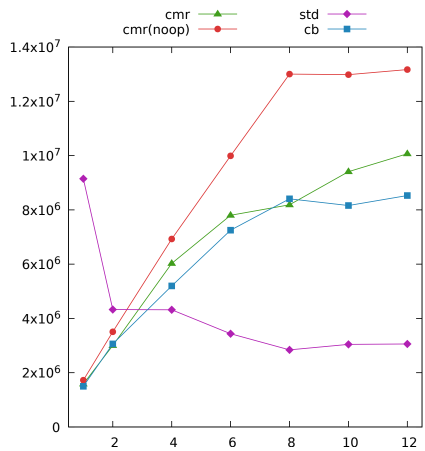
(c) `Queue` with 50% push, 50% pop

Figure 8.3: Queue performance on Gribb

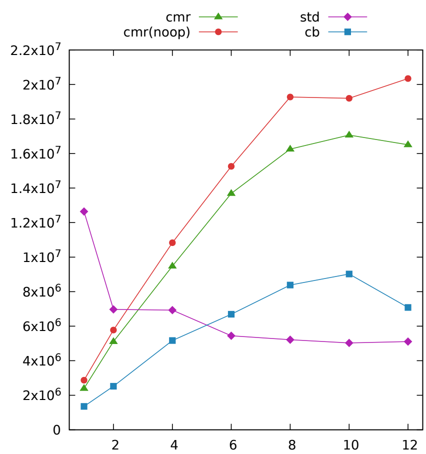
8.2.4 HashMap



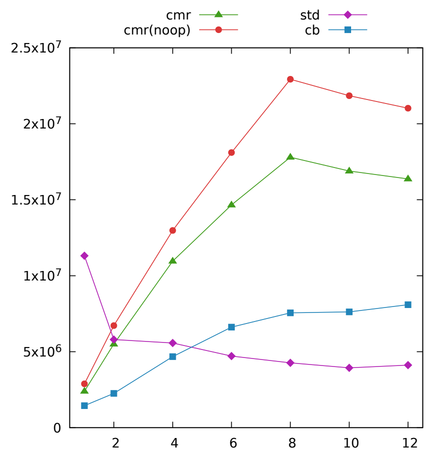
(a) `HashMap::insert`



(b) `HashMap::remove`



(c) `HashMap::contains`



(d) `HashMap` with 80% contains, 10% insert, 10% remove

Figure 8.4: HashMap performance on Gribb

8.3 Thirt Party Use

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 9

Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

9.1 Is CMR Useful?

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

9.2 Alternatives

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit

blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

9.3 Closing Words

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix A

Extended Results

Abbreviations

`pthreads` POSIX threads. 30, 33

NLL Non-Lexical Lifetimes. 14, 43

RAII Resource Allocation Is Initialization. 13, 32

RFC Request For Comments. 9

TLS Thread-Local Storage. 25, 33

Bibliography

[1] Rust programs vs C gcc. https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust.html. [Online; accessed 31-May-2018].

[2] The LLVM Compiler Infrastructure. https://llvm.org, 2017.

[3] More flexible coherence rules that permit overlap; Issue 1053. https://github.com/rust-lang/rfcs/issues/1053, 2018. [Online; accessed 4-Jun-2018].

[4] Brown, T. Good Data Structure Experiments are R.A.R.E. https://www.youtube.com/watch?v=x6HaBcRJHFY. [Online; accessed 15-May-2018].

[5] Buxton, J. N., and Randell, B. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee.* NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[6] CMR - Concurrent Memory Reclamation. https://github.com/IST-DASLab/rust-drop-box, 2018.

[7] Crossbeam. https://github.com/crossbeam-rs/, 2018.

[8] GitHub. http://github.com/.

[9] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing* (2001), Springer, pp. 300–314.

[10] Jemalloc. http://jemalloc.net/, 2018.

[11] Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM, pp. 73–82.

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

[13] POSIX.1-2017. http://pubs.opengroup.org/onlinepubs/9699919799/.

[14] The Rust Programming Language. http://rust-lang.org/.

[15] The Rust Programming Language on GitHub. http://github.com/rust-lang/rust. [Online; accessed 31-May-2018].

[16] Shalev, O., and Shavit, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM) 53*, 3 (2006), 379–405.

[17] The Rust Programming Language. https://doc.rust-lang.org/book/second-edition/.

[18] Treiber, R. K. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center New York, 1986.

[19] trench, the threaded benchmarking library. https://github.com/martinhath/trench, 2018.

[20] Valgrind. http://valgrind.org/, 2018. [Online; accessed 12-Nov-2017].