TDT4900: Masters Thesis

CMR: A concurrent memory management system for Rust

Martin Hafskjold Thoresen

supervised by
Magnus Lie Hetland

June 7, 2018

Abstract

Concurrent memory reclamation is the problem in deciding whether a memory allocation is still in use or not in a concurrent system. This thesis presents a new system called CMR for the Rust programming language and proves its correctness. We also show implementations of four concurrent data strctures using CMR. Experimental results show that CMR may be viable for certain workloads, although intrinsic properties of the system may prevent it from general adoption.

Sammendrag

Minnehåndtering i samtidige systemer er problemet å avgjøre om en minneallokasjon er i bruk i et system med samtidige utførelsestråder. Denne avhandlingen presenterer et nytt system kalt CMR for programmeringsspråket Rust, og bevisers dets korrekthet. Vi viser også implementasjoner av fire samtidige datastruktures som bruker CMR. Eksperimentelle resultater viser at CMR kan være forsvarlig for visse arbeidsmendger, til tross for at grunnleggende egenskaper ved systemet kan hindre generell adopsjon.

Preface

This thesis is presented in partial fultilment of the requirements for the degree Master of Science at the Norwegian University of Science and Technology in the spring of 2018. The topic of this thesis has been worked out in part from a semester project in the fall of 2017 and discussions with the members of the Distributed Algorithms and Systems research group by Dan Alitarh at IST Austria.

Acknowledgements

The writing of this thesis has been a long project, but it has also been a very rewarding one. I would like to thank my advisor, Magnus Lie Hetland, for helping me through the project, despite my lack of questions for him. I would also like to thank IST Austria, and Dan Alistarh in particular, for offering me an internship, and later inviting me back for two weeks, while I was planning and working on this thesis.

Martin Hafskjold Thoresen
Trondheim, June 2018

Contents

Resolve These Things

Chapter 1
Introduction

In this chapter we introduce the problem space in which this thesis operates. We look at the general problem that we want to solve, and why it is an interesting problem. We also draw an outline of the structure of this thesis, and summarizes each chapter in short.

1.1 History

The clock speed of computer processors have had a very fast increase in the last 50 years; Gordon Moore observed in 1965 that he number of components in an integrated circuit roughly had doubled every 18 months, and speculated that this trend would continue for the next 10 years. The law, which now is knows as Moores Law, still holds true today. Moores Law has also been used on the clock speed of CPUs. This trend, however, has grinded to a halt; the clock speed of desktop processors in the last 10 years have been more or less stagnant. Despite the lack of clock increase, there have been major improvements in modern CPUs. Improvements include pipelining, branch prediction, out-of-order execution, and, most important for this thesis, multiple processing cores.

Multi-core processors have been increasingly mainstream in the last 10 years; modern enthusiast desktop CPUs, like the recently announced AMD ThreadRipper 2, has 32 cores and support for 64 hardware threads. Even embedded systems, like smartphones, often come in variants with 4 or 8 threads. While multi-core systems may offer increase on computation speed, they also introduce new problems; utilization of parallel systems is not trivial. Many problems are inherently serial and serial solution to problems are often much easier to develop than efficient parallel systems. Efficient synchronization between processes is also, perhaps surprisingly, a difficult problem.

Many modern programming languages aim for developer productivity. Many programming abstractions and runtime subsystems are introduced in the name of convenience; one of these is a memory management system, often refered to as a garbage collector. Despite the developer ergonimic inprovements such systems claim to improve, they often come at a cost: efficiency. Garbage collectors have to support a wide range of use cases, like both short-lived allocations of small objects *and* bulk allocation of large objects, the overhead of a memory manager may be far from optimal.

Rust

Rust is a new programming language aiming to unify the developer ergonomics promised by managed languages, and the efficiency of unmamaged languages. The work of managing memory is left to the compiler, which at compile time analyses the program, and enforces certain rules about how memory is handled. This way the programmer does not need to manage memory in the traditional sense, although it does impose a cognitive overhead when developing programs, which larger than that of managed languages. One of the features of Rust that may make it viable for systems programmers with strong requirements for performance and stability is that the memory management system in Rust is fully controlled by the programmer, as there is no runtime.

1.2 This Thesis

In this thesis we aim to develop a concurrent memory management system for Rust called CMR, which is based on Forkscan 2.4.4. We believe this is an interesting topic since it is not clear how to incoorporate memory management for concurrent system in the ownership model that Rust provides, although there are existing projects which does similiar things; these projects are often implementation of other known memory reclamation schemes like Hazard Pointers and Epoch-Based reclamation. This thesis develops a new scheme.

Despite one of the promises that Rust makes is one of predictability and control, CMR works more like a traditional garbage collector. However, as the scheduling of threads in a process inherently is fuzzy and unpredictable, we believe that this is an intrinsic property of any concurrent memory reclamtion system.

The aims for this thesis is to consider CMR as an alternative to more traditional methods for memory reclamation. The viability of such a management system is both performance and usability, and we will comment both.

1.3 Outline

We mention briefly the contents of each chapter. The chapters are also prefaced with a more fine grained introduction.

Chapter 2 highlights the most important background material that is needed in order to appreciate the contents of this thesis. Much of the material is a part of a standard computer science curriculum, but we repeat it here nevertheless. Relevant sections from this chapter is cited in the remaining of the text.

Chapter 3 introduces the reader to the Rust programming language, in which the impementation of CMR is written. The way Rust functions makes it an integral part of how CMR is designed. While this could be considered background, itis its own chapter due to its importance for this project as a whole.

Chapter 4 presents the memory management system CMR at a high level. We look at primitives of CMR and common operations such as allocation and reclamation protection. The chapter presents CMR on a high level in order to clearly differentiate the challenges of CMR from the challenges of implementing CMR on a real system in a real programming language. There is no real code in this chapter.

Chapter 5 describes the implementation of CMR in Rust, including the most important primitives and procedures. With the exception of a few omitted code sections, which are clearly marked as such, all program code in this chapter is fully functionally Rust code taken from the impementation.

Chapter 6 contains example usage of CMR for external applications. We have implemented four data strucutres using CMR, and these four along with their implemetations are discussed further in this chapter.

Chapter 7 mentiones practical matters when it comes to testing and benchmarking of the system. This is included in the thesis as it may be quite different from testing and benchmarking of sequential systems, or systems written at a higher abstraction layer.

Chapter 8 discusses experimental results of both operations from CMR as well as the data structures from the previous chapter. We look at the performance on a range of different machines, from affordable desktop computers to high-end multiple socket server systems.

Chapter 9 concludes this thesis with a short summary of the results and discussion about the applicability and usability of the system as presented, in addition to suggestions for future work.

Chapter 2
Background

In this chapter we briefly sum up the most important backgroud material we depend on in this text. Parts of the material is covered in a standard computer science education, but we will summarize it nevertheless.

Sections 2.4.1 to 2.4.3 and parts of Section 2.5 are taken from [39].

2.1 Operating Systems

The operating system is one of the most crucial parts of a modern computer. The functionality the operating system provides is generally an abstraction layer over the hardware of the machine, but it also acts as a resource manager for resources such as memory and processing time. We summarize a few of the most important topics within operating systems that are relevant for this thesis.

2.1.1 Virtual Memory

One of the most important features of modern operating systems is to provide virtual memory. Instead of having programs use the memory of the system directly, the operating system acts as if each process have the entire adrress space for itself. Behind the scenes the operating systems maps the programs address space to addresses on the physical memory. Naturally, the memory addresses on the physical memory does not overlap for different programs.

The operating system handles memory in segments called *pages*. The page size is configurable, but is usually 4096kB in size. Memory addresses in the program space is mapped to an adrress in a specific page, and the page is again mapped to the address on the physical memory. A common optimization in modern operating systems is for the pages to have Copy-on-Write (CoW) semantics; this means that when a page is copied, it is only marked as copied; it is first when either of the two copies of the page is changes that the actual copy is performed. This is an important optimization since many page copies are never modified.

Memory Maps

Memory maps is another feature that operating systems may provide. Typically a program may memory map a file, which maps the contents of the file to the virtual address space of the process. Then the program can read and write to the memory directly, and have the operating system take care of mirroring the changes to the file, which resides on disk. The main motivation behind memory mapping a file is to abstract away the fact that the underlying data is not all in memory, but may be read and written to incrementally.

2.1.2 Threads and Processes

Operating systems run programs as *processses*. A process have a uniqe address space, in which only the process itself can operate[1]. While not so common in modern programming, a common pattern in handling processes is *forking*, where the process clones itself, and both copies, the child and parent process, continue their execution. This is an excelent application of the CoW semantics of page cloning as mentioned in Section 2.1.1, since the entire address space of the process is copied.

Each executing process may have multiple execution units called *threads*. The main difference between threads and processes is that a process have its own address space

[1] this is a truth with modifications; the kernel has naturally the privilege to touch all memory it pleases.

whereas a thread does not. This allows multiple threads to communicate and share data by simplty sharing the location of the data they want to share. Threads are also much lighter, meaning there is a smaller overhead in creation and switching execution of threads than that of processes.

The most common implementation of threads on Unix based systems is POSIX threads (pthreads). pthreads standarizes thread management, such as thread creation and joining, but also thread communication primitives such as mutexes, condition variables, and barriers.

2.1.3 Signals

Signals is a Inter-Process Communication (IPC) feature in POSIX operating systems used for asynchronous communication between processes. Most system programmers have encountered a few signals in their carreer, such as SIGSEGV, SIGINT, SIGKILL and SIGTERM. Signals are caught by the receiving process and a signal handler is executed. Certain signals, like SIGTERM cannot be caught, as the intent behind the signal is to abruptly terminate the process.

In addition to using signals for IPC, pthreads supports signals as well. The interface is similar to that of POSIX signals, but instead of sending signals to processes, they are send to threads within the sending process.

2.2 Programming Languages

Programming languages have been a hot topic since the birth of computing; we have always been interested in being able to express our intent for the computer clearly and effectively. Despite computing being a field of growing experience, new programming languages are still introduced by the dozen, and programming languages that are older than the average programmer are still in heavy use.

Many programming languages have a formal specification to specify the operational semantics of the language constructs, as well as guarantees of what subroutines and data types are generally available, in addition to details about said subroutines and data types. Having a specification of the programming language one uses can greatly improve security, predictability, and stability of language implementations, such as compilers or run-times.

Sometimes it may be wiser for language designers not to define parts of the language; many parts of any language is heavily introduced by the features or constraints of the hardware of the era that the language is defined in. For instance, as the C programming language was designed, having an own data type for floating point numbers float and double might not have been obvious, considering that few machines supported them. Being faced with the same choice today is quite different — no modern programming language ships without native support for floating point numbers.

There are usually gray zones of the definitions of parts of programming languages. For instance, in the C and C++ world it is common to differentiate between *implementation defined*, meaning an implementation is free to define the semantics, but it must be documented; *unspecified behaviour*, meaning an implementation needs not define its choice; and *undefined behaviour*, where all bets are off — a program containing Undefined Behaviour (UB) does not make sense.

Other reasons for not having a completely specified language is that compiler writers may make assumptions about the indented semantics of the code. For instance, we could say that if $a > 0$ then $x + a > x$ is always `true`. However, if the addition of x and a overflows, it may lead to a number that is *smaller* than x. Another example is that $a = 2x$ should imply $\frac{a}{2} = x$. Again, should the multiplication $2x$ overflow, this might not be true. If we abstain from defining the overflow of numbers, we can get away with making these assumtions; the program may end up producing non-sensible results, but from a language specification standpoint that is alright, as the program was invalid to begin with: its behaviour was undefined.

Good introductions to the tradeoffs regarding UB includes [33, 13].

2.2.1 Garbage Collectors

Another important distinction in programming langauges are *managed* and *unmanaged* languages. The former usually refers to languages in which memory management is abstracted away by having a runtime with a subsystem that manages memory allcation and reclamation automatically; such a system is often refered to as a *Garbage Collector*. Most of the mainstream programming languages today, including Python, Ruby, Java, Go, Javascript, and C#, are managed languages.

There are many variations of garbage collectors. The most common is the *tracing garbage collector*, where the system traces memory addresses through the application memory in order to find out what subset of the total memory are still in use. Depending on the programming language, the garbage collector may be optimized to handle specific allocation patterns. One common pattern is the fact that most allocated objects are only in use for a short time; thus it might make sense to handle new objects different from old objects, as the probability that a new object is already not in use may be quite high.

2.3 Concurrency

Modern hardware and operating systems makes heavy use of concurrency; processes are continously preempted in order to have more processes executing than processors available on the system. When the processes are running independently of each other this works rather seamlessly. However, the hardware deals with many quite difficult concurrency problems that programmers seldom think about: for instance *cache coherency*.

Due to the increasing gap between memory access speed and compute speed, modern CPUs employ a range of caching schemes. By moving a copy of the memory a process is accessing physically closer to the execution unit on the processor, the access time is greatly reduced. However, with multiple processors on a single system, this data duplication introduces problems when two processes are accessing the same data, as the hardware must realize that the local data that each process have may be changed by the other process, and hence invalidated. This synchronization can be, and very often is, very expensive compared to the usual work of the CPU.

Still worse, the memory location that the processes change does not need to be the same address, but just be in proximity of each other. This is because the cache of a processor does not operate on single words, but on whole segments called *cahce lines*. Even ajacent cache lines may cause unneded synchronziation if the lines are read and modified by processes that does not share all levels of cache. Having superfluous synchronization due to the locality of modifies data among processes is called *false sharing*.

Another problem that the slow memory access speed realizes is that for communicating processes on different processors, the order of operations may be of the uttermost significance. This strongly imposes requirements on the hardware forces the inter-processor communication to be of a certain nature. It turns out, however, that this also decreses the performance of the CPU significantly. Attempting to both have our cake and eat it too, CPU architectures define a *memory model*: rules about limitations on instruction reordering. Weak memory orderings, such as ARM and PowerPC, impose very few restrictions on the reorderings, so that programmers must write memory *fence* instructions to explicitly manage the ordering relationship in the code. Strong memory orderings, like x86, on the other hand, allows very few reorderings. We will not discuss memory models and orderings further in this text, but it is useful to keep it in mind.

Many operations require that multiple operations appear to happen as a single unit for all other processes. We call such operations *atomic*. A simple atomic operation is the `fetch_and_add`: given a memory location, it reads the location of a number, increments the number, and writes the new incremented number back to the original location. Simply reading, incrementing, and writing back using regular instructions will not work in a concurrent system: assume we have two threads T_1 and T_2, that both wants to count an event, sharing the counter. If the system only has one physical processor, T_1 might read a number n, increment it to $n + 1$, and then become preempted before writing the value back to the original memory location. Then T_2 gets execution time, and records successfully m events. Now, the next time T_1 is ran, it will write $n + 1$ to the location, which will effectively remove the m events that T_2 counted. One of the most important atomic instructions is the `compare_and_swap(l, a, b)`, which reads a location l, and writes b to it, if it read a.

2.3.1 Common Patterns in Concurrent Programming

Many programming languages supports higher level concurrency constructs for concurrent programming, such as threads pools and the message passing pattern. A thread pool acts as a thread manager; given work to do it will manage the execution of the work on threads. The user of a thread pool does not need to know how this management functions, only that the work is executed concurrently, such that it hopefully utilizes the parallel nature of modern processors. An often used idea in implementing a thread pool is *work stealing*[8], in which threads have their list of work avaiable to the other threads, which may "steal" a part of their work, should they run out themselves.

Message passing is a concurrent computational model [23], which has seen somewhat of a renessance with programming languages such as Go [3], and programming with co-routines, a popular pattern in Kotlin [4]. Message passing is often simpler than other means of communication, since the processes communicate in a clear manner, and can be programmed to act reactively.

In lock-free programming, the `compare_and_swap`, or `cas`, operation is heavily used. The general idea is to attempt to perform an operation, having the comparison check that nothing has changes in between reading the value that we perform the `cas` on. If the `cas` fails, that is the read value was different, we restart. Otften operations look like Lock-Free-Op.

```
Lock-Free-Op($l$)
while
    $m = $ Read(l)
    ⟨some operation yielding a result $x$⟩
    if cas($l, m, x$)
        return
```

2.3.2 The ABA-Problem

However, there are pitfalls to this approach. Occationally checking that the location l still has its value a might not be sufficient to see that the remaining of the program is in the state that one expects. This problem is called the *ABA-Problem*.

Consider the following real world analogy: assume you have an opaque bottle that is filled with water. If you leave the bottle on your desk and return to it after lucnh, there is no way to see whether anyone has been drinking your water by simply inspecting the bottle from the outside. Someone might have taken the bottle, drunk the water, and put the bottle back as it were. Even worse, someone might have replaced your bottle with an identical bottle filled with bees.

The ABA problem is often due to the fact that we may only look at a single word when performing the `cas`; had we been able to validate arbitrary memory this would not be a problem. Certain CPU architectures mitigate this problem by providing double `comare_and_swap` (`dcas`), which is two `cas` operations only executed if both suceed, or double-word `compare_and_swap` (`dwcas`), which checks, say, 128 bits instead of the word size of 64. It is possible to implement `cas` operations with arbitrary number of locations (`casn`)[22, 26], but the implementations are often not practical. Other alternatives include transactional memory, but this is not covered in this thesis.

2.4 Memory Reclamation

Most programs require blocks of memory which size is only known at runtime, but the operating system usually only deals with memory in pages. Memory allocation is a system designed to unify the two by managing large blocks of memory and handing out portions of it to the process. The subsystem managing this is called the allocator. Many general purpose allocators exist, such as [20, 24, 36], and general ideas can be found in [25].

The use of a general purpose allocator ifor all allocations is a program is often a source of performance problems; specialized allocators for short lived objects, small objects, large objects, or bulk allocated and reclaimed memory (an “arena”) are often used.

In concurrent systems, memory reclamation is much harder. The main source of problems seems to be that scheduling of threads is unpredictable, and it is hard to differentiate whether a thread is done using some memory or if it just has not used the memory in a long time, which again may be due to it being preempted.

We look at a few schemes for concurrent memory reclamation.

2.4.1 Reference Counting

Reference counting (RC) is a natural solution for memory reclamation. It was introduced in 1960 by G. E. Collins[16], where it was used for collecting nodes of a linked list. The idea is that we count the number of references to data, so that we can tell if we are holding the only reference to some data. When we no longer need this reference, we know it is safe to reclaim the memory the reference points to, since no other reference to that memory exists. The primary downsides of RC is that it is rather expensive, and that a naïve implementation does not reclaim cycles. Today reference counting is still used, although it is unusual to have it be the primary mechanism for memory management, due to its performance overhead.

Atoimc reference counting (ARC) is RC using atomic variables, and is a natural extension of RC. However, the naïve implementation is not correct: consider two threads operating on some `Rc<T>`. When thread A want to create a new reference to the data, it increments the count in the `RC` object. Upon destruction, the count is decremented and the data is freed if the count is 0. However, it is possible that thread B has a reference to the RC object and that it got preempted right before incrementing the count. Then the whole object gets freed by thread A, since the count is 0, and when thread B gets execution time again, it has a pointer to freed memory which it indents to read.

A way to mitigate this problem is by indirection: we can use intermediate `Rc` nodes which are the counter and a pointer to the actual data. The intermediate nodes are never free’d, and by `CAS`ing the count to a sentinel value upon destruction of the data, thread B can detect that it is about to read free’d memory and abort its operation. By allocation the `Rc` objects with a memory arena and freeing them in bulk, this might be acceptable for certain problems, as the data itself is not leaked, but only the `Rc` nodes, which may be comparably small.

Despite the problem of atomic reference counting, there are still use cases for it. A thread A may create an `Arc` object, and make a copy of its reference to it, incrementing the count, and only *then* pass it to another thread. This avoids the problem in the previous paragraphs, since the only threads that need to increment the reference count is already holding onto another reference, thus making it impossible that the count reaches zero before we get to increment it. When all threads have dropped their reference, the count will drop to zero, and the `Arc` will be free’d, not risking that any other thread is just about to increment its count.

2.4.2 Epoch Based Reclamation

Epoch Based Reclamation (EBR) was introduced by Fraser in [18]. It is a reclamation scheme based on the observation that most programs have no references to internal data structure memory in between of operations on the structure. The time interval in between operations on the data structure are therefore safe-points (also called grace periods) for memory reclamation to occur, since we do not risk invalidating any data that other threads are using in this period. EBR uses the concept of an *epoch*, a global timestamp which we use to find out when it is safe to reclaim retired memory. The epoch is a global counter. In addition we have a global list with one entry for each running thread, which the threads use for broadcasting their state, which includes the last epoch they read as well as whether they are currently performing an operation. We call a thread performing an operation *pinned*, and the action of marking and unmarking *pinning* and *unpinning* the thread.

When starting an operation a thread reads the global epoch, stores it in its entry, and pins the thread. Upon retiring memory the thread marks the memory with the global epoch and puts it in a *limbo list*. Every once in a while, the threads try to increment the epoch, which succeeds if all current pinned threads have seen the current epoch. Note that we only have to look through the thread entries once: if another thread is pinned while we are searching, it will read the current epoch, and cause no problems for us. The requirement for epoch incrementation means that all threads that have references to memory we might want to free is either in the current or the previous epoch. Thus, after incrementing an epoch to e we know that garbage that was added in epoch $e - 2$ is safe to be freed.

Note that it is important that the thread inserting into the limbo list uses the global epoch, and not the epoch it read when it was pinned. If we use the previously read epoch, we may run into the following scenario:

1. A pins the thread at $e = 5$, and wants to remove O from the data structure.

2. B increments the epoch to $e = 6$, and obtains a reference to O.

3. A unlinks O from the data structure, and adds it to the limbo list, with $e = 5$. A unpinns, and increments the epoch to $e = 7$.

4. It is now safe, by our rules, to free O, although B is still holding a reference to it.

By reading the global epoch before pushing to the list we avoid this problem, since O is unlinked from the data structure before reading the epoch. This makes it impossible for B to have incremented the epoch, and *then* get a reference to O, without A reading the incremented epoch.

EBR is very popular, due to its extremely low overhead. However, there are still a few challenges with EBR. A problem is that we are not allowed to keep references to data across operations, since the thread must be pinned while we are using the references. A natural way to mitigate this constraint is to leave the thread pinned. However, this will stop the advancement of the global epoch, and thus effectively halting the memory reclamation. An immediate consequence of this is that EBR is not lock-free, which is not acceptable for all use cases.

2.4.3 Hazard Pointers

Hazard pointers were introduced by Michael in [28]. The paper formalizes hazardous pointers, and includes a proof of correctness. We will settle for a informal view of them. It is based on the observation that in most operations on data structure we only need a small constant number of references to memory that is shared between running threads. The technique exploits this by allowing each thread to register the pointers, called *hazard pointers*, the thread wants to use, but which it cannot be sure are safe (meaning invalid og prone to ABA errors). We call potentially unsafe pointers *hazardous*. The number of pointers we need varies with the algorithm performed, but a typical value is one or two.

After reading a hazardous pointer the thread registers it as one of its hazard pointers. It then have to *validate* that the pointer is still in the data structure, as it might have been removed in between the initial read and the hazard registration. When we want to free memory we look through the hazard pointers of all running threads. If no other thread has registered the memory it will be safe to free, since the object is already unlinked from the data structure, and is hence no longer reachable. If a thread has read the pointer before it became unlinked but not yet registered it, it will fail validation. We note that again, as with EBR, a single pass through this list is sufficient: the object is unlinked before searching, so if a thread has a reference to it, but is yet to register it as one of its hazard pointers, then it will fail validation.

If the memory is registered in a thread, we cannot immediately free the memory. We now have two options: wait for the thread to finish, or defer the deallocation. By waiting on the thread, we are relying on that the other thread will ever deregister the pointer. Hence, we give up lock-freedom, as this is prone to deadlocking. It has, however, very low overhead, and will be very fast assuming all threads are fairly scheduled and have similar work load. Deferring the deallocation is a safer option, although it have a higher overhead, since wee need to push the pointer into a queue (or a similar scheme). We would then occasionally visit the queue and see if any of the pointers in it have been deregistered by all threads.

A challenge in the usage of HP is that we need to identify which pointers in our algorithms are hazardous. In comparison, we have no such concerns in EBR, in which we only need to register memory as garbage when we remove it from the data structure (we do need to make sure that this memory is only registered by a single thread). Another challenge is that of validation, as there is no general way to do this. For most structures there is an obvious way of doing this. For instance, in a queue we can validate the front element by reading the `head` pointer again, observing that it has not changed. However, it still requires local knowledge of the data structure in question.

2.4.4 Forkscan

Forkscan is a recent addition to the family of memory reclamation schemes, and was introduced in [7]. The high level idea is to have a thread `fork()` off a new process, and scan the stacks of all threads in the system, looking for pointers. This way the system can find all addresses that are reachable, and thus also addresses that are no longer reachable. It also employs signals (Section 2.1.3) to have threads do some work before the process forks. Since the signal handler is a new procedure signaling the threads ensures that they push out their registeres to the stack.

2.5 Related Works

Automatic memory management exist for languages that does not provide this luxury themselves; the most known example is the BDW-GC [9]. There is also ongoing work for standarizing such systems in some languages; A proposal for including hazard pointers and RCU (read-copy-update, not covered in this report) into the C++ standard library was released in November 2017 [32]. There is also ongoing work in managed languages, despite the presence of a garbage collector. An example is Project Snowflake [30] which combines ideas from both EBR and HP to get more efficient reclamation for concurrent systems on the.NET platform.

2.5.1 Crossbeam

In the Rust ecosystem the most notable contribution to the space of concurrency is the Crossbeam umbrealla project[17]. Crossbeam aims to offer concurrent data structures and primitives, in addition to memory reclamation systems. As of June 2018 a system using EBR is available, and ideas for a system using HP are voiced.

Chapter 3
Rust

> Rust seems sensible
>
> *John Carmack on Twitter*

> If you rest, you Rust.
>
> *Helen Hayes*

Rust [34] is a new programming language focusing on safety, performance, and concurrency. 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[19] by over 2000 contributors as of May 2018[35]. 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 [38].

3.1 Introduction

Rust is a compiled language with a minimal runtime, similar to C and C++. `rustc` uses LLVM[6] 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).

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

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

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 foos 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. structs 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!

Rust is marketed as a safe programming language; it is however important to realize that this is only a half-truth. In principle Rust, due to the `unsafe` keyword, is no more safe than any codebase in C or C++ is, and third party librarlies might hide the fact that

[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`.

they utilize unsafe code in order to appear more “safe”. The language offers many ways to avoid having to type the dreaded six letters and enter the world where all bets are off, but nobody is stopping crate authors, co-workers, or even yourself, to write unsafe code.

3.5 Concurrency

One of the main focuses of Rust is concurrency, and the language does offer a helping hand in writing concurrent code; many of these arises naturally from the type system, and the ownership model, like the single owner principle, and the single mutable reference rule.

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 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 Atomics. 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.

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.

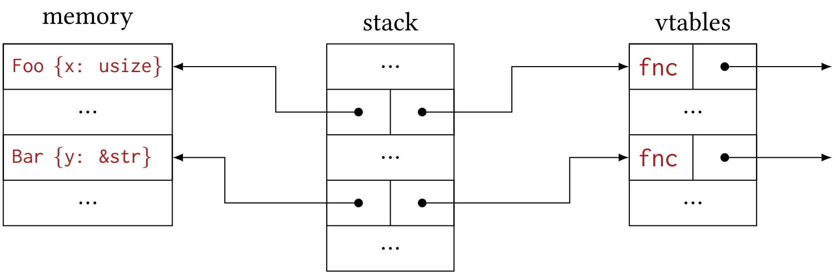


Figure 3.2: Illustration of memory when using Trait Objects.

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` 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> {
```

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

```
  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 specializations.

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[24]. 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

Garbage removal is a citizen responsibility.

Jaime Lerner

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, and thus gives the same guarantees as we give here.

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.

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

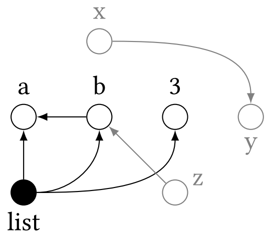


Figure 4.1: Code sample (left) with possible heap layout (right). If the black filled node is the only root, the black nodes are reachable, and the grey nodes are not.

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.

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 2.2). 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`[31].

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.

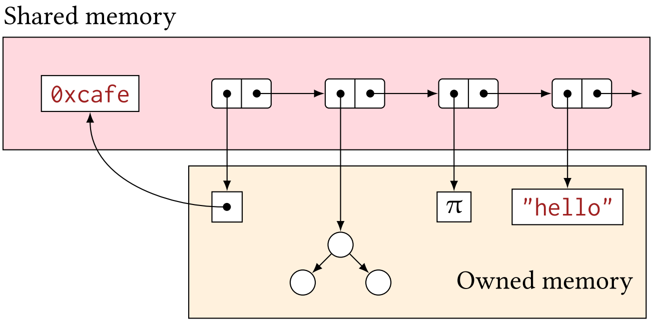


Figure 4.2: Example of memory layout showing owned memory (beige) and shared memory (red). Types in shared memory may contain pointers to owned memory, and vice versa.

We divide up A into two disjoint parts $O \cup S$: *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.

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.

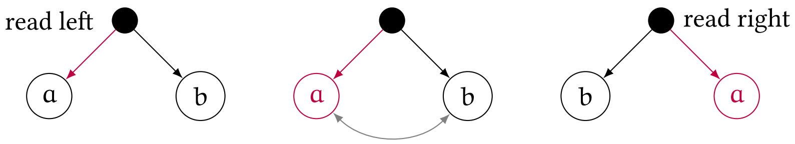


Figure 4.3: Mutation in the memory graph may lead to reachable blocks being observed as unreachable.

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 $\perp$. The *Guard* is non-movable in memory. All roots are stored in *Guards*.

The *Guard* is the only type that CMR defines that are different from conventinal memory management systems; they are solely used for managing the storage of the roots in the memory graph.

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.

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

4.3.1 Operations

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

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

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

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

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

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

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*.

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

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

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:

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

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:

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

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.

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: $g = p \neq \bot$. □

Theorem 4.10 (*Guard* is valid)**.** *If a Guard is not* $\bot$*, 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 $\bot$. □

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

Talk is cheap. Show me the code.

Linus Torvalds

In this chapter we look at the Rust implementation of CMR. The source code is openly available on GitHub under some license[15].

license

This chapter is organized as follows: Section 5.1 discusses briefly the most important data that CMR defines, both global and thread local; Section 5.2 shows the implementation of the primitives from Section 4.3, and argues for their correctness by the definitions in the previous chapter; Section 5.3 explains how memory snapshotting, an important part of CMR, is implemented; Section 5.4 describes the reacability analysis including important details of the Rust type system; we finish the chapter with Section 5.6 where we highlight a few of the complications that we encoutered during the implementation of CMR.

5.1 Data

In order to better understand how CMR is laid out, we start out by looking at the data. As Fred Brooks[10] said:

> *Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.*

Allocated addresses are stored in a global `HashSet`, `ALLOCS`, which uses a `Mutex` for thread synchronisation. Only addresses in `ALLOCS` are subject for reclamation. Section 5.4.1 explains more about the way allocations are stored in order to preserve type information.

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. In addition all threads store the allocations they have done since the last reclamation pass; these allocations are "stolen" by the reclaiming thread in each pass. See Section 5.6.2 for the details.

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. We summarize some of the problems in Section 5.6.3, and note that this is still a pain point of the implementation.

Only one thread may be in a reclamation pass at any given time, and we limit this by having a global `reclaim_lock`. A thread wanting to reclaim grabs the lock before doing anything else in the reclamation procedure; this lock is later freed (see Section 5.5). If a thread attempts to do a reclamation pass but finds that the lock is taken, it simply does not do the pass; waiting for the lock to be released would only increase the latency of the reclamation pass for that thread, and since a reclamation pass was recently performed, chances are that there will be very few new allocations to free in the pass.

5.2 Primitives

In this section we present implementation of the primitives as presented in Section 4.3, and show that the implementation is unifiable with the definitions of Chapter 4. All `struct`s are shown with their full definitions, but we show only highlights of the methods of the `struct`s, as most of them are trivial.

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.

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(); } }
```

Guard::register gets a mutable reference to the thread local Vec of Guards, 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 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 Atomics 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 Ts 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 2.1.3).

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 `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 Listing 5.6. 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.

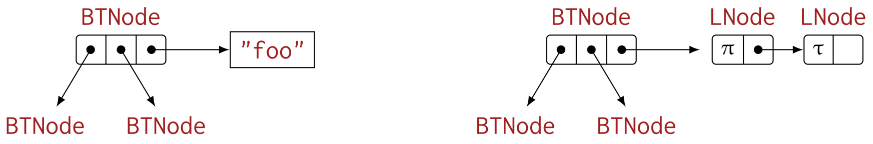
Listing 5.8: Rust implementation of Find-Reachable

```
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.



(a) Illustration of a `BTNode<String>`

(b) Illustration of a `BTNode<List>`

Figure 5.1: Generics influence the number of potential roots a type has, as shown here with a binary tree node `BTNode<T>`. The `String` does not contain a root, but a node in a linked list `LNode` does.

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 `TraitObject`s, 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 [5] 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 Communication

When `fork`ing the child process continues the thread of the parent process that called `fork()`, such that it has access to everything that the original thread had. As such, we don't need to communicate from parent to child. However, the job of the child process is to run reachability analysis, and we do need its result. CMR uses *memory map* (Section 2.1.1) for IPC, which is set up before the fork. Since we need to know when the child is done writing its results, we write a marker word as the first word in the memory map. Then we `fork()`. The child process does the reachability analysis, and writes the result after marker in the memory map. When it is done writing, it overwrites the marker with the number of elements written.

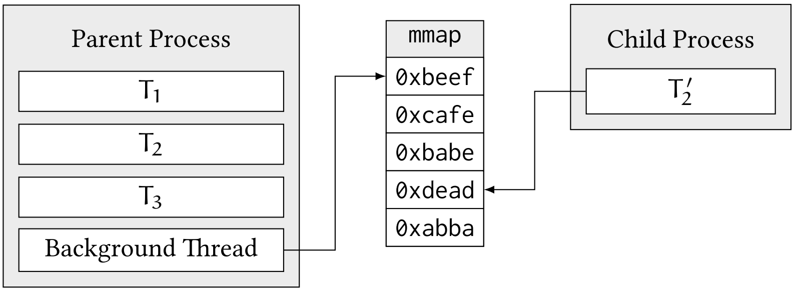


Figure 5.2: Illustration of IPC through a memory map. T_2 in the parent process is the reclaiming thread, so T_2 is the one thread in the child process. Both processes have access to the same memory in the memory map.

In order to minimize the delay of the reclamation pass from the point of view of the user, we spawn a background thread which handles the part of the reclamation performed after we `fork()`. This ensures that no user thread has to wait for the child process to finish its reachability analysis before going back to running application code. After forking, the parent process unfreezes the other threads, and sends all necessary data to the background thread. Note that the `reclaim_lock` is not released, even though the thread is exiting the procedure. The lock is only released when the background thread is fully done with the reclamation pass.

5.6 Complications

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

5.6.1 Allocation Lock

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. In order to be more thread friendly we do not grab the global lock every time we allocate; instead each thread flips a bit when they allocate, to signal to other threads that they might be holding some internal lock. Upon exit they flip the bit back. When some other thread wants to make sure that no thread is holding the lock, it flips a flag, and waits for all threads to have their bit set to zero. This is effectively a read-write lock.

The locking scheme 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.

5.6.2 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.

5.6.3 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
Usage of CMR

In this chapter we look at usage code for CMR. The goal of this chapter is twofold: we mainly 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.

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

6.1 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 [40].

The definitions of the Stack and Node structs and the two most important operations on a stack, push and pop, is shown in Listing 6.1. 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.

6.1.1 Push

push allocates the stack node itself, so it takes the value we want to push onto the stack (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.

6.1.2 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

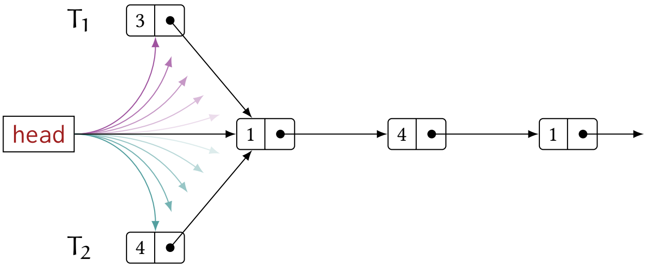


Figure 6.1: T_1 and T_2 both tries to swap the head pointer towards their node.

Listing 6.1: 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()); } } } }
```

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.

6.2 Lock-free Queue

The queue implemented is based on the well known Michael-Scott Queue from [29]. 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. 6.2.

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 6.2; pop is omitted due to its similarity with Stack::pop.

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

Listing 6.2: 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; } } } }
```

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 cased it to the last node (MS8).

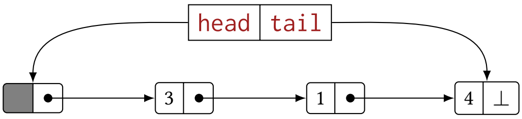


Figure 6.2: The Michael-Scott Queue. The first node in the queue is a sentinel node.

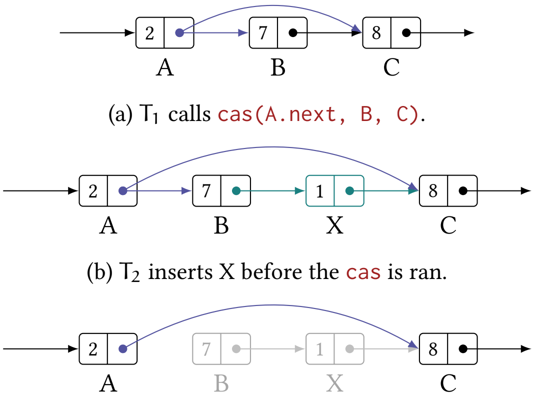
6.3 Lock-free List

Michael introduced a concurrent List in [21], which this implementation is based upon. The list is similar to the Stack from Section 6.1, 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. 6.3. 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 2.3.2).

A solution to this problem, as shown in eg. [27], is to exploit memory alignment on modern CPU architectures: structs 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

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



(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 6.3: Double removal with List::remove.

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.

6.3.1 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 since due to the pointer juggeling and the lifetime issues that arises. For this reason, the API uses an abstraction for iterating through the list: Entry.

Listing 6.3: 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>>,
    next: &'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_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 references to Guards rather than the Guards themselves. This makes construting a Guard nearly free. Another implication of this is that Entry is movable in memory (as Guard is not), which can be practical.

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, since we still need to handle the pointer management inside the Entrys methods.

Listing 6.4: 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; } } }
```

6.4 Lock-free Hash Table

The hash table is a versatile and popular data structure, and is widely used due to its fast operations, which includes queries, insertions, and removals. The hash table is also a more interesting data structure to look at from a concurrency perspective, as it has the potential to scale much better, since not all operations is on the same memory; in comparison a queue only has two locations, namely the front and back, which are of interest. Multiple inserts may be done concurrently in a hash table without touching the same memory at all; this greatly increases the gain from having multiple threads.

Lock-free hash tables are interesting for the same reasons as regular hash tables. Despite the interest, designing a concurrent hash table turns out to be a difficult problem. Usually the problems revolve around *resizing* the hash table. A common approach to implementing hash tables is to have an array of *buckets* in which the elements resides. Keys which hashes share some property (egȧ common prefix or suffix) may be put in the same bucket. When the *load factor* — the ratio of elements in the table to the number of buckets — is too large, we increase the number of buckets: we resize the hash table. The hash table implemented does not deal with this problem: we have a fixed number of "buckets", although it is possible to extend the implementation to better handle large load factors by recursively constructing a new sentinel array to handle the next n bits of the hash. The size of the array in the implemented hash table is $2^{20} = 1048576$.

6.4.1 Split-Ordered List

We start by describing the *Split-Ordered List*, which were introduced in [37]. The list is a regular linked list where the nodes are ordered by the *reverse hash* of the value in the node. The list also contains *sentinel nodes*, nodes without meaningful data, but which marks the beginnings of a bucket 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; this effectively differentiates between *one* new bit of the reverse hash. See Fig. 6.4 for an illustration of the numbers [0, 15] ordered by their bit-reverse. Note that when adding numbers up to the next power of two no two inserted numbers are consecutive; they spread nicely.

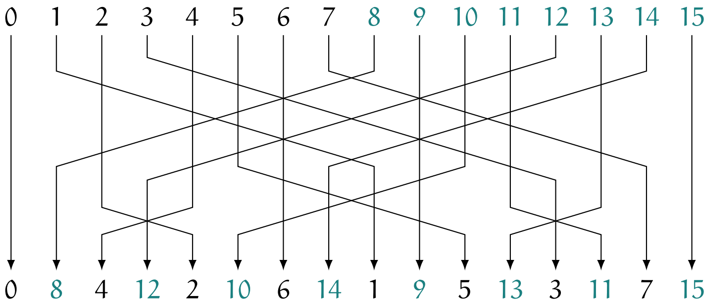


Figure 6.4: The numbers [0, 15] ordered by their bit-reversal

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 jump to the preceding sentinel node, and iterate through the list, finding the place in which our new node should be. Assuming a small load factor, this is a fast operation.

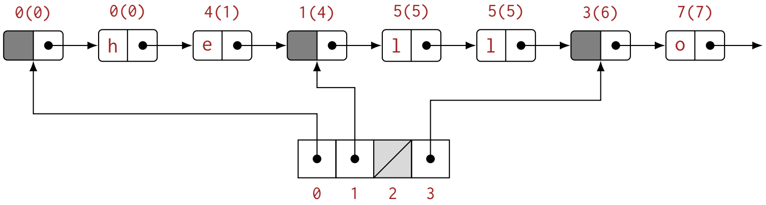


Figure 6.5: The Split-Ordered List. Node labels shows the `hash` and its reverse in parenthesis.

Fig. 6.5 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. It is only in the iteration where we insert our new node that we use the reverse hash. Inserting a node where `hash(n) == 10`, we would get `bucket == 2`, which is `null`, so we need to insert the sentinel node first. This sentinel node would be inserted in between the `h` and `e` node, since sentinel nodes preceed data nodes with the same reverse hash.

Next we look at the most important operations in the hash table: contains and insert. Removals are similar to in the List; we remove the data node from the list. Sentinel nodes are never removed. While this increases the memory usage of hash tables, it does not reduce the performance of lookups since the number of sentinel nodes we need to look at does not change.

6.4.2 Contains

Listing 6.5 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 6.5: 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() } }
```

6.4.3 Insert

HashMap::insert is more complicated, as there are multiple things that can go wrong, and that some operations require cleanup. Listing 6.6 shows the implementation of insert. Due to the complexity of the method, we have omitted certain sections of the code. The omitted code is either similar operations to previously shown methods, or explained in the text.

Listing 6.6: 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));
```

```
 HI8          'restart: loop {
 HI9              let mut entry = list::Entry::from_node_ptr(curr, _curr, _prev);
HI10              let res = entry.seek_with(|e| match e {
HI11                  &Entry::Value((h, ref key, _)) => h >= rev_hash,
HI12                  &Entry::Sentinel(h) => h > rev_hash });
HI13              if let Err(list::Error::Empty) = res {
HI14                  ⟨End of list case⟩ }
HI15              if res.is_err() { continue 'restart; }
HI16              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 (HI13), 2) we fail with another failure case and restart (HI15), and 3) we succeed and actually insert our new node into the list (HI16), 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 (HI17). 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.

6.4.4 Remove

HashMap::remove is, in comparison to insert, simpler. Finding the sentinel is similar to insert (HR3-HR7), but in (HR8) we use seek_with_opt which allows for early termination of the search, since we should stop if we reach a node with a larger hash than the one we want to remove. We test the result of the search (HR14) and branch appropriately, and call Entry::delete (HR18) if we found the node we are looking for. Only if we suceed in removing the node we break the loop.

Listing 6.7: Implementation of HashMap::remove.

```
 HR1  impl<K, V> HashMap<K, V> {
 HR2      pub fn remove(&self, k: &K) -> bool {
 HR3          let (bucket, rev_hash) = self.bucket_and_revhash(k);
 HR4          let curr: cmr::Ptr<_> = self.get_or_insert_bucket(bucket);
 HR5          guards!(_curr, _prev, _en);
 HR6          loop {
 HR7              let mut entry = list::Entry::from_node_ptr(curr, _curr, _prev, _en);
 HR8              let ret = entry.seek_with_opt(|data|
 HR9                if data.hash() > rev_hash { None }
HR10                else { match data {
HR11                          &Entry::Value((h, ref key, _)) if h == rev_hash &&
HR12                                                            k == key => Some(true),
                         _ => Some(false), } });
            match ret {
                Err(list::Error::Empty) => return false,
                Err(_) => continue,
                Ok(_) => {} }
            if entry.delete().is_ok() { break; } }
        self.count.fetch_sub(1, Ordering::SeqCst);
        return true; } }
```

An imporotant detail about `Entry::remove` is that the actual `cas` to remove the node from the list does not need to be successful for the operation to be considered as such; it is sufficient for the node to be tagged as removed. This is due to the fact that if the node is tagged as removed, any thread `step`ping over it will remove it.

Chapter 7
Methodology

This chapter summarizes some of the practical matters surrounding the implementation part of the thesis. This is neither a description of, nor a part of the implementation, of CMR. Despite this we believe documenthing the methods of experimentation is of equal importance in Computer Science as in any other science.

We look at testing in Section 7.1, as testing of concurrent systems are of a different nature than that of sequential programs. Section 7.2 shows how we have performed the benchmarks, which results are presented in Chapter 8.

7.1 Testing

Testing is an important part of software 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, while testing is useful to improve the quality of software, it is far from sufficient. As Edsger Dijkstra famously said[12]:

> *Testing shows the presence, not the absence of bugs*

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[42] and our own sanitizer (Section 7.1.1). Tests were also ran with and without compiler optimizations, as these optimizations often reveal yet more bugs.

7.1.1 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 the snippets below:

```
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) } }

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 it freed? {}", self.data, was_freed); } }
        unsafe { &*(self.as_raw()) } } }
```

This feature was of great help during development and testing, and it quickly reported illegal memory accesses, without having the overhead of other sanitizers like Valgrinds `memcheck`. The downside of this sanitizer is that it did not show the source of the errors,

[1] Some would call interleavings that reveal bugs fortunate

[2] *features* are similiar to `#ifdef`s in C and C++

only their precense. However, with this in mind we could rerun the program using memcheck and try to force the illegal accesses to manifest themselves, as we then knew that they were present.

7.2 Benchmarking

The benchmarks are ran with timed trials, where a function is ran repeatedly for a specified duration with any number of threads. The number of executions is counted for each thread and the total operations per second is summed and 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. Threads also time their execution time to catch skewage in the executed wall time for each thread, due to unfortunate scheduling.

There are a number of pitfalls when it comes to benchmarking code. We discuss a few of them; [11] 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 NUMA systems with multiple CPU sockets. The effect of having a single thread initializing all data is also dependent on the allocator used.

Having a constant overhead, and assuming that all workloads are equally hit by the overhead of the performance profiling system may also lead to errors; smaller workloads will naturally be more affected, and percentage wise changes to the reported data may get biased.

7.2.1 Trench

In order to more effectively benchmark threaded applications in Rust, an open source benchmarking library called trench[41] 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.

Listing 7.1: 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)); }
```

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

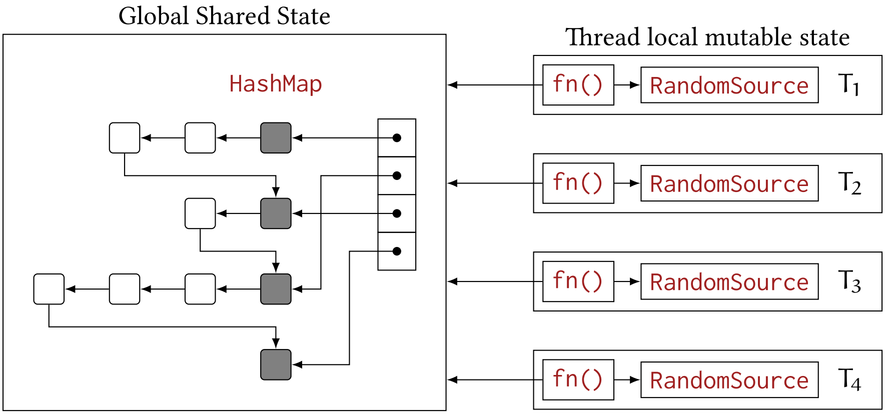


Figure 7.1: Illustration of the HashMap benchmark. The four threads all have their own RandomSource which supplies thread local random numbers, and they share the HashMap.

(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 7.1 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.

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 8
Results

> I pass with relief from the tossing sea of Cause and Theory to the firm ground of Result and Fact.
>
> *Winston S. Churchill, The Story of the Malakand Field Force*

In this chapter we look at the expermental results of the system, both piecewise and as a whole. We start by summarizing the hardware that the benchmarks is ran on in Section 8.1. In Section 8.2 we look at the overhead of the operations done by CMR. In Section 8.3 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[17] and data structures in the standard library wrapped in a `Mutex`.

8.1 Hardware

We start this chapter by looking at the hardware the benchmark suite is ran on. The benchmark suite is ran on four separate machines: one desktop machine Intel® i7−4770, an ARM based cloud server Cavium ThunderX and two quad-socket high-end server CPUs, Intel® Xeon® E7−8870 and Intel® Xeon® Gold 6150. The four CPUs all have different indented usage and price range, with the exception of the two high-end servers. It is therefore interesting to look at the performance on all systems, as oposed to limiting ourselfes to a certain price range or intended usage.

Intel® i7−4770 is an x86 desktop CPU running at 3.40GHz, with 4 cores and 8 threads. The system has 16 GB of RAM, and is the system with the least abount of free memory.

Cavium ThunderX is an ARMv8 CPU at 2.50GHz, having 32 cores and 32 threads over two NUMA cores, with 32 GB of RAM.

Intel® Xeon® E7−8870 is an x86 CPU for the server market running 10 cores with 20 threads at 2.40GHz. The system we ran tests on was a quad socket system, making the total thread count 80. The system had 1 TB of main memory.

Intel® Xeon® Gold 6150, the last server, is similar to the Intel® Xeon® E7−8870 except that it has a slightly fast clock rate, 2.70GHz, and it has 18 cores with a total of 36 cores. This was also a quad-core system, making the total thread count a whopping 144. The system used had 512 GB of main memory.

The number of threads for the benchmark ranges from 1 to slightly above the number of hardware threads on the CPU the benchmark is ran on. It is expected that the performance evens out when the number of threads reaches the maximum number of hardware threads in all benchmarks. The duration of the benchmarks also varies, due to memory constraints of the system they are ran on.

8.2 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.

8.2.1 Primitives

We begin by looking at the performance of `Guard` construction and allocation. The generated code from the `guard!` macro contains some initialization checks, which the compiler could not remove despite constructing multiple guards in a row. For this reason the `guards!` macro were written, which reduced the execution time by 20% for 10 declarations. We measure the time one `Guard` declaration takes, and the time for 10 `Guard`s to be declared using the `guards!` macro. All measurements are amortized over 1000 runs as shown in Fig. 8.1, but the reported numbers are per operation.

The results for all machines are summarized in Table 8.1. Note that a single `guard!` is faster than `guards!` per declaration. This can be attributed to that destruction of a

```
#[bench]
fn cmr_guard_1k(b: &mut Bencher) {
    global_init();
    let _t = ::test::test_init();
    b.iter(|| for _ in 0..1000 { guard!(g);
                                 let _: &mut Guard<u64> = g; }); }
```

Figure 8.1: Benchmark for `Guard` construction.

`Guard` must find itself in the `Vec` of `Guard`s, so more `Guard`s take longer.

Table 8.1: Summary of the execution of selected CMR operations. All numbers are per single operation.

Machine	`guard!`	`guards!`	`cmr::alloc`	`Box::new`
Cavium ThunderX	78 ns	92 ns	335 ns	185 ns
Intel® i7−4770	13 ns	13 ns	45 ns	28 ns
Intel® Xeon® E7−8870	28 ns	30 ns	73 ns	50 ns
Intel® Xeon® Gold 6150	12 ns	18 ns	56 ns	33 ns

8.3 Data Structures

As mentioned in Chapter 6, the stack and the queue both have operational bottlenecks; that is most operations contest the same memory locations, which causes poor scaling with more cores. In addition, since the list from Section 6.3 is the primary building block for the hash table, we do not look at the performance of the list explicitly. Thus, the only remaining data structure to look at is the hash table. This is also the most interesting.

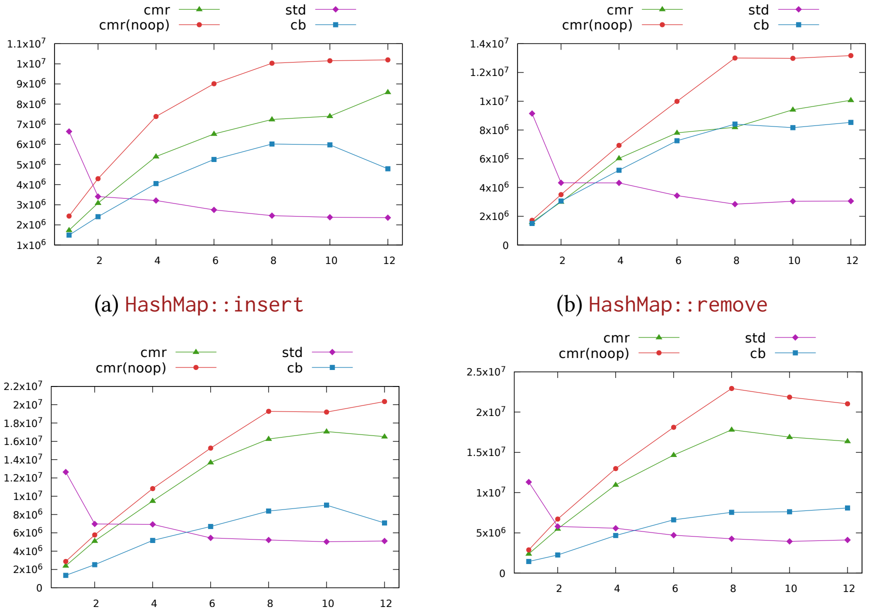
We compare the four hashmap variations: 1) the hashmap from Section 6.4 (`cmr`) 2) the same hashmap, but with all operations of CMR to be no-ops (`cmr(noop)`) 3) an external SkipList implementation from the Crossbeam project[2] (`cb`) and 4) `std::HashMap` wrapped in a `Mutex` (`std`).

The `HashMap` benchmarks consists of four operations: `insert`, `remove`, `contains`, and a combination of the three: a 80/10/10 split of `contains`, `inserts`, and `remove` respectively. This is shown experimentally to mirror real world usage of hashmaps quite well, and is common in concurrent performance testing.

cite !!

Naturally, this is not quite an apples-to-apples comparison; the hashmap of CMR is implemented quite differently than in Crossbeam, and even more different than the one in the standard library. Therefore we can, and should, attribute parts of any experimental difference to the difference in implementation.

8.3.1 Intel® i7–4770



(a) HashMap::insert

(b) HashMap::remove

(c) HashMap::contains

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

Figure 8.2: HashMap performance on Intel® i7–4770

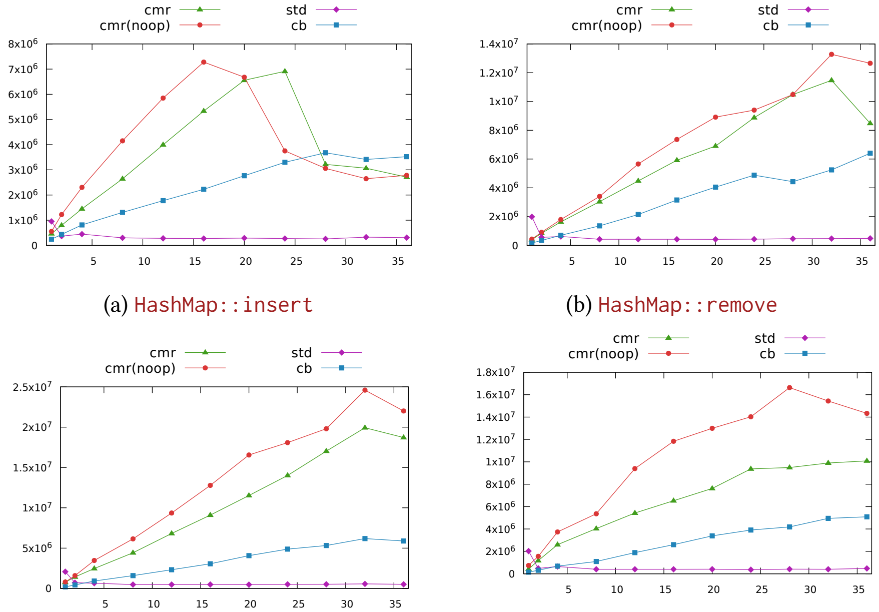
The experimental results reveal that the hash table scales properly up to the thread limit of the machine. As mentioned in Section 8.1 this is expected. We also see that the variant of CMR with all overhead removed performs strictly better than the real CMR; this acts as a fine sanity check. Crossbeams SkipList also scales well, although its performance is slightly lower than that of CMR, with the expection of the very last data points from the remove benchmark.

It is also nice to see that the naïve approach of wrapping a HashMap in a Mutex scales rather poorly; however we should point out that for Crossbeam, it makes sense to use the Mutex with up to four threads; which, for many applications, might be sufficient.

8.3.2 Cavium ThunderX

The Cavium ThunderXis of a different nature than the remaining CPUs in this section, since it is an ARM machine. It also has a relative low clock speed. This manifests itself here in that the throughput in terms of absolute numbers is lower than the Intel® i7−4770 on multiple benchmarks, despite having four times the thread count.

In the `insert` benchmark we see a dip in throughput at 16 cores. This may be attributed to the number of threads being too high to run effectively on a single socket. However, if we calculate how many elements are inserted, we get $5M \times 16 = 80M$ elements; since the size of the pointer array is only $\approx 1M$, we get a load factor of ≈ 80, which means that inserts risk looking at 80 nodes before finding the correct place in the list to insert! In addition, the `remove` benchmark seems not to run into this problem. This suggests that it is in fact the capacity of the hash table that is the limiting factor, and not the cross-socket synchronization.



(a) `HashMap::insert`

(b) `HashMap::remove`

(c) `HashMap::contains`

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

Figure 8.3: HashMap performance on Cavium ThunderX

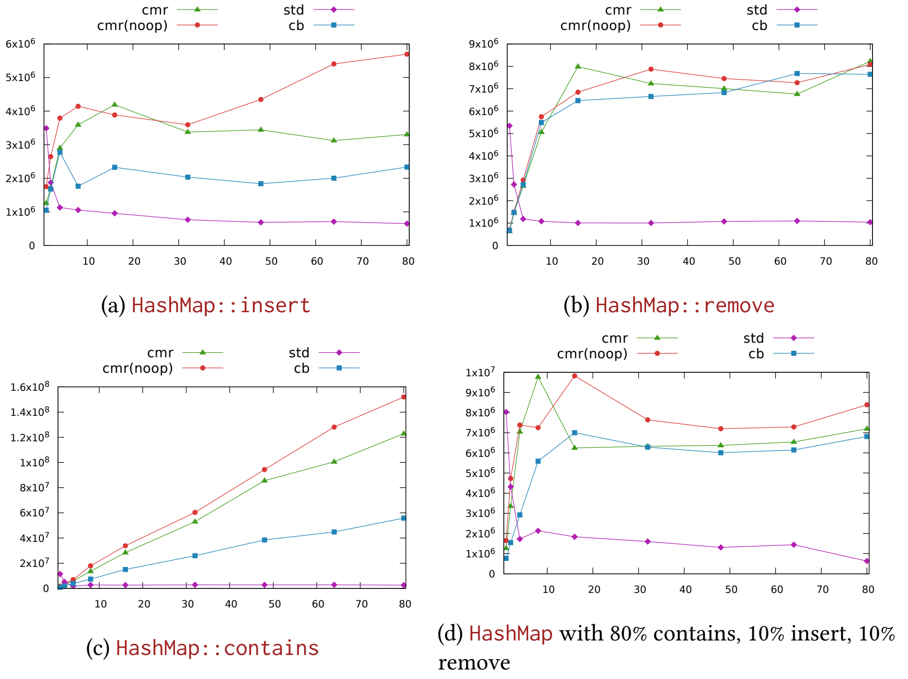
8.3.3 Intel® Xeon® E7−8870 and Intel® Xeon® Gold 6150

This section contains both the Intel® Xeon® E7−8870 and the Intel® Xeon® Gold 6150; this is done due to the similarities of both the CPUs and of the data.

The HashMap performance on the quad sockets is much more pessimistic than the graphs from the earlier sections; the operational throughput on both insert and remove evens out already after 16 and 36 cores for the Intel® Xeon® E7−8870 and Intel® Xeon® Gold 6150 respectively. This is probably because of the fact that only a limited numebr of threads are running on one socket, so that any shared memory loction that is modified by the HashMaps operations must be flushed to main memory. For us, this is the number of elements in the HashMap, which we need for resizing approproately.

A back of the envelope calculation supports this claim: The Intel® Xeon® E7−8870 runs at 2400 GHz, and with 16 threads we manage about $4\mathrm{M}$ inserts per second in total. This means that if we assume that all accesses to the count field happend sequentially, each access takes $\frac{2400\,\mathrm{GHz}}{4\mathrm{M}} = 600$ cycles per operation. While this is a lot for a single memory access, it is not too far off from main memory access latencies, which are often around 100 ns [14].

Yet another observation which supports the claim is that contains seems unaffected by the NUMA effects, as it does not mutate the HashMap in any way.



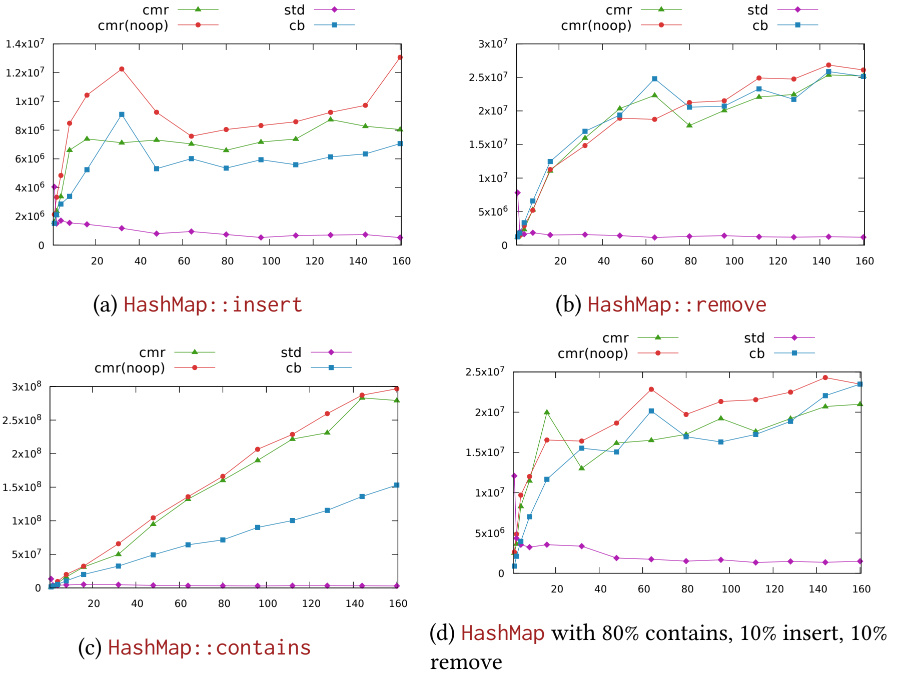
(a) `HashMap::insert`

(b) `HashMap::remove`

(c) `HashMap::contains`

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

Figure 8.4: HashMap performance on Intel® Xeon® E7−8870



(a) `HashMap::insert`

(b) `HashMap::remove`

(c) `HashMap::contains`

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

Figure 8.5: HashMap performance on Intel® Xeon® Gold 6150

8.4 Allocator

The choice of allocator is also shown to have a real effect. Fig. 8.6 shows the `HashMap::insert` benchmark while using the JeMalloc allocator (drawn lines) and the default system allocator (dashed lines). JeMalloc is optimized for multiple threads; however in this benchmark we clearly see a large increase in favour of the system allocator with 32 threads. This lead is however unique for all other data points, with the exception of a few of the data points from the Crossbeam hash table, when the thread count is 96, 128, and 144.

The general trend for the system allocator for both variants of CMR is downwards from its maxima at 16 threads, while neither allocator seems to affect the hash table from Crossbeam. This might mean that the allocation is not in the bottleneck for Crossbeam, whereas it is for the hash table implemented using CMR.

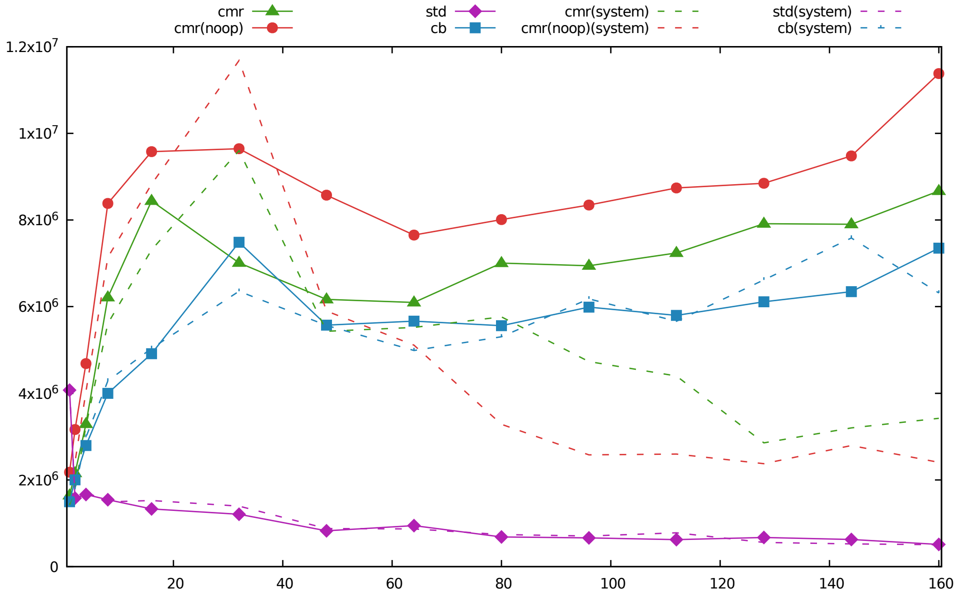


Figure 8.6: The `HashMap` insert benchmark using JeMalloc and the system allocator

Chapter 9
Conclusion

Are we there yet?

Unknown

In this report we have presented CMR, a new memory reclamation system for concurrent systems, in the Rust programming language. We have presented an abstract description of the system, and walked through the most important parts of the implementation. Then, we have shown usage code for the system, namely four concurrent lock-free data structures. After notes on practical matters of the implementation work, we have looked at experimental results of both CMRs primitive operations as well as the performance of the data strucutres implemented using CMR. What remains is a conclusion

9.1 Is CMR Useful?

The big question of this text is that of the usefulness of CMR. While the operational throughput measured of the hash table that used CMR was almost always above that of its main competitor in this thesis, the SkipList from Crossbeam, we note that the difference with and without the overhead of CMR often was quite significant; occationally the functioning version of CMR laid right in between Crossbeam and non-functioning CMR. It is therefore fair to assume that much of the difference is due to the difference in implementation, as noted in Section 8.3. This was not always the case however, and there is clear use cases where CMR really shows a low overhead, namely low write-intensive workloads.

A pain point of CMR is that it depends on forking the process. While modern operating systems leverate CoW optimizations on such operations, these might still be very expensive. Worse still, the cost of this operation, and thus also of CMR in total, is not dependant on the size of the subset of the process that is performing concurrent work, but the process as a whole: any large application, like a compiler, will most likely not be able to utilize CMR in a meaningful way, since the overhead of the fork operation is likely to be too high.

Still, we argue that CMR presents a simple API for programmers to work with; we require no explicit free calls, and implementation of rather intricate data structures has shown that a very low number of `unsafe` blocks is required in order to handle memory — this despite the very unsafe nature of concurrent programming. We believe that CMR may act as a good instructional example of a way of making a safe interface over an intrinsicly unsafe one, by utilizing the Rust type system.

9.2 Alternatives

There are still a numerous variants of concurrent memory reclamation scemes. For most applications it seems that is is hard to beat Epoch-based reclamation (Section 2.4.2), due to its very lean overhead. Considering that Rust also has a well implemented third-party crates for EBR, it is fair to say that the Rust ecosystem does not lack viable options for managing memory in a concurrent setting.

Other alternatives also exist, and there are numerous implementation of Hazard Pointers (Section 2.4.3), despite no one crate is sticking out as *the* go-to implementation of HPs. It seems that most users that require a memory management system in their code base prefer to implement their own version, so that they can tailor the implementation to their needs.

It seems that there is still room in the Rust ecosystems for contenders within the concurrency space, memory reclamation being no exception.

9.3 Future Work

Working in a relatively new and sparse problem space allows for many ideas to come to life during development, and CMR has not been an exception to this. Plenty of ideas have been considered, only for the author to realize that time is sparse.

The subset of a program in which threads are operating concurrently with other threads is usually rather small; not many tasks fit in this space. Additionally, for the data structures implemented in this thesis, much of the execution time is spent on allocation. For this reason it would make sense to have a specialized allocator for, say, a data structure. The allocator would then have access to patterns, like the size of allocations, or the lifetime of objects. For instance, an allocator for a `Queue` of a certain type would always be the same, and the lifetime would often be the same as the allocation order. This might allow performance gains.

Despite forking being a pain point of CMR, it might be possible to use the idea of a custom allocator to limit the pages in which shared memory resides. If we could limit the types of objects referenced from the shared memory, we might get away with not having to copy the entire memory space when we `fork`, but only the parts of the memory that is allocated for concurrent use. This could greatly reduce the overhead of forking, and would make the overhead of such as scheme independent of the total memory space of the process, but only dependent on the memory used for concurrent operations.

Many data structures and applications still remains to consider for CMR, in order to see whether a system that is similar to CMR is featuble for real-world usage. The primary issue in performing such a survey today is that there is simply a lack of use cases: CMR is heavily dependent on Rust, and there are simply not many large enough applications to make a fair comparison between CMR and, say, the GDW-GC.

It is still not clear how memory management for concurrent systems can — or even *if* they can — be unified with static analysis, such as the Rust borrow checker, for concurrent systems, or in what degree programming langauge rules can help programmers utilize the system they are programming on while still helping the programmer not to make mistakes. While Rust certainly is a

Bibliography

[1] Rust programs vs C gcc. https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust.html.

[2] Crossbeam SkipList. https://github.com/crossbeam-rs/crossbeam-skiplist, 2018.

[3] The go programming language. https://www.golang.org, 2018.

[4] The kotlin programming language. https://www.kotlinlang.org, 2018.

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

[6] The LLVM Compiler Infrastructure. https://llvm.org, 2018.

[7] Alistarh, D., Leiserson, W., Matveev, A., and Shavit, N. Forkscan: Conservative memory reclamation for modern operating systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 483–498.

[8] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748.

[9] Bohem, H. Bdw: A garabge collector for c and c++. http://www.hboehm.info/gc/, 2018.

[10] Brooks Jr, F. P. The mythical man-month (anniversary ed.).

[11] Brown, T. Good Data Structure Experiments are R.A.R.E. https://www.youtube.com/watch?v=x6HaBcRJHFY.

[12] 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.

[13] Carruth, C. Garbage in, garbage out: Arguing about undefined behavior, cppcon 2016. https://www.youtube.com/watch?v=v1COuU2vU_w, 2018.

[14] Chang, J. Memory Latency - Return to Single Processor. http://www.qdpma.com/ServerSystems/MemLat2018.html, 2018.

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

[16] Collins, G. E. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (1960), 655–657.

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

[18] Fraser, K. Practical lock-freedom. Tech. rep., University of Cambridge, Computer Laboratory, 2004.

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

[20] glibc Wiki. MallocInternals, glibc wiki. https://sourceware.org/glibc/wiki/MallocInternals, 2018.

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

[22] Harris, T. L., Fraser, K., and Pratt, I. A. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing* (2002), Springer, pp. 265–279.

[23] Hewitt, C., Bishop, P., and Steiger, R. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference* (1973), vol. 3, Stanford Research Institute, p. 235.

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

[25] Knuth, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd Edition*. Addison-Wesley Professional, 1997.

[26] Luchangco, V., Moir, M., and Shavit, N. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures* (2003), ACM, pp. 314–323.

[27] 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.

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

[29] 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.

[30] Parkinson, M., Vaswani, K., Costa, M., Deligiannis, P., Blankstein, A., McDermott, D., Balkind, J., and Vytiniotis, D. Project snowflake: Non-blocking safe manual memory management in .net. Tech. rep., July 2018.

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

[32] Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU). https://issues.isocpp.org/show_bug.cgi?id=382.

[33] Regehr, J. Undefined behavior in 2017, cppcon 2017. https://www.youtube.com/watch?v=v1COuU2vU_w, 2018.

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

[35] The Rust Programming Language on GitHub. http://github.com/rust-lang/rust.

[36] Sanjay Ghemawat. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2018.

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

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

[39] Thoresen, M. H. Implementing concurrent memory reclamation schemes. Tech. rep., Norwegian University of Science and Technology, December 2017.

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

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

[42] Valgrind. http://valgrind.org/, 2018.