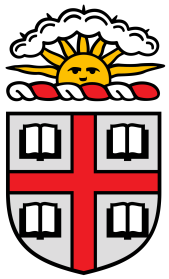
SoftRS: Programming Abstractions for Safe Soft Memory

Megan Frisella

Advisor: Malte Schwarzkopf

Reader: Shriram Krishnamurthi



Department of Computer Science

Brown University

Providence, RI

April 2022

Contents

Abstract

Memory is the bottleneck resource in today's datacenters because it is inflexible: low-priority processes are routinely killed to free up resources during memory pressure. This wastes CPU cycles upon re-running killed jobs and incentivizes datacenter operators to run at low memory utilization for safety. Prior work introduced *soft memory*, a software-level abstraction on top of standard primary storage that, under memory pressure, makes memory revocable for reallocation elsewhere. Unless soft memory access is handled with extreme care, dynamically-revocable memory can lead to memory and concurrency bugs. This makes programming with soft memory challenging for application developers. This thesis introduces SOFTRS, a Rust crate that provides abstractions for programming with soft memory. SOFTRS proposes and prototypes *use guard*, *soft pointer*, and *soft data structure* abstractions, designed for ergonomic, memory-safe, and concurrency-safe programming with soft memory. SOFTRS abstractions add some overhead compared to working with traditional data structures and pointers, but most of the overhead is incurred during slow-path operations.

Acknowledgements

Many people made this work possible. Thank you to Will Crichton for his advice and recommendation to pursue a mutex guard-like design for use scopes, which became use guards. Thank you to Shriram for being the reader on this work, and for advocating for and advising me throughout my research journey at Brown. Thank you to Shirley and Howie for being excellent collaborators on the broader soft memory project. It has been a pleasure to work and develop this project together. Thank you to the ETOS lab for the community that it provides. Research can be hard, so it has been invaluable to have a supportive community of peers and collaborators.

Finally, thank you to my advisor Malte, who believed in me enough to let me start running with the soft memory idea back in the Fall of 2021. I appreciate that he has given me space to ideate my own designs and explore new directions as they have interested me. Malte's support and encouragement of me has been unwavering in the face of technical and personal ups and downs over the years, and is to thank for the success of this work and my growth as a researcher. I am very grateful to have gotten a start to my research career in the ETOS lab.

Chapter 1

Introduction

Memory is inherently inflexible: Once DRAM is allocated to applications on a machine, it cannot be used elsewhere until the application explicitly frees the resources or terminates. Typical processes only actively use a small fraction of their allocated memory at any time; many of them have cold memory or extensive application-specific caches [7]. The inflexibility of memory allocation means that infrequently-accessed or unimportant memory can't be re-purposed to needier applications. This makes memory a bottleneck resource in many of today's data centers [16, 18].

A common way to make memory flexible is to swap out memory content to more abundant, higher-latency storage, such as flash or disk. This temporarily frees up space for new allocations until the program swaps old memory back when accessing its content. Swapping solutions introduce performance overhead to send and retrieve memory to/from higher-latency storage. This does not make sense for applications whose data loses its utility when it is no longer in memory, e.g., as with in-memory caches. Swapping solutions are also transparent to the application. Transparency is nice from a developer perspective because the developer does not need to reason about swapped-out pages in application logic – swapping pages back in is taken care of by the system. However, a lack of transparency also means that developers cannot incorporate application semantics into swapping policy, which is useful for performance engineering.

1.1 Soft memory idea and overview

Prior work proposed *soft memory* [5], a software-level abstraction on top of standard primary storage, which makes memory allocations revocable under memory pressure for reallocation in other applications. Soft memory investigates what would happen if memory was a more fungible resource—like CPU or I/O time, resources that OS kernels reallocate dynamically between processes—and what abstractions are needed to make this work. Soft memory differs from swapping by actually *revoking* and dropping memory contents (which we refer to as soft state), rather than moving them to slower storage. Soft memory is also explicit rather than transparent to the application, allowing application semantics to inform reclamation policies.

In a data center that deploys soft memory, a developer uses soft state for caches, look-up tables, temporary requests queues, and data structures with similar non-essential purposes. They continue to use standard memory to store critical state that enables the correct functioning of the application (i.e., authentication records, data structure metadata, etc.). For example, the entries in database caches may employ soft memory, but table schemas and active user metadata would remain in traditional memory.

In large-scale computing clusters, jobs typically have a memory limit above which they get terminated by the scheduler. Granting a soft memory budget on top of the traditional memory limit allows productive use of memory left idle because other jobs are operating below their limit. The scheduler can continue to constrain traditional memory by maintaining limits on it, but developers are encouraged to use soft memory to opportunistically take advantage of extra available resources.

In effect, soft memory has three major benefits. First, it reduces the number of low-priority job terminations due to memory pressure. Such *evictions* waste cluster resources on incomplete executions, including resources scarcer than memory, such as accelerators (e.g., GPUs, TPUs). Second, rather than failing `malloc` when there is insufficient memory on a machine, soft memory allows an allocator to retrieve it from other processes. Since most applications cannot handle failed memory allocations gracefully, soft memory avoids crashes by satisfying the immediate memory need. Third, it incorporates application semantics into soft memory management and reclamation to enable performance-aware reclamation strategies.

1.2 Challenges

Programming with soft memory is hard. Unless soft memory access is handled with extreme care, dynamically-revocable memory can lead to memory and concurrency bugs. Prior work identified that new abstractions are needed to make soft memory work [5], but did not build out these abstractions. Developers were expected to manage the memory safety of reclamation by repairing dangling references into reclaimed regions of memory. The reclamation runtime was also vulnerable to racing with application threads to access soft memory. In summary, we identify the following key challenges of soft memory:

1. Abstractions and ergonomics for soft memory
2. Concurrency safety of reclamation
3. Memory safety of reclamation

We identify the need for abstractions that enable safely programming soft memory in a way that does not deviate too much from traditional programming patterns. The abstractions should ensure *concurrency safety* by preventing the reclamation runtime from racing with application threads when dropping soft objects. The abstractions should ensure *memory safety* by preventing an application from dereferencing memory that the application no longer owns.

1.3 Key ideas and contributions

This thesis builds on prior work to design, prototype, and evaluate abstractions for soft memory. The abstractions aim to offer an ergonomic, memory-safe, and concurrency-safe interface for using soft memory. Our key contributions are:

1. The co-design of a soft memory manager and soft data structures for managing soft state and reclamation in an application. This design introduces *soft pointer* and *soft data structure* abstractions.
2. The design of a *use guard* abstraction to enforce concurrency-safe soft memory reclamation.

3. SOFTRS, a Rust crate that prototypes *use guard*, *soft pointer*, and *soft data structure* abstractions to provide ergonomic, memory-safe, and concurrency-safe programming with soft memory.

1.4 Related publications

Some material in this thesis was previously published in HotOS 2023 [5] in collaboration with Shirley Loayza-Sanchez and Malte Schwarzkopf (background sections 2.1, 2.3 and 2.4 contain text from that publication). The design and implementation of SOFTRS presented in this thesis is followup work completed by myself. There is parallel work which is complementary to this thesis by Shirley Loayza-Sanchez and Howie Chen that extends soft memory to remote servers and introduces soft memory recomputation.

Our prior work [5] presents a prototype soft memory allocator (also referred to as a soft memory manager) that manages soft memory in an application, and a userspace soft memory daemon (also referred to as a soft memory coordinator) that coordinates soft memory budgets and reclamation on a machine. Upon a reclamation request, the soft memory allocator returns memory resources to the OS by unmapping pages.

We also present soft data structures, which offer familiar data structure APIs to developers, implement data structure-specific reclamation and repair logic, and expose a callback interface for application logic to be notified of soft memory allocations that are about to be dropped during reclamation. We added soft memory support to the Redis key-value store to investigate the practicality and performance of soft memory, and found low memory reclamation overheads during memory pressure. See Frisella, Loayza-Sanchez, and Schwarzkopf [5] for a full discussion of the system.

Chapter 2

Background and Related Work

2.1 Soft memory

Soft memory presents an opportunity to improve resource management under in common datacenter settings. This section expands on the benefits and use cases of soft memory.

Memory Under-utilization. Cluster schedulers allocate initial resources to jobs in response to requirements specified by the developer or to predicted peak load. However, engineers are notoriously bad at estimating actual resource consumption and workload requirement estimation remains fairly conservative in data centers as deployments provision for peak load [2, 14]. This leads to widespread resource under-utilization, as identified in large computing cluster traces [3, 9, 16]. At the core of the problem, there is a trade-off between maximizing the use of hardware resources and avoiding performance degradation and disruption when requests exceed availability. Large-scale schedulers such as Google's Borg decide to terminate lower-priority jobs when they receive memory requests that cannot be satisfied otherwise [17]. This is undesirable, as often work completed by the evicted job must be recomputed at a later time. Soft memory eliminates the utilization-performance trade-off for the memory resource, opening the doors to maximizing memory utilization without risking process terminations. A soft memory allocator can allocate memory even when the memory on a machine is fully utilized, because it can revoke older soft memory allocations, transferring allocated memory between jobs without triggering evictions.

Shifting Resource Consumption Patterns. Even though compute clusters run many heterogeneous workloads, some universal resource consumption patterns exist in most large data centers. For example, low nocturnal user interaction with web services leads to reduced utilization of pre-assigned resources [1, 18]. In particular, spare CPU resources exist when the load on long-running services is low, even though their memory footprint remains the same. Soft memory helps applications and datacenter operators scale out during low-utilization periods. Extra workloads can reclaim the soft memory in under-utilized services and use it productively, which reduces CPU stranding. This suggests a two-level memory scheduling strategy: a cluster scheduler primarily decides a-priori on traditional resource memory allocations, while a lower-level soft memory scheduler redistributes revocable memory while jobs run. This increased fungibility allows reclaimed soft memory to be repurposed when needed, and we expect that jobs employing soft memory will benefit from higher likelihood of being scheduled.

Example Use-cases: Consider a datacenter where a long-running web service uses Redis [12] as an in-memory cache to reduce tail-latency. During nocturnal lulls in traffic, the web service can operate on a much smaller cache footprint without harming tail latency. Redis can put the cache in soft memory, so that when batch jobs in the datacenter scale up at night, they can reclaim part of the cache memory. The cache can be scaled back up during the day when latency is critical and batch jobs have finished.

The input data pipeline is a bottleneck in machine learning (ML) training jobs as accelerators process data faster than a dataset batch gets loaded into memory [6]. Increasing cache size via soft memory can provide performance gains while productively using otherwise idle memory. Once this memory is needed again, the soft memory subsystem re-configures the cache to its original size. This slows down the ML training, but makes memory available for other workloads like latency-critical service jobs.

2.2 Midas

The Midas system also proposes soft memory, and was developed independently of this work [11]. Midas develops similar abstractions for soft memory, including soft pointers and soft data structures, but unmaps pages from a kernel module rather than relying on applications to cooperate with reclamation requests. Midas does not prevent the runtime from

evacuating a soft object while it is dereferenced by the application. Midas does not explicitly manage dangling raw pointers into reclaimed memory regions, but instead handles such dereferences with page faults, which cause the application to recompute the page contents.

2.3 Swapping and far memory

Soft memory is related to work on swapping and far memory. Swapping solutions make memory dynamic by moving data to slower storage under memory pressure, to be swapped back into memory when it is accessed. For example, zswap [7] proactively compresses cold memory pages and maintains their compressed version in DRAM. Far memory systems facilitate swapping to remote machines. AIFM [13] introduces far data structures that swap part of their memory to remote storage under memory pressure. AIFM presents a *remotable pointer* abstraction which gracefully catches page faults triggered by dereferencing memory that was remoted, and swaps the memory back in. AIFM presents a *dereference scope* abstraction that facilitates synchronization between application threads and the evacuator, which swaps pages to remote storage.

Soft memory differentiates itself from swapping solutions in two ways. First, it actually *drops* memory under memory pressure, which is more performant than swapping and useful in cases where data loses its utility once no longer in memory, e.g., in-memory caches. Second, soft memory is not transparent to the application. Developers must explicitly opt-in to using soft memory via soft data structures, and can write per-data structure memory reclamation policies that are informed by application semantics. This enables the application developer to do performance engineering by implementing policies that prioritize soft allocations for reclamation based on application semantics.

2.4 Garbage collection

Under memory pressure, managed language garbage collectors free inaccessible objects to make space for new allocations [4]. By contrast, soft memory enables live objects to be freed when free memory is unavailable on a system. Prioritized garbage collection realizes space-aware caches (“Saches”) via a soft-reference-style API that allows the garbage collector to eagerly reclaim objects in caches [10]. This realizes one key use case for soft memory, but

```rust
fn main() {
    let r: &String;
    {
        let s1 = String::from("chowder");
        let s2 = s1;                // s2 takes ownership of the String
        // println!("{}", s1);      <-- compile-time error: s1 was dropped
        let r = &s2;
        let l = get_length(&s2);    // multiple immutable borrows OK
        println!("{}", s2);         // s2 is still valid
    }
    // println!("{}, {}", r, l);    <-- compile-time error: s2 was dropped
}
fn get_length(s: &String) -> usize {
    s.len()
}
```

Listing 2.1: Example Rust program demonstrating Rust ownership and borrow-checking.

realizes it in the context of a managed language with GC. We implement soft memory in C/C++ and also provide an interface for Rust, none of which are managed languages.

2.5 Rust

Programming languages manage memory in different ways. Some languages, like C/C++, leave memory management entirely to the programmer. The programmer must allocate and free the memory resources they wish to use. Other languages, like Java and C#, implement automatic memory management via garbage collection. This entails periodically scanning the entire heap for dead objects (objects that are no longer reachable via reference) and freeing them. Rust takes a different approach. Rust's compiler ensures that memory is managed safely at compile time using two key concepts: *ownership* and *lifetimes*.

Ownership. Rust's compiler enforces a set of ownership rules, which enable the compiler to automatically insert memory management operations like allocations and frees. The rules are:

1. Each value in Rust has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped and its memory freed.

When the developer follows these rules, the compiler knows exactly when to free an object because when the owner goes out of scope, the object is no longer reachable.

References, borrows, and lifetimes. Things become more complicated when developers want to work with a pointer to a value. Rust has *references*, which are like pointers, but Rust guarantees at compile time that the reference is valid (it actually points to a value of the specified type, and not some random bytes). A reference *borrows* the underlying value, so it must eventually give it back. Borrows are specified as a mutable or immmutable. Mutable borrows have permission to modify the underlying value, and non-mutable borrows do not. Rust's borrow rules are:

1. At any time, a value can only have either one mutable borrow or multiple immutable borrows.

2. A value cannot be dropped while it is borrowed.

Rust enforces these rules at compile-time. The first rule helps prevent data races; when a mutable reference exists, it is guaranteed to be the only reference mutating the value until it goes out of scope. As for the second rule, every reference in Rust has a *lifetime*. Lifetimes provide static information about the scope that references are valid for. Rust uses borrow rules and lifetimes to ensure that a value is not dropped while there is a reference to it, preventing dangling pointer memory bugs. See Listing 2.1 for an example of Rust ownership, borrow-checking, and lifetimes.

2.6 Smart pointers

C++ smart pointers add memory management semantics to C++ by ensuring that an object is deleted once it is no longer referenced. Smart pointers introduce a notion of *ownership* to C++ objects. This section details the three types of C++ smart pointers and their Rust analogues, where applicable.

Unique pointers. A C++ unique pointer owns and manages and object. The pointer drops the object when either the pointer goes out of scope or the pointer is assigned another object. A unique pointer can give up ownership of an object, like a *move* in Rust. Crucially, a unique pointer does not support copying; copying a unique pointer will error at compile time.

Shared pointers. C++ shared pointers own and manage an object, and support copying to create multiple shared pointers that own/manage the same object. Shared pointers do

reference counting to track how many pointers share ownership of an object. When the program drops or resets the last shared pointer owning an object, that shared pointer drops the managed object. Rust provides an analogue to C++ shared pointers, called reference-counted pointers in Rust (`std::rc::Rc`). Reference-counted pointers in Rust support cloning to create multiple pointers that reference the same object. Like C++ shared pointers, when a Rust program drops the last reference-counted pointer to an object, that reference-counted pointer drops the managed object.

Weak pointers. C++ weak pointers are smart pointers that reference an object without owning the object. Weak pointers enable programs to safely reference objects that may be deleted by some owner. A weak pointer maintains a reference to the shared pointer that constructed it. The program "locks" a weak pointer to get either a shared pointer to the underlying object, if it exists, or a null value if some owner dropped the underlying object. Rust provides an analogue to C++ weak pointers, called weak pointers in Rust (`std::rc::Weak`). Rust weak pointers are a variant on the Rust reference-counted pointer, and hold a reference to an object without owning it. A Rust program "upgrades" a weak pointer to get either a reference-counted pointer to the underlying object, if it exists, or a null value; weak pointers instrument this using the Rust `Option` type. Java provides an analogue to C++ weak pointers, called weak references (`java.lang.ref.WeakReference<T>`).

Weak pointers do not contribute to the managed object's reference count, so they do not prevent shared pointers in C++, reference-counted pointers in Rust, and the garbage collector in Java from dropping the underlying object. The use case for weak pointers is to avoid cycles that prevent unreachable objects from being freed; a cycle of references means that an object's reference count never reaches zero, even if the memory is no longer in use.

AIFM remotable pointers. AIFM introduces *remotable* shared pointers and unique pointers based on C++ shared and unique pointer semantics. Remotable pointer dereferences handle swapping memory back in if the AIFM evacuator swapped the memory to remote storage.

Chapter 3

Design

3.1 Soft memory abstraction

This section presents abstractions that make soft memory allocation and reclamation realizable with memory safety and in Rust. We propose a *soft pointer* abstraction similar to C++ smart pointers, and present a co-design for a soft memory manager and soft data structures, which together enable soft memory management and reclamation.

3.1.1 Soft pointers

A soft pointer maintains a reference to soft state that is either available or dropped. A soft pointer overloads the dereference operation to catch the case when the underlying soft state was dropped and handles this in a configurable way. This thesis presents a soft pointer that can be plugged into different backends, each of which may take different approaches to handling dropped memory. For example, a backend may be equipped to recompute soft state, or instead opt to return a `None` or `Error` value.

3.1.2 Soft memory manager and soft DS co-design

The soft memory manager is the runtime component that manages soft memory requests and reclamation in an application. We propose design principles for a soft memory manager that are agnostic to its implementation. The actual memory allocator used by the manager is implementation-dependent. We identify the following principles to realize practical soft memory management and reclamation:

Per-data structure soft memory management. The runtime handles reclamation by dropping individual soft objects. This approach incorporates application semantics into reclamation by enabling application logic to prioritize certain soft objects over others for reclamation. The application must be careful to drop enough soft objects in order to free up whole pages for a reclamation request. This poses a trade-off between space and the number of allocation frees required to free up entire pages for reclamation. A policy where applications free objects from arbitrary heap locations until enough entire pages are free would result in large numbers of allocation frees to fulfill a reclamation quota. A policy where the memory manager grants each allocation its own page permits straightforward reclamation (an entire page is freed by one allocation free) but wastes copious amounts of space if most allocations are small (as is commonly the case [8]). The proposed memory manager design manages memory per-data structures to balance this trade-off; the memory manager tracks a separate page pool per soft data structure. This leads naturally to the next design point.

Per-data structure soft memory reclamation. The soft memory manager should accept reclamation requests from a machine-wide soft memory coordinator[1] and choose data structure(s) to reclaim from. As described in the previous paragraph, concentrating object reclamations within a few data structures facilitates effectively freeing up entire pages. We propose that developers encode their own logic for dropping elements and repairing the data structure during reclamation. For example, a soft vector reclamation policy may opt to give up older elements sooner than newer ones, or colder elements sooner than hotter ones. Reclamation-per-DS is a key abstraction that enables the programmer to reason about how reclamation will impact their application, and lets developers mix-and-match data structures with the particular reclamation semantics they desire.

Consider how a soft vector could repair itself. For some application semantics, it is reasonable to simply remove the dropped element and shrink the vector, while for others it is useful to maintain a placeholder for the soft object, which is no longer backed by a memory page, in order to facilitate later recomputation. In SOFTRS, the advanced developer has the freedom to encode their own reclamation and reconstruction logic into a soft data

[1]The coordinator is responsible for managing soft memory resources on a machine and sends reclamation requests to applications when memory pressure is high. Its implementation is independent of this work.

structure. Off-the-shelf soft data structures provide reasonable default policies for developers who don't need this level of customization.

Reclamation runtime is concurrent with application threads. In order to minimize the performance impact of memory reclamation on an application, the soft memory manager should not interrupt application threads to run reclamation logic. Instead, the soft memory manager should manage a separate thread designated for reclamation. This is similar to existing designs for remote memory, such as AIFM's evacuator, which runs concurrently to application threads [13].

3.1.3 Soft Data Structures

SOFTRS designs soft data structures based on the following principles:

1. Soft data structures store data behind soft pointers, which store data in soft allocations.
2. Soft data structures provide reclamation logic that implements a policy for freeing elements upon request by the memory manager.
3. Soft data structures offer a programming experience similar to traditional data structures, except for well-defined semantics for access to reclaimed data.

The application must inform the soft memory manager of each soft data structure that exists, so that the manager can target data structures for reclamation at runtime. To this end, SOFTRS registers soft data structures with the soft memory manager upon data structure construction. Upon receiving a reclamation request from a machine-wide soft memory coordinator, the memory manager chooses data structure(s) to reclaim from and calls into its corresponding reclaim logic, passing in the number of bytes the data structure should free up.

3.2 Programming with soft memory

The soft pointer and soft data structure abstractions lay the foundations for managing soft state in an application and effectively releasing it under memory pressure. This section details the API for soft pointers and soft data structures in SOFTRS. On their own, however, these abstractions provide only part of the story for safely accessing soft state during reclamation. The reclamation runtime is concurrent with application threads, leaving soft

```
pub struct SoftPtr<T> { data: Option<Box<T, SoftAllocator>> }
pub fn new(v: T, id: usize) -> Self
pub fn deref(&self) -> Result<&T, Error>
pub fn deref_mut(&mut self) -> Result<&mut T, Error>
pub fn use_guard(&self) -> Result<UseGuard<SoftPtr<T>>, Error>
pub fn mut_use_guard(&mut self) -> Result<MutUseGuard<SoftPtr<T>>, Error>
```

Listing 3.1: Soft pointer Rust API. Use guards are discussed in §3.2.3.

pointer dereferences in the application vulnerable to racing with the reclamation thread. To address this challenge, this section introduces a *use guards* abstraction to enforce thread-safety in the presence of a reclamation runtime.

3.2.1 Soft pointer API

Listing 3.1 shows the soft pointer Rust API. SOFTRS only permits soft pointers that are associated with a soft data structure because reclamation logic and associated synchronization is managed per-DS. This is why the soft pointer constructor requires a `usize` argument that represents the unique ID of the pointer's associated soft data structure.

To get the contents of a soft object, the programmer dereferences it via the `deref` and `deref_mut` API methods. These methods return a Rust `Result` that contains an error if the data was reclaimed. It would be convenient to implement the Rust `Deref` and `DerefMut` traits for `SoftPtr`, but the type signature for `Deref::deref` and `DerefMut::deref_mut` is too restrictive; a `SoftPtr` needs flexibility to return an error instead of a reference to the underlying value in the case that the data was dropped. The programmer must handle the case when the object was dropped by unpacking the `Result`. Internally, the soft pointer calls into the memory manager to allocate and free soft objects. The soft pointer stores data as an option type so that, upon reclamation, it can replace the dropped data with `None`.

Ongoing work builds out a more sophisticated soft memory manager than the prototype presented in this thesis. The new backend will support the recomputation of dropped objects upon dereference. This backend can be swapped into the current Rust soft pointer API with minimal changes: First, the FFI call into the soft memory manager needs to be swapped out. Second, the soft pointer can store data as `Box` rather than `Option<Box>`.

```
pub struct SoftDSMeta<T> {
  id: usize,
  callback_fn: fn(ptr: &SoftPtr<T>),
  status: Arc<AtomicU8>,
}
pub trait Reclaimable {
  fn probe(&mut self) -> Option<&mut SoftPtr<T>>;
  fn reclaim(&mut self, nbytes: usize);
}
pub trait Guardable {
  fn use_guard(&self) -> Result<UseGuard<Self>, Error>;
  fn mut_use_guard(&mut self) -> Result<MutUseGuard<Self>, Error>;
}
```

Listing 3.2: Soft data structure Rust API. See Listing 3.1 for `UseGuard` details.

3.2.2 Soft data structure API.

Listing 3.2 presents the soft data structure Rust API. SOFTRS implements a layered design. The details of the layered design and the *guardable* interface relate to synchronization with the reclamation runtime. Section §3.2.3 discusses these details. This section discusses the *reclaimable* interface, by which programmers tailor soft data structure reclamation to application semantics. The `Reclaimable` trait requires two method implementations: `probe` and `reclaim`.

`Probe` selects and returns a mutable reference to a non-reclaimed element in the data structure. If there are no objects or all objects have been reclaimed, it returns `None`. The application programmer implements `probe` in order to specify the order in which the data structure gives up objects. SOFTRS guarantees that an arbitrary `probe` implementation actually returns a reference to a `SoftPtr` stored in the corresponding soft data structure; Rust borrow-checking guarantees that if probe returns a `&mut SoftPtr<T>`, then the data structure must own the soft pointer and the soft pointer must live as long as the data structure.

`Reclaim` calls into probe as a subroutine in order to implement data structure reclamation. A reasonable implementation for `reclaim` probes one element at a time and frees them until the data structure has freed enough bytes to satisfy the request. Depending on application semantics, an application programmer may wish to alternatively batch object reclamations. Different soft data structure designs also facilitate different reclamation procedures. This discussion has focused on soft data structures which store elements in separate soft allocations, but a data structure implementation which backs the entire data structure by a single soft allocation facilitates “all-or-nothing” reclamation.

```
pub struct UseGuard<'a, T> {
  inner: Arc<AtomicU8>,
  data: UnsafeCell<&'a T>,
}
UseGuard::new(ds: &'a T,
  inner: Arc<AtomicU8>);
UseGuard::deref(&self)
  -> &&'a T;
UseGuard::drop(&mut self);
```

```
pub struct MutUseGuard<'a, T> {
  inner: Arc<AtomicU8>,
  data: UnsafeCell<&'a mut T>,
}
MutUseGuard::new(ds: &'a mut T,
  inner: Arc<AtomicU8>);
MutUseGuard::deref_mut(&mut self)
  -> &mut &'a mut T;
MutUseGuard::drop(&mut self);
```

Figure 3.1: The SOFTRS `UseGuard` and `MutUseGuard` API, and trait implementations.

3.2.3 Use guards

SOFTRS presents a *use guard* abstraction that synchronizes soft data structures with the memory manager's reclamation runtime. The goal is to prevent the reclamation runtime from reclaiming objects behind soft pointers while the data structure is in-use in the application. At a high level, use guards block the reclamation runtime from touching a soft data structure while an application thread is using it. AIFM dereference scopes and Rust mutex guards inspire this abstraction [13]. Figure 3.1 presents the Rust use guard API.

Motivation. Application-level synchronization can only guarantee race-freedom with the reclamation runtime as long as the application *correctly* synchronizes soft data structure usage. Suppose that SOFTRS provided `SoftDS::in_use` and `SoftDS::all_done` methods that indicate to the reclamation runtime when a data structure is in use. The application may forget to call `in_use` before invoking a method on the soft data structure. If the soft memory manager begins reclamation at the same time, the application is at risk of dereferencing a pointer that points to memory which was just dropped. The application may call `in-use`, but forget to call `done`. This is okay from a correctness perspective, but starves the reclamation runtime from ever accessing the data structure. Instead of leaving it up to the application programmer to correctly synchronize soft data structure access with the reclamation runtime, SOFTRS proposes a solution that:

1. guarantees correct synchronization of application soft data structure usage with the memory manager's reclamation thread,

2. avoids forcing the application to synchronize with the reclamation thread upon every data structure access (e.g. by embedding synchronization logic inside every method call),

3. allows the programmer to manage the tradeoff between synchronization overhead and reclamation latency, and

4. supports application semantics which require a sequence of data structure operations to be performed atomically w.r.t. the reclamation runtime.

SOFTRS implements one design to meet these goals, which is inspired by AIFM dereference scopes and Rust mutex guards. Section 3.2.5 explores alternative designs.

AIFM dereference scopes. AIFM dereference scopes create an evacuation fence which block evacuations until the program drops the dereference scope [13]. Programs must construct a dereference scope before accessing a remotable object and destruct it when they are done. AIFM data structure methods require a `DerefScope` parameter to ensure that the program only invokes data structure methods within a dereference scope. The `DerefScope` is not data structure-specific; dereference scopes block evacuation on the entire application whenever the program accesses a single data structure. The SOFTRS design takes inspiration from AIFM dereference scopes with a few key differences. SOFTRS instruments synchronization per-data structure to allow the reclamation runtime to get memory from an application even while some data structures on the application are in-use. Another difference is that AIFM aborts an active reclamation as soon as the application starts using a soft object. The SOFTRS implementation instead utilizes the `status` atomic integer for two-way synchronization. During an active reclamation, the affected data structure is blocked until the reclamation is done.

Rust mutex guards. Rust mutexes take ownership of the object that they are synchronizing. In order to access the object, a program locks the mutex. The action of locking a mutex grants the program access to the object via a mutex guard. The program dereferences the mutex guard to obtain the underlying object, and when the program drops the mutex guard then the lock is implicitly released. Crucially, the `std::Mutex` implementation ties the lifetime of the mutex guard to the lifetime of the reference to the object it guards, so that the object cannot be referenced after the program drops the mutex guard. The SOFTRS design for

```
1 fn mutex_guard() {
2   let vec = Mutex::new(Vec::new());
3   {
4     let mut g = vec.lock()
5                    .unwrap();
6     g.push(1);
7   }
8   // vec.push(1); <-- compile error
9 }
```

```
1 fn use_guard() {
2   let vec = SoftVec::new();
3   {
4     let mut g = vec.use_guard()
5                    .unwrap();
6     g.push(1);
7   }
8   // vec.push(1); <-- compile error
9 }
```

Figure 3.2: Rust `std::sync::Mutex` and `std::sync::MutexGuard` API (top) compared with the SoftRS use guard API (bottom). Both solutions leverage `method not found` compile-time errors to prevent un-guarded access to internal state.

synchronizing the reclamation runtime with the application takes inspiration from mutex guards to provide guarded access to soft data structure methods.

SoftRS use guards. Figure 3.1 presents the Rust use guard API. The `UseGuard` synchronizes read-only access to the underlying soft data structure. A generic type parameter `T` represents the type of the underlying soft data structure. The `MutUseGuard` synchronizes write access to the underlying soft data structure. Rust's borrow rules guarantee that the program can only ever construct either one `MutUseGuard` or multiple `UseGuards`. This section covers the `UseGuard` design, without loss of generality, because the immutable and mutable versions are designed in the same way.

There are relevant parallels between use guards and Rust mutex guards. Figure 3.2 compares `UseGuard` usage to `MutexGuard` usage. A `UseGuard` is associated with a soft data structure, and synchronizes with the soft DS's `status` atomic in order to provide access to data structure methods. Analogously, the soft data structure here is the mutex, the data structure's internal state is the data being guarded, and `UseGuard` is the `MutexGuard`. In order to access the soft data structure's internal state, the program calls `use_guard` to get a `UseGuard`, which references the internal data structure (Fig. 3.2, lines 4) with a lifetime tied to the lifetime of the `UseGuard` object. The program can dereference the `UseGuard` to use the data structure (Fig. 3.2, lines 6), but once the use guard goes out of scope (Fig. 3.2, lines 7), the use guard `Drop` logic resets the `status` atomic to 0 and drops the reference to the internal data structure. The key difference between mutex guards and use guards is the synchronization that they perform. Mutex guards synchronize access to the data they guard

across application threads. Use guards synchronize access to the data they guard between the reclamation runtime and application. Use guards do not synchronize access between application threads.

Key to preventing the programmer from accessing soft data structure methods without a use guard is a layered soft data structure implementation. The top-level soft data structure API only implements `SoftDS::new` and `use_guard`. This top-level struct stores an inner struct which is the actual soft data structure implementation. The inner data structure implements `Guardable::reclaim`, `Guardable::probe`, and all the expected data structure functionality (e. g., `push` for a soft vector).

The use guard design meets the SOFTRS design goals:

1. The program must correctly use `UseGuard`s, by design; a program cannot access a data structure without constructing a `UseGuard`.

2-4. Use guards are flexible; the program can persist a single use guard for one or multiple data structure operations. Operations via the same use guard are atomic w.r.t. the memory manager.

Synchronizing the memory manager. AIFM opts to set an "evacuating" bit in an object header when the evacuator is swapping out an object. This would be akin to setting a "reclaiming" field in the soft pointer when the runtime is reclaiming an object. SOFTRS takes an alternative approach; the reclamation runtime obtains a mutable use guard to block application threads from accessing the data structure during reclamation. This solution was convenient to implement in Rust, but incurs additional overheads because applications are forced to wait for reclamation to finish before accessing an affected data structure.

3.2.4 Soft vector case study

This section walks through the steps it takes to build a soft data structure SOFTRS and demonstrates the API for programming with SOFTRS data structures. Figure 3.3 diagrams an example soft vector implementation.

Data structure logic. The data structure engineer first builds the inner struct, `SoftVecImpl`, that implements the soft data structure logic. This particular data structure engineer borrows from the `std::Vec` implementation and makes it soft by storing individual elements

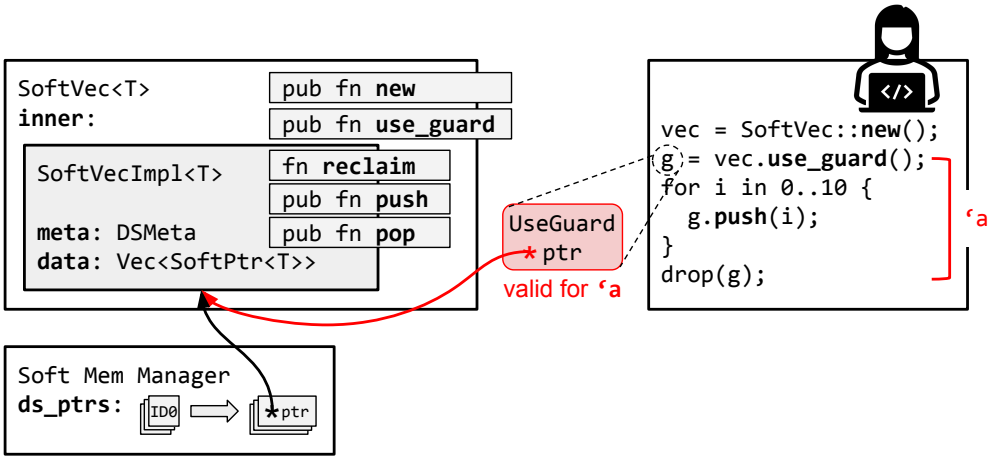


Figure 3.3: On the left, our soft vector design, which is one example of how a data structure engineer can use SoftRS to build soft data structures. `SoftVecImpl::reclaim` is not marked `pub` because it is not used by the application, but it is registered with and called by the soft memory manager. On the right, an example program that uses the soft vector.

behind `SoftPtrs`. The data structure engineer implements the usual vector methods for `SoftVecImpl` by wrapping `std::Vec` methods with a `SoftPtr::deref` or `SoftPtr::deref_mut` call, e.g.

```
data.get(idx)?.deref().
```

The choice to store vector elements in separate soft allocations facilitates reclamation per element. Another valid design is to store the entire vector inside a single soft allocation. If the data structure engineer opts for this design, then they might program the `data` field of the `SoftVecImpl` to be a single `SoftPtr<std::Vec <T> >`. They implement the usual vector methods by doing something like

```
data.deref()?.get(idx).
```

It is not fundamental that the `SoftVecImpl` uses a `std::Vec` at all. The data structure engineer can store any state they want in `SoftVecImpl` in order to implement their desired methods, so long as the state is backed by soft allocations in some way.

Object reclamation logic. Next, the data structure engineer implements `Reclaimable::probe` and `Reclaimable::reclaim` on the `SoftVecImpl`. The SoftRS prototype implements `Reclaimable::probe` by returning elements in ascending order

by index and maintains a probe ID to remember where it left off. The prototype implements `Reclaimable::reclaim` by iteratively probing and dropping elements until the data structure has freed enough bytes to meet the quota specified by the caller (the reclamation runtime).

Data structure shim. Now the data structure engineer is ready to write the outer shim of the soft data structure. They start with the constructor, which registers the soft data structure with the soft memory manager. The memory manager needs to be able to invoke the reclaim logic on the inner struct, so it expects to be passed a pointer to the `SoftVecImpl`[2]. The `status` atomic integer is a crucial component of `DSMeta`. The soft data structure constructor instantiates a new atomic integer. The data structure engineer wraps the atomic integer in an atomic reference counter (`Arc`) so that both application and memory manager threads can reference the same underlying atomic integer. All the code described for setting up the shim is boilerplate, so the data structure engineer can re-use the code that SOFTRS provides in its `SoftVec` implementation.

Use guard. For the final step, the data structure engineer implements `use_guard`. The `SoftVec::use_guard` method returns a `UseGuard<'_, SoftVecImpl>`. Thus, the data structure engineer implements `Guardable::use_guard` for `SoftVecImpl`. Then, they implement a `use_guard` method on the outer shim which calls into the inner `SoftVecImpl::use_guard`. This is the key to accessing the underlying data structure implementation behind a synchronization guard.

3.2.5 Design alternatives

This section explores three design alternatives to the use guard design presented above. Each of the design alternatives is a variation on a *use scope*, which is essentially a per-data structure AIFM dereference scope. The actual synchronization that the use scope and use guard enable are similar; they both synchronize with a data structure-level atomic integer. Their major difference is in the programming interface.

[2]The inner struct should be pinned in memory so that the memory manager will always find it in the same spot, as long as it is allocated; in Rust, the `Pin<T>` type ensures this.

```rust
fn use_guard(mut vec: SoftVec<u32>) -> Result<u32, Error> {
  let mut sum = 0;
  let len = vec.use_guard().unwrap().len();
  let g = vec.use_guard()?;
  for i in 0..len {
    sum += g.get(i)?;
  }
  Ok(sum)
}
```

```rust
fn use_scope_explicit(vec: SoftVec<u32>) -> Result<u32, Error> {
  let mut sum = 0;
  let len = {
    let s = vec.use_scope()?;
    vec.len(&s)
  };
  let s = vec.use_scope()?;
  for i in 0..len {
    sum += vec.get(i, &s)?;
  }
  Ok(sum)
}
```

```rust
fn use_scope_implicit(vec: SoftVec<u32>) -> Result<u32, Error> {
  let mut sum = 0;
  let len = {
    let s = vec.use_scope()?;
    vec.len()
  };
  let s = vec.use_scope()?;
  for i in 0..len {
    sum += vec.get(i)?;
  }
  Ok(sum)
}
```

Figure 3.4: A program that sums all the elements in a soft vector, implemented using the use guard API (top), the use scope API with explicit use scope method arguments (middle), and the use scope API with implicit use scope association via static analysis (bottom).

Design alternative 1. This design proposes a per-DS use scope and requires data structure methods to accept a `UseScope` argument like in AIFM. See Listing 3.4 (b). This design essentially parallels AIFM's design. There are a few challenges to this setup:

1. The soft data structure must check that the use scope passed in is actually a use scope that it created, rather than a different soft DS's use scope.

2. The API for soft data structures looks different from traditional data structures because there is an additional argument for all soft DS methods.

Design alternative 2. To solve both of these challenges, we consider an alternative design where DS methods take care of synchronizing with the memory manager by creating a use scope internally. See Listing 3.4 (c). This moves the burden of synchronization onto the data structure engineer rather than the application programmer. However, it forces the application to synchronize with the reclamation thread upon every data structure access. This may incur considerable performance overhead, and is not very flexible. The programmer should have flexibility to manage the tradeoff between synchronization overhead (constructing many fine-grained use scopes) and reclamation latency (blocking the memory manager from reclaiming with coarse-grained use scopes).

Design alternative 3. The final design alternative solves the enumerated challenges while enabling programmers to create fine- and coarse-grained use scopes. See Listing 3.4 (a). In this design, the soft data structure API does not require a `UseScope` argument for DS method calls, nor does it synchronize internal to the DS. Instead, a static analysis checks whether every data structure access occurs within the lifetime of a valid use scope for that data structure. See Figure 3.4 for a comparison of the API in design alternative 1 vs. 3.

The static analysis categorizes SOFTRS API usage as safe or unsafe and traverses the Rust Mid-level Intermediate Representation (MIR) in order to identify occurrences thereof. The MIR is a control-flow graph intermediate representation in the Rust compiler. When the analysis finds occurrences of unsafe API usage, it emits a compiler warning that points the programmer to the unsafe usage. The analysis is written in a modular fashion so that it can analyze SOFTRS API usage over compositions of functions. Safe usage includes calling a `SoftDS` method while a valid `UseScope` is in scope, and referencing/aliasing soft data

structures in absence of a use scope. Unsafe usage includes calling a `SoftDS` method while a valid `UseScope` is *not* in scope. Section 5.4 discusses the benefits and drawbacks of the use guard and use scope designs.

Chapter 4

Implementation

SOFTRS prototypes a soft memory manager in 1974 lines of C/C++. The soft memory manager is a textbook memory allocator that manages multiple heaps (one heap per-data structure). SOFTRS provides a soft array and linked list in C++, and a soft vector in Rust. We implement the SOFTRS Rust crate in 295 lines of Rust, and the Rust soft vector in 221 lines of code. The Rust soft vector communicates with the underlying soft memory manager via two-way foreign function interfacing (FFI) between Rust and C/C++. The static analysis for the use scope design alternative is implemented in 759 lines of Rust.

Soft array. A single soft allocation backs the soft array. Upon reclamation, the soft array drops the allocation and the array returns to an un-initialized state.

Soft linked list. Individual soft allocations back each linked list node. Upon reclamation, the soft linked list frees elements in order of insertion, and repairs itself by removing reclaimed elements from the list.

Soft vector. Individual soft allocations back each vector element. Upon reclamation, the vector drops elements in order of index, and avoids doing repair by maintaining soft pointers in the vector. Soft pointers to reclaimed elements return `None` upon dereference. By maintaining soft pointer placeholders, the application is explicitly aware of which data was dropped, which can facilitate later recomputation. Alternatively, deleting reclaimed elements from the vector is a reasonable strategy that targets different application semantics.

Chapter 5

Evaluation

In this chapter, we seek to answer the following questions:

1. What is the performance overhead of soft memory programming abstractions in the absence of soft memory reclamation? (§5.1)
2. What is the performance overhead of soft memory programming abstractions in the presence of soft memory reclamation? (§5.2)
3. How does the soft pointer abstraction compare to Rust weak pointers? (§5.3)
4. What are the safety guarantees and ergonomics of the use scope design alternative compared to the use guard design? (§5.4)

5.1 Performance overhead of soft memory abstractions without reclamation

This experiment aims to determine whether there is overhead imposed by the proposed soft memory design in the absence of reclamation. Table 5.1 benchmarks read and write operations, comparing the soft vector described in §3.2.4 to the Rust standard vector. We

	Rust `std::Vec`	`SoftVec`
Read op. (get)	21 ns	18 ns
Write op. (push)	1.0 ns	38 ns

Table 5.1: Performance of read and write operations on a SOFTRS soft vector compared to a Rust standard vector.

calculate the benchmark by timing workloads that read and write between 0-2MB of data to/from a soft vector. There is no reclamation pressure in this benchmark, and the soft vector is backed by the default system allocator in order to rule out discrepancy due to our unoptimized prototype memory allocator. The benchmark does coarse-grained guarding: it uses a single use guard for all the soft vector reads. A good result would show comparable performance between the soft and traditional vectors. Due to the allocation-per-element design of the `SoftVec`, we expect some performance overhead because the memory layout is less compact than in Rust's standard vector, which stores elements in a single allocation. We observe comparable read performance between the vectors. The operations that constitute a `SoftVec` read include:

1. dereferencing the `UseGuard` that wraps the `SoftVecImpl`, which consists of an `UnsafeCell::get` operation; and
2. dereferencing a `SoftPtr`, which consists of unwrapping an `Option`.

We conclude that the enumerated operations do not impose significant overhead. However, the benchmark reveals considerable write overhead in the soft vector compared to the standard vector (38ns vs. 1ns). The operations that constitute a `SoftVec` write include:

1. dereferencing the `UseGuard`, which consists of an `UnsafeCell::get` operation; and
2. putting the data into a `SoftPtr`, which consists of creating a `std::Box` and storing the data inside it.

The read benchmark shows that (1) does not impose considerable overhead. Thus, we conclude that creating a new `SoftPtr` for each vector is costly. This makes sense because each `SoftPtr` construction comes with a heap allocation, while `std::Vec` writes to an existing allocation and occasionally resizes it. This cost highlights a tradeoff that the data structure engineer faces; designing a soft data structure to store elements in separate soft allocations has the advantage of facilitating fine-grained reclamation policies, but at the cost of more expensive write operations. Note also that write operations in typical workloads are much less frequent than read operations.

Coarse-grained vs. fine-grained use guards. We evaluate coarse-grained compared to fine-grained use guard usage. Figure 5.1 runs a write workload with coarse-grained locking

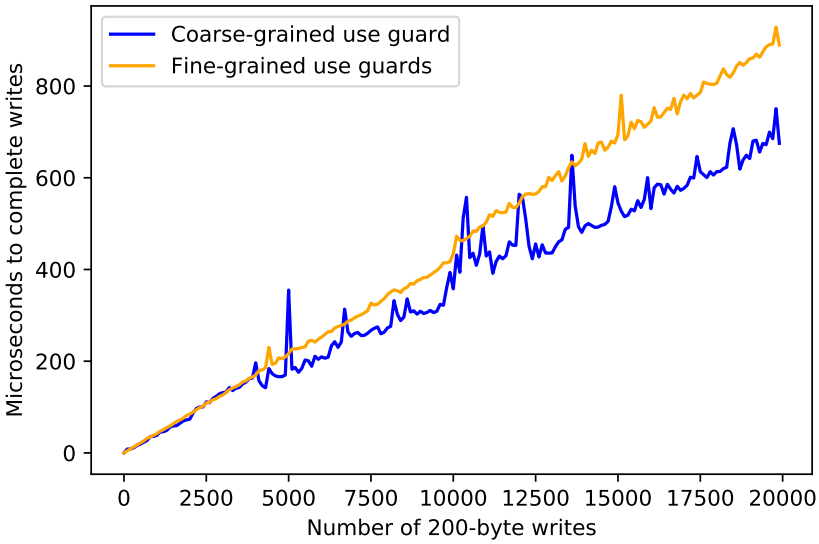


Figure 5.1: A comparison of write workload performance on a `SoftVec` using coarse-grained synchronization (blue) vs. fine-grained synchronization (orange). Fine-grained synchronization imposes some overhead.

(one use guard for all writes) in blue and fine-grained locking (one use guard per write) in orange. There is no reclamation pressure in this benchmark. The soft vectors are backed by the default system allocator in order to rule out overhead due to the memory allocator. The x-axis is the number of 200-byte elements written to the vector, and y-axis is total time in microseconds to write all the elements. We expect worse performance in the fine-grained locking experiment because constructing a use guard has some cost.

At 17,500 writes, a program that opts for fine-grained use guards is 28% slower than a program that opts for coarse-grained use guards. The SOFTRS `UseGuard` constructor does a compare and exchange operation on an atomic integer, which is likely the cause of the overhead. This implementation detail is not fundamental to the use guard design; AIFM dereference scopes do not require an atomic operation in the fast path (no reclamation case) and instead use RCU-like synchronization that avoids atomic operations in the common case. In future work, we will build our use guard abstraction on top of AIFM dereference scopes to eliminate this overhead.

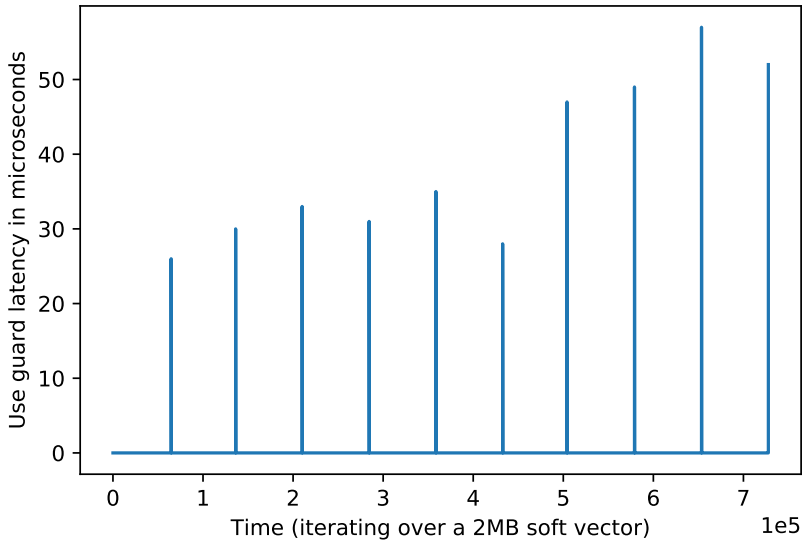


Figure 5.2: Latency in milliseconds for constructing `SoftVec` `UseGuard`s while the application is under reclamation stress. The application experiences some latency due to data structure access during reclamation.

5.2 Performance overhead of soft memory abstractions during reclamation

We now evaluate use guard overhead during reclamation. The memory manager blocks access to a data structure during reclamation on that data structure, so we expect the application to experience latency during reclamation because a data structure may be blocked. To simulate this scenario, we run a read workload on a large soft vector (2MB), and attempt to reclaim 0.2MB from the vector every 100ms. The soft vector is backed by the default system allocator to rule out overheads incurred by slow frees in our prototype memory allocator.

Figure 5.2 presents the latency, in microseconds, of constructing a use guard for each read. When there is no active reclamation, the latency is low (average of 28ns). When there is an active reclamation, the latency is much higher (average of 39ms). The latency comes from the use guard constructor waiting until reclamation has completed. The overhead is not incurred often, since reclamation is infrequent. This overhead is fundamental to the design of the two-way blocking implemented by SoftRS synchronization between the application and reclamation runtime, but would potentially be reduced with a design like AIFM's where application threads never get paused by reclamation.

```rust
fn soft() {
  let mut soft: SoftPtr<u8> = SoftPtr::new(0);

  soft.soft_free();

  if let Ok(r) = soft.deref() {
    println!("Val: {}", r);
  } else {
    println!("Dropped!");
  }
}
```

```rust
fn weak() {
  let strong: Rc<u8> = Rc::new(0);
  let weak: Weak<u8> = Rc::downgrade(&strong);

  drop(strong);

  if let Some(rc) = weak.upgrade() {
    println!("Val: {}", rc);
  } else {
    println!("Dropped!");
  }
}
```

Figure 5.3: Comparing Rust soft pointer and weak pointer APIs. The APIs impose similar unwrapping patterns on pointer dereferences. Both programs print "Dropped!". The key difference is that weak pointers implement sharing semantics via reference counting, and soft pointers do not. Note: the soft pointer is declared as mutable because the program drops its data, which requires a mutable borrow.

5.3 Soft pointer API comparison

Weak pointers are an existing pointer abstraction in Rust for managing state which may or may not be dropped. Figure 5.3 compares the Rust APIs of soft pointers and weak pointers. The key similarity between soft pointers and weak pointers is in how a program obtains the underlying object, if it is available. Soft pointer `deref` and `deref_mut` return a `Result` that contains a pointer to the underlying object, upon success. The weak pointer's `upgrade` returns an `Option` that contains a reference-counted pointer to the underlying object, upon success. Both implementations impose a pattern where programs need to case-check the result of pointer dereferences.

The key difference is that weak pointers are built on top of RC sharing semantics, whereas soft pointers do not support sharing. A Rust program obtains a weak pointer by "downgrading" a reference-counted (RC) pointer. Mutating the object managed by an RC pointer is governed by Rust borrow-checking rules; there can only be either one mutable reference or multiple immutable references. As such, an RC pointer can only mutate its managed object if there are no other RC pointers or weak pointers to the object. By contrast, soft pointers are only shared between the application thread that owns them and the reclamation runtime. The use guard abstraction synchronizes soft pointer sharing with the reclamation runtime, so the soft pointer itself does not reason about sharing semantics.

5.4 Safety and ergonomics of use scope design alternative

Recall design alternative 3 from §3.2.5, which proposes a use scope design similar to AIFM dereference scopes, and prototypes a static analysis to ensure correct use scope usage. This section evaluates this solution against the design goals for synchronizing application soft DS usage with the memory manager that §3.2.3 presented.

Use scopes and use guards both meet goals 2-4 because they provide an abstraction for synchronizing data structure access per-method call, while allowing multiple method calls to be atomic w.r.t. the memory manager. The use scope plus static analysis solution falls short in goal 1, because it does not support all of Rust. The analysis leaves out of scope, e.g., functions external to the module it is run on and programming patterns that move soft data structures or use scopes into structs. This excludes a wide range of programs, including

programs that share data structures between threads by consuming the data structure within a Rust `std::sync::Arc` and `std::sync::Mutex`.

The use scope and use guard designs also differ, albeit slightly, in the programs they can express. Use guard mutability is tied to the mutability of the reference to the data structure it guards. This means the programmer needs to be explicit about constructing a `MutUseGuard` when they need mutability, and a `UseGuard` otherwise. Use scopes, on the other hand, are more flexible because they are not tied to the data structure they are guarding. Use scope programs have the flexibility to create arbitrarily many use scopes, whereas use guard programs are limited to either one mutable guard or multiple immutable guards at any time. This flexibility comes with a tradeoff; the analysis is more difficult because use scopes are not tied to their corresponding data structure by Rust's type system. It takes a lot of effort to write a static analysis that enforces correct use scope usage on an interesting subset of Rust; our prototype took 759 LoC and still lacked support for some common patterns in Rust. Hence, we conclude that the use guard design is more practical and compatible with a wide range of Rust programs.

Chapter 6

Conclusion and Future Work

Programming with soft memory is hard. Dynamically-revocable memory leads to memory and concurrency bugs if soft state access is not handled with extreme care. In prior work [5], developers were expected to manage the memory safety of reclamation by repairing dangling references into reclaimed regions of memory. The reclamation runtime was also vulnerable to racing with application threads when dropping soft memory. This thesis presents abstractions and APIs for programming with soft memory that provide memory safety and concurrency guarantees. Specific contributions are:

1. *soft pointer* and *soft data structure* abstractions,
2. the co-design of a soft memory manager and soft data structures for managing soft state and reclamation in the application,
3. the design of *use guards* to enforce safe concurrent memory reclamation, and
4. SOFTRS, a Rust crate that prototypes use guard, soft pointer, and soft data structure abstractions to provide ergonomic, memory-safe, and concurrency-safe programming with soft memory.

Performance evaluation demonstrates that the soft pointer and use guard abstractions prototyped in this work add some overhead to soft data structure operations. Future work aims to replace the textbook soft memory manager implemented for this work by a more sophisticated runtime built on top of AIFM. AIFM dereference scopes will back SOFTRS use guards in order to eliminate the performance overhead that SOFTRS use guards currently impose due two-way synchronization.

Bibliography

[1] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. “On the Diversity of Cluster Workloads and Its Impact on Research Results”. In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. Boston, Massachusetts, USA, 2018, pages 533–546.

[2] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 2017, pages 153–167.

[3] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-Efficient and QoS-Aware Cluster Management”. In: *SIGPLAN Notices* 49.4 (Feb. 2014), pages 127–144.

[4] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. “Garbage-First Garbage Collection”. In: *Proceedings of the 4th International Symposium on Memory Management*. Vancouver, BC, Canada, 2004, pages 37–48.

[5] Megan Frisella, Shirley Loayza-Sanchez, and Malte Schwarzkopf. “Towards Increased Datacenter Efficiency with Soft Memory”. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. Providence, RI, USA, June 2023, pages 127–134.

[6] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. “Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines”. In: *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys)*. Volume 4. 2022, pages 33–51.

[7] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. "Software-Defined Far Memory in Warehouse-Scale Computers". In: Providence, Rhode Island, USA, 2019, pages 317–330.

[8] Per-Åke Larson and Murali Krishnan. "Memory Allocation for Long-Running Server Applications". In: *Proceedings of the 1st International Symposium on Memory Management (ISMM)*. Vancouver, British Columbia, Canada, 1998, pages 176–185.

[9] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. "Imbalance in the cloud: An analysis on Alibaba cluster trace". In: *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*. 2017, pages 2884–2892.

[10] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. "Prioritized Garbage Collection: Explicit GC Support for Software Caches". In: *SIGPLAN Notices* 51.10 (Oct. 2016), pages 695–710.

[11] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. "Harvesting Idle Memory for Application-Managed Soft State with Midas". In: *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA, USA, Apr. 2024, TODO.

[12] Redis Ltd. *Redis*. URL: `https://redis.io/` (visited on 05/23/2023).

[13] Zhenyuan Ruan, Malte Schwarzkopf, Marcos Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Canada, Nov. 2020, pages 315–332.

[14] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. "Autopilot: Workload Autoscaling at Google". In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Heraklion, Greece, Apr. 2020.

[15] Shirley. *Shirley*.

[16] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. “Borg: the Next Generation”. In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Heraklion, Crete, Apr. 2020.

[17] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[18] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. “Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly”. In: *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*. San Francisco, California, 2022, pages 210–225.