Rust as a Language for High Performance GC Implementation

Yi Lin[†] Stephen M. Blackburn[†] Antony L. Hosking[†*$] Michael Norrish[*]

[†]Australian National University [*]Data61, Australia [$]Purdue University, USA

[†]{yi.lin,steve.blackburn,antony.hosking}@anu.edu.au [*]{antony.hosking,michael.norrish}@data61.csiro.au

Abstract

High performance garbage collectors build upon performance-critical low-level code, typically exhibit multiple levels of concurrency, and are prone to subtle bugs. Implementing, debugging and maintaining such collectors can therefore be extremely challenging. The choice of implementation language is a crucial consideration when building a collector. Typically, the drive for performance and the need for efficient support of low-level memory operations leads to the use of low-level languages like C or C++, which offer little by way of safety and software engineering benefits. This risks undermining the robustness and flexibility of the collector design. Rust's ownership model, lifetime specification, and reference borrowing deliver safety guarantees through a powerful static checker with little runtime overhead. These features make Rust a compelling candidate for a collector implementation language, but they come with restrictions that threaten expressiveness and efficiency.

We describe our experience implementing an Immix garbage collector in Rust and C. We discuss the benefits of Rust, the obstacles encountered, and how we overcame them. We show that our Immix implementation has almost identical performance on micro benchmarks, compared to its implementation in C, and outperforms the popular BDW collector on the gcbench micro benchmark. We find that Rust's safety features do not create significant barriers to implementing a high performance collector. Though memory managers are usually considered low-level, our high performance implementation relies on very little unsafe code, with the vast majority of the implementation benefiting from Rust's safety. We see our experience as a compelling proof-of-concept of Rust as an implementation language for high performance garbage collection.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Run-time environments

General Terms Experimentation, Languages, Performance, Measurement

Keywords memory management, garbage collection, Rust

ISMM'16, June 14, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4317-6/16/06...$15.00
http://dx.doi.org/10.1145/2926697.2926707

1. Introduction

A fast yet robust garbage collector (GC) is the key to garbage collected language runtimes. However, implementing such a GC is not easy. First, a collector must manipulate raw memory, depending on carefully optimized code to do so, making it naturally prone to memory bugs. Second, high performance GCs are rich in concurrency, typically featuring thread parallelism, including thread-local allocation, parallel tracing, and possibly mutator concurrency, making it prone to race conditions and extremely time consuming bugs.

What makes the situation worse is that the implementation language usually does not provide help in terms of memory safety and thread safety. The imperative of performance encourages the use of languages such as C and C++ in collector implementations. But their weak type system, lack of memory safety, and lack of integrated support for concurrency [5] throws memory and thread safety squarely back into the hands of developers.

Poor software engineering leads not only to hard-to-find bugs and performance pitfalls, but decreases reuse, inhibiting progress by thwarting creativity and innovation. Unfortunately, programming languages often place positive traits such as abstraction and safety at odds with performance. However, we are encouraged: first, by prior work [10, 11] that shows that in an implementation such as a garbage collector, low-level code is the exception, not the rule; and second by the Rust programming language, which rather boldly describes itself as *a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety* [14].

We evaluate the software engineering of a high performance collector, and our experience confirms the prior work. In particular, we confirm that: (*i*) performance-critical code is very limited in its scope, (*ii*) memory-unsafe code is very limited in its scope, and (*iii*) language-supported, high performance thread-safe data structures are fundamental to collector implementation. For these reasons, a well-chosen language may greatly benefit collector implementations *without* compromising performance.

Our prior experience in collector implementation includes both C/C++ and high level languages. This, and the emergence of Rust led us to evaluate it as a language for high performance collector implementation. Rust is type, memory, and thread safe; all safety features that we believe will help in delivering a robust collector. Rust also permits unsafe operations (and inline assembly[1]) in unsafe blocks, allowing us to access bare memory, and to fine-tune performance on fast paths when needed. Furthermore, Rust uses a powerful compile-time safety checker to shift as much as possible of the safety burden to compile time, avoiding runtime overheads where possible. The checker is based on Rust's model of object ownership, lifetimes, and reference borrowing. The model elimi-

[1] Inline assembly is currently only available in nightly releases, and not used in this work.

nates the possibility of dangling pointers and races, and ensures memory safety. However, this model also restricts the language's expressiveness. Mapping semantics required of a collector implementation into Rust's language model was the major challenge that we faced during the work. Ultimately, we found that not only was this achievable, but that we could do so with no discernable overhead compared to an implementation of the same collector written in C.

The principal contributions of this paper are: (*i*) a discussion of the challenges of implementing a high-performance collector in a type, memory and thread-safe language, (*ii*) a discussion of the semantic impedance between Rust's language model and the semantic requirements of a collector implementation, (*iii*) a performance comparison evaluating Rust and C implementations of the same high performance collector design, and (*iv*) a comparison with the popular Boehm-Demers-Weiser (BDW) collector implemented in C [7, 8].

We start by describing how we are able to *use* Rust's particular language features in our high performance collector implementation. We then discuss cases where we found it necessary to *abuse* Rust's unsafe escape hatches, avoiding its restrictive semantics, and ensuring the performance and semantics we required. Finally, we conduct a head-to-head performance comparison between our collector implementation in Rust and a mostly identical implementation in C to demonstrate that if used properly, the safety and abstraction cost from Rust is minimal, compared to an unsafe language such as C. Also we show that both implementations outperform BDW, a production-level GC. This suggests that it is possible to have a fast GC implementation that also benefits from its implementation language's safety guarantees.

2. Related Work

There has been much previous work addressing the implementation of efficient collectors in safe languages [1, 2, 4, 11, 13, 18]. The advantages of using safe languages demonstrated by these projects motivate our work. However, our use of Rust takes some further steps: (*i*) Rust guarantees type, memory, and thread safety. Previous work uses languages that make weaker safety guarantees such as type safety and weaken memory safety (by disabling GC) and leave thread safety exposed. (*ii*) Rust has considerably more restricted semantics and expressiveness, since it performs most safety checks at compile time. This constrains GC implementation, and invites the question of whether implementing a high performance GC in Rust is even viable. (*iii*) Rust is an off-the-shelf language. We take the challenge to map GC features efficiently to the language semantics and we use Rust without changes to the base language. Previous work changed or augmented the semantics of the implementation languages to favor GC implementation [11, 18].

There are also projects[2] that implement collectors in Rust *for Rust*. Though these projects use Rust as the implementation language, their focus is in introducing GC as a language feature to Rust. Their implementations do not reflect the state of the art of GC, and they do not report performance: the delivery of both significantly adds difficulty in a collector implemented in Rust. Our work takes on the challenge of achieving both, to deliver a high-performance advanced collector implementation.

[2] A reference counted type with cycle collection for Rust: `https://github.com/fitzgen/bacon-rajan-cc`; a simple tracing (mark and sweep) garbage collector for Rust: `https://github.com/Manishearth/rust-gc`.

3. Rust Background

We now introduce some of the key concepts in Rust used in this paper.

Ownership. In Rust, variable binding grants a variable unique ownership of the value it is bound to. This is similar to C++11's `std::unique_ptr`, but it is mandatory for Rust as it is the key concept upon which Rust's memory safety is built. Unbound variables are not allowed, and rebinding involves `move` semantics, which transfers the ownership of the object to the new variable while invalidating the old one.[3] When a variable goes out of scope, the associated ownership expires and resources are reclaimed.

References. Acquiring the ownership of an object for accessing is expensive because the compiler must emit extra code for its proper destruction on expiry. A lightweight approach is to instead *borrow* references to access the object. Rust allows one or more co-existing *immutable* references to an object or exactly one *mutable* reference with no immutable references. The ownership of an object cannot be moved when it is borrowed. This rule eliminates data races, as mutable (write) and immutable (read) references are made mutually exclusive by the rule. More interestingly, this mutual exclusion is guaranteed mostly at compile time by Rust's borrow checker.

Data Guarantees (Wrapper Types). An important feature of Rust is that the language and its library provide various wrapper types with different guarantees and tradeoffs. For example, plain references such as `&T` and `&mut T` statically guarantee a read-write 'lock' for single-threaded code with no runtime overhead, while `RefCell<T>` offers the same guarantee at the cost of runtime checks but is useful when the program has complicated data flow. Our implementation uses the following wrapper types as described in later sections of the paper. `Box<T>` represents a pointer which uniquely owns a piece of heap-allocated data. `Arc<T>` is another frequently used wrapper which provides an atomically reference-counted shared pointer to data of type `T`, and guarantees the data stays accessible until every `Arc<T>` to it goes out of scope (i.e., the count drops to zero). A common idiom to share mutable data among threads is `Arc<Mutex<T>>` which provides a mutual exclusive lock for type `T`, and allows sharing the mutex lock across threads.

Unsafe. Rust provides a safe world where there are no data races and no memory faults. However, the semantics in safe Rust are in some cases either too restrictive or too expensive. Rust allows unsafe code, such as raw pointers (e.g., `*mut T`), forcefully allowing sharing data across threads (e.g., `unsafe impl Sync for T{}`), intrinsic functions (e.g., `mem::transmute()` for bit casting without check), and external functions from other languages (e.g., `libc::malloc()`). `Unsafe` is a powerful weapon for programmers to wield at their own risk. Rust alerts programmers by requiring unsafe code to be contained within a block that is marked unsafe, or exposed to the caller by marking the containing function as itself unsafe.

4. Using Rust

We now describe key aspects of how we *use* Rust's language features to construct a high performance garbage collector. In Section 5 we discuss how we found it necessary to *abuse* Rust, selectively bypassing its restrictive semantics to achieve the performance and semantics necessary for a high performance collector.

For the sake of this proof of concept implementation, we implement the Immix garbage collector [3]. We use it because it: (*i*) is

[3] Rebinding of `Copy` types, such as primitives, makes a copy of the value for the new variable instead of moving ownerships; the old variable remains valid.

```
#[derive(Copy, Clone, Eq, Hash)]
pub struct Address(usize);

impl Address {
  // address arithmetic
  #[inline(always)]
  pub fn plus(&self, bytes: usize) -> Address {
    Address(self.0 + bytes)
  }

  // dereference a pointer
  #[inline(always)]
  pub unsafe fn load<T: Copy> (&self) -> T {
    *(self.0 as *mut T)
  }

  // bit casting
  #[inline(always)]
  pub fn from_ptr<T> (ptr: *const T) -> Address {
    unsafe {mem::transmute(ptr)}
  }

  // cons a null
  #[inline(always)]
  pub unsafe fn zero () -> Address {
    Address(0)
  }

  ...
}
```

Figure 1. An excerpt of our `Address` type, showing some of its safe and unsafe methods.

a high-performance garbage collector, (*ii*) has interesting characteristics beyond a simple mark-sweep or copying collector, and (*iii*) has a well-documented publicly available reference implementation. Our implementation supports parallel (thread-local) allocation and parallel collection. We have not yet implemented opportunistic compaction, nor generational or reference counting variants [16]. We do not limit our discussion to the Immix algorithm, but rather we consider Rust's broader suitability as a GC implementation language.

Our implementation follows three key principles: (*i*) the collector must be high performance, with all performance-critical code closely scrutinized and optimized, (*ii*) we do not use *unsafe* code unless absolutely unavoidable, (*iii*) we do not modify the Rust language in any way.

The remainder of this section considers four distinct elements of our experience of Rust as a GC implementation language: (*i*) the encapsulation of `Address` and `ObjectReference` types, (*ii*) managing ownership of address blocks, (*iii*) managing global ownership of thread-local allocations, and (*iv*) utilizing Rust libraries to support efficient parallel collection.

4.1 Encapsulating Address Types

Memory managers manipulate raw memory, conjuring language-level objects from raw memory. Experience shows the importance of abstracting over both arbitary raw addresses and references to user-level objects [4, 11]. Such abstraction offers type safety and disambiguation with respect to implementation-language (Rust) references. Among the alternatives, raw pointers can be misleading and dereferencing an untyped arbitrary pointer may yield unexpected data, while using integers for addresses implies arbitrary type casting between pointers and integers, which is dangerous.

Abstracting address types also allows us to distinguish addresses from object references for the sake of software engineering and safety. Addresses and object references are two distinct abstract concepts in GC implementations: an address represents an arbitrary location in the memory space managed by the GC and address arithmetic is allowed (and necessary) on the address type, while an object reference maps directly to a language-level object, pointing to a piece of raw memory that lays out an object and that assumes some associated language-level per-object meta data (such as an object header, dispatch table, etc). Converting an object reference to an address is always valid, while converting an address to an object reference is unsafe.

Abstracting and differentiating addresses is important, but since addresses are used pervasively in a GC implementation, the abstraction must be efficient, both in space and time. We use a single-field *tuple struct* to provide `Address` and `ObjectReference`, abstracting over Rust's word-width integer `usize` to express addresses, as shown in Figure 1. This approach disables the operations on the inner type, and allows a new set of operations on the abstract type. This abstraction adds no overhead in type size, and the static invocation of its methods can be further marked as `#[inline(always)]` to remove any call overhead. So while the types have the appearance of being boxed, they are materialized as unboxed values with zero space and time overheads compared to an untyped alternative, whilst providing the benefits of strong typing and encapsulation.

We restrict the creation of `Address`s to be either from raw pointers, which may be acquired from `mmap` and `malloc`, or derived from an existing `Address`. `Address` creation from arbitrary integers is forbidden, with the single exception of the constant `Address::zero()`. This serves as an initial value for some fields of type `Address` within other `structs`, since Rust does not allow `structs` with uninitialized fields. A safer alternative in Rust is to use `Option<Address>` initialized as `None` to indicate that there is no valid value. However, this adds an additional conditional and a few run-time checks to extract the actual address value in the performance-critical path of allocation, which adds around 4% performance overhead. We deem this tradeoff not to be worthwhile given the paramount importance of the allocation fast path and the infrequency with which this idiom arises within the GC implementation. Thus we choose to allow `Address::zero()` but mark it as **`unsafe`** so that implementors are explicitly tasked with the burden of ensuring safety.

Our implementation of `ObjectReference` follows a very similar pattern. The `ObjectReference` type provides access to per-object memory manager metadata (such as mark-bits/-bytes). An `Address` cannot be safely cast to an `ObjectReference`; the allocator code responsible for creating objects must do so via an **`unsafe`** cast, explicitly imposing the burden of correctness for fabricating objects onto the implementer of the allocator. An `ObjectReference` can always be cast to an `Address`.

4.2 Ownership of Memory Blocks

Thread-local allocation is an essential element of high performance memory management for multithreaded languages. The widely used approach is to maintain a global pool of raw memory regions from which thread-local allocators take memory as they need it, and to which thread-local collectors push memory as they recover it [1]. This design means that the common case for allocation involves no synchronization, whilst still facilitating sharing of a global memory resource. The memory manager must ensure that it correctly manages raw memory blocks to thread-local allocators, ensuring exclusive ownership of any given raw block. Note, however that once objects are fabricated from these raw blocks, they may (according to the implemented language's semantics) be shared among all threads. Furthermore, at collection time a parallel collector may have no concept of memory ownership, with each thread marking objects at any place in the heap, regardless of any

```
// thread local allcator
pub struct AllocatorLocal {
  ...
  space: Arc<Space>,

  // allocator may own a block it can allocate into
  // Option suggests the possibily of being None,
  // which leads to the slow path to acquire a block
  block: Option<Box<Block>>
}

// global space, shared among multiple allocators
pub struct Space {
  ...
  usable_blocks : Mutex<LinkedList<Box<Block>>>,
  used_blocks   : Mutex<LinkedList<Box<Block>>>
}

impl AllocatorLocal {
  fn alloc_from_global (&mut self,
       size: usize, align: usize) -> Address {
    // allocator will return the ownership of
    // current block (if any) to global space
    if block.is_some() {
      let block = self.block.take().unwrap();
      self.space.return_used_block(block);
    }

    // keep trying acquiring a new block from space
    loop {
      let new_block
        = self.space.get_next_usable_block();
      ...
    }
  }
}
```

Figure 2. Ownership transfer between the global memory pool and a thread local allocator.

notion of ownership over the object's containing block. We make this guarantee by using Rust's ownership semantics.

Ownership is the key part of Rust's approach to delivering both performance *and* safety. We map the ownership semantics to this scenario to make the guarantee that each block managed by our GC is in a coherent state among *usable*, *used*, or *being allocated into by a unique thread*. To achieve this, we create `Block` objects, each of which uniquely represents the memory range of the block and its meta data. The global memory pool *owns* the `Blocks`, and arranges them into a list of usable `Blocks` and a list of used `Blocks` (Figure 2). Whenever an allocator attempts to allocate, it acquires the ownership from the usable `Block` list, gets the memory address and allocation context from the `Block`, then allocates into the corresponding memory. When the thread-local memory block is full, the `Block` is returned to the global *used* list, and waits there for collection. The Rust's ownership model ensures that allocation will not happen unless the allocator *owns* the `Block`, and, further every `Block` is guaranteed to be in one of the three states: (*i*) owned by the global space as a usable `Block`, (*ii*) owned by a single allocator, and being allocated into, (*iii*) owned by the global space as a used `Block`. During collection, the collector scavenges memory among *used* `Block`s, and classifies them as *usable* for further allocation if they are free.

4.3 Globally Accessible Per-Thread State

A thread-local allocator avoids costly synchronization on the allocation fast path because mutual exclusion among allocators is ensured. This is something that Rust's ownership model ensures can be implemented very efficiently. However parts of the thread-local allocator data structure may be *shared* at collector time (for example, allocators might be told to yield by a collector thread via this data structure). Rust will not allow for a mixed ownership model like this except by making the data structure shared, which means that all accesses are vectored through a synchronized wrapper type, ensuring that every allocation is synchronized, thus defeating the very purpose of the thread-local allocator.

We deal with this by breaking the per-thread `Allocator` into two parts, a thread-local part and a global part, as shown in FIgure 3. The thread-local part includes the data that is accessible strictly within current thread and an `Arc` reference to its global part. All shared data goes to the global part (with a safe wrapper if mutability is required). This allows efficient access to thread local data, while allowing shared per-thread data to be accessed globally.

```
pub struct AllocatorLocal {
  // fields that are strictly thread local
  ...

  // fields that are logically per allocator
  // but need to be accessed globally
  global: Arc<AllocatorGlobal>
}

pub struct AllocatorGlobal {
  // any field in this struct that requires
  // mutability needs to be either be atomic
  // or lock-guarded
  ...
}

// statics that involve dynamic allocation
lazy_static! {
  pub static ref ALLOCATORS
    : Vec<Arc<AllocatorGlobal>> = vec![];
}
```

Figure 3. Separating a per-thread `Allocator` into two parts. The local part is strictly thread local, while the global part can be accessed globally.

4.4 Library-Supported Parallelism

Parallelism is essential to high performance collector implementations. Aside from the design of the high level algorithm, the efficiency of a collector depends critically on the implementation of fast, correct, parallel work queues [12]. In a marking collector such as Immix and most tracing collectors, a work queue (or 'mark stack') is used to manage pending work. When a thread finds new marking work, it adds a reference to the object to the work queue, and when a thread needs work, it takes it from the work queue. Ensuring efficient and correct operation of a parallel work queue is a challenging aspect of high performance collector implementation [4, 12].

We were pleased to find that Rust provides a rich selection of safe abstractions that perform well as part of its standard and external libraries (known as *crates* in Rust parlance). Using an external crate is as simple as adding a dependency in the project configuration, which greatly benefits code reusability. The use of standard libraries is deeply problematic when using a modified or restricted language subset, as has been commonly used in the past [1, 2, 4, 11, 13, 18]. For example, if using a restricted subset of Java, one must be able to guarantee that any library used does not violate the preconditions of the subset, which may be extremely restrictive (such using only the fully static subset of the language, excluding allocation and dynamic dispatch). Consequently, standard libraries are off limits when using restricted Java to build a garbage collector.

We utilize two crates in Rust, `std::sync::mpsc`, which provides a multiple-producers single-consumer FIFO queue, and `crossbeam::sync::chase_lev`[4], which is a lock-free Chase-Lev work stealing deque that allows multiple stealers and one single worker [9]. We use these two abstraction types as the backbone of our parallel collector with a modest amount of additional code to integrate them.

Our parallel collector starts single-threaded, to work on a local queue of GC roots; if the length of the local queue exceeds a certain threshold, the collector turns into a controller and launches multiple stealer collectors. The controller creates an asynchronous `mpsc channel` and a shared `deque`; it keeps the receiver end for the channel, and the worker for the deque. The sender portion and stealer portion are `cloned` and `moved` to each stealer collector. The controller is responsible for receiving object references from stealer threads and pushing them onto the shared deque, while the stealers steal work (`ObjectReferences`) from the deque, do marking and tracing on them, and then either push the references that need to be traced to their local queue for thread-local tracing or, when the local queue exceeds a threshold, send the references back to the controller where the references will be pushed to the global deque. When the local queue is not empty, the stealer prioritizes getting work from the local queue; it only steals when the local queue is empty.

Using those existing abstract types makes our implementation straightforward, performant and robust: our parallel marking and tracing features only 130 LOC (shown as Appendix B) while there are over one thousand lines of well tested code from the libraries to support our implementation. We measure and discuss the performance of our parallel marking and tracing implementation in Section 6.

5. Abusing Rust

In the previous section, we described how Rust's semantics affect the implementation of a high performance garbage collector. Though Rust's model is sometimes restrictive, in most cases we were able to fairly straightforwardly adapt the collector design to take full advantage of Rust's safety and performance. However, there are a few places where we found that Rust's safety model was too restrictive to express the necessary semantics efficiently, and thus found ourselves having to dive into **`unsafe`** code, where the programmer bears responsibility for safety, rather than Rust and its compiler.

5.1 Shared Bit and Byte Maps

Garbage collectors often implement bit maps and byte maps to represent collection state, mapping addresses to table offsets. Examples include card tables (which remember modified memory regions), and mark tables (which remember marked objects). To implement these correctly and efficiently, they are frequently byte maps (allowing atomic update). Semantics may include, for example, multiple writers but idempotent transitions: during the mark phase, the writers may only set the mark byte (not clear it). For example, an object map indicates the start of objects: in a heap where every object is 8-byte aligned, every bit in such a bitmap can represent whether an 8-byte aligned address is the start of an object. In Immix, a *line mark table* is used to represent the state of every line in the memory space — an unsigned byte (`u8`) for every 256-bytes of allocated memory.

During *allocation*, the line mark table may be accessed by multiple allocator threads, *exclusively* for the addresses that they are allocating into. Since every allocator allocates into a non-overlapping memory block, they access non-overlapping elements in the line mark table. However, in Rust, if we were to create the line mark table as a Rust array of `u8`, Rust would forbid concurrent writing into the array. Ways to bypass this within the confines of Rust are to either break the table down into smaller tables, or to use a coarse lock on the large table, both of which are impractical.

On the other hand, during *collection*, the mutual exclusion enjoyed by the allocator does not exist: two collector threads may race to mark adjacent lines, or even the same line. The algorithm ensures that such races are benign, as both can only set the line to 'live' and storing to a byte is atomic on the target architecture. However, in Rust, it is strictly forbidden to modify a shared object's non-atomic fields without going through a lock. We are unaware of a reliable solution to this in stable Rust releases, which do not support an `AtomicU8` type, nor intrinsic atomic operations as in the nightly releases.

Instead, we use the work-around shown in Figure 4. We generalize the line mark table as an `AddressMapTable`. We wrap the necessary unsafety into the `AddressMapTable` implementation which almost entirely comprises safe code. We acknowledge also that for proper atomicity of the byte store (with respect to both the compiler and target) we should also be using an atomic operation to store the value rather than a normal assignment. Here we rely on the Rust compiler to generate an x86 byte store which is atomic. Otherwise, there are reasonable compiler optimizations that could defeat the correctness of our code [6]. What is more, the target architecture might not have an atomic byte store operation. The availability of LLVM intrinsics in the non-stable nightly Rust releases would allow us to use a relaxed atomic store to achieve the correct code, as shown in the comment. This exposes a shortcoming in Rust's current atomic types where we desire an `AtomicU8` type, along the lines of the existing `AtomicUsize`. This need is reflected in the recently accepted Rust RFC #1543: *'Add more integer atomic types'* [15].

```
1  pub struct AddressMapTable {
2    start : Address,
3    end   : Address,
4
5    len : usize,
6    ptr : *mut u8
7  }
8  // allow sharing of AddressMapTable across threads
9  unsafe impl Sync for AddressMapTable {};
10 unsafe impl Send for AddressMapTable {};
11
12 impl AddressMapTable {
13   pub unsafe fn set (&self, addr: Address, value: u8)
14   {
15     let index = addr.diff(self.start) >> LOG_PTR_SIZE;
16     unsafe {
17       let ptr = self.ptr.offset(index);
18       // intrinsics::atomic_store_relaxed(ptr, value);
19       *ptr = value;
20     }
21   }
22 }
```

Figure 4. Our `AddressMapTable` allows concurrent access with unsafe methods. The user of this data structure is responsible for ensuring that it is used safely.

6. Evaluation

The two primary objectives of our proof-of-concept implementation were to establish: (*i*) to what extent we are able to exploit Rust's safety (hopefully minimizing the amount of unsafe code), and (*ii*) the impact of Rust on performance. In this section, we dis-

[4] `https://github.com/aturon/crossbeam`

cuss our evaluation of our proof-of-concept collector, focusing on these concerns.

6.1 Safe Code

Our first major challenge was to map our collector design into the Rust language. As we discuss in Sections 4 and 5, for the main part, the collector implementation can be expressed entirely in safe Rust code. As shown in Table 1, 96 % of 1449 lines of the code are safe. This suggests that though GC is usually considered to be a low-level module that operates heavily on raw memory, the vast majority of its code can in fact be safe, and can benefit from the implementation language if that language offers safety.

Language	Files	Lines of Code	Unsafe LOC (%)
Rust	13	1449	58 (4.0%)

Table 1. Unsafe code is minimal in our implementation.

The unsafe code amounts to 4.0 % and mainly comes from just two sources. The first is where **`unsafe`** is required for access to raw memory, such as dereferencing raw pointers during tracing, manipulating object headers, zeroing memory, etc. This is unavoidable in memory manager implementations. Our experience shows that through proper abstraction, the unsafe code for accessing raw memory can be restricted to a small proportion of the code base. The second source of unsafety is due to Rust's restricted semantics. Rust trades expressiveness for the possibility of statically enforcing safety. Section 4 shows that for most of the cases, we are able to adapt our collector implementation to Rust's constraints. In the exceptional case described in Section 5 where Rust stands in our way, we are able to encapsulate it in a small amount of unsafe code.

Our experience demonstrates that a garbage collector can be implemented in a safe language such as Rust with very little unsafe code. Furthermore, we can report that, subjectively, the discipline imposed upon us by Rust was a real asset when we went about this non-trivial systems programming task with its acute *performance and correctness* focus.

6.2 Performance

Our second challenge was to deliver on our goal of high performance. Since at this stage we are implementing a standalone garbage collector, not yet integrated into a larger language runtime, it is hard to provide performance evaluation via comprehensive benchmarks; instead we use micro benchmarks to evaluate the collector. We are not interested in evaluating garbage collection algorithms per se (we take an existing algorithm off the shelf). Rather, we simply wish to provide proof of concept for an implementation of a high performance collector in Rust and show that it performs well in comparison to an equivalent collector written in C. To this end, we are particularly interested in the performance of critical hot paths, both for collection and allocation since the performance of the algorithm itself is already established [3], and our prior experience demonstrates the overwhelming criticality of these hot paths to performance.

6.2.1 Micro Benchmarks

To evaluate the performance of our implementation in Rust, we also implemented the collector in C, following the same Immix algorithm. We did not try to make the two implementations exactly identical, but used the features of the available language in a naturally fluent way. For most scenarios described in Section 4 and 5, it is either unnecessary or simply impossible to write C code the same way as Rust code. The C implementation allows us to set a baseline for performance in an implementation language that is known to be efficient and allows a head-to-head comparison for Rust performance. We took particular care to ensure that the performance-critical hot paths were implemented efficiently in the respective languages.

We chose three performance-critical paths of the collector to run single-threaded as micro benchmarks: allocation, object marking, and object tracing. Each micro benchmark allocates 50 million objects of 24 bytes each, which takes 1200 MB of heap memory; we use a 2000 MB memory for each run so that the GC will not collect spontaneously (we control when tracing and collection occurs in the respective micro benchmarks). In each micro benchmark, we measure the time spent on allocating, marking, and tracing the 50 million objects. We use rustc 1.6.0 stable release for Rust, and clang 3.7 for C, both of which use LLVM 3.7 as backend. We run each implementation with 20 invocations on a 22 nm Intel Core i7 4770 processor (Haswell, 3.4 GHz) with Linux kernel version 3.17.0. The results appear in Table 2.

	C	Rust (% to C)
`alloc`	370 ± 0.1 ms	374 ± 2.9 ms (101%)
`mark`	63.7 ± 0.5 ms	64.0 ± 0.7 ms (100%)
`trace`	267 ± 2.1 ms	270 ± 1.0 ms (101%)

Table 2. Average execution time with 95% confidence interval for micro benchmarks of performance critical paths in GC. Our implementation in Rust performs the same as the C implementation.

From the micro benchmark results, we can see that with careful performance tuning, the Rust implementation matches the performance of our C implementation across all the three micro benchmarks (identifying most performance critical paths in a collector implementation). In our initial implementation (without fine performance tuning), Rust was within 10% slowdown of C on micro benchmarks. We found it encouraging, considering: (*i*) our source code in Rust offers stronger abstraction than C, a low-level imperative language, and (*ii*) the source code enjoys Rust's safety guarantees. We then fine-tuned the performance, examining assembly code generated for each implementation, where necessary altering fast path code to avoid idioms with negative performance implications. Our micro benchmarks have tiny kernels and are memory intensive, and one instruction may affect results. We found although rustc is agressive it is quite predicatable making it not difficult to generate highly performant code. Lack of tools for finer control on the generated code such as branch hints may be a drawback of the Rust compiler, but did not hinder performance in the micro benchmarks. Appendix A shows Rust code for the allocation fast path, and Appendix B shows the code for mark and trace fast path in a parallel collector.

6.2.2 Library-based Parallel Mark and Trace

We evaluate the performance scaling of parallel GC in our implementation. As described in Section 4.4, we quickly implemented the parallel mark and trace collector by completely basing its parallelism on existing Rust crates: `std::sync::mpsc` and `crossbeam:: sync::chase_lev`. They provide all the concurrency as the backbone of our parallel collector. This implementation approach is high-level and productive, but as we shall show, it is also performant.

We use a micro benchmark to trace 50 quad trees of depth ten to allow parallel collectors to build a reasonable local work queue for thread-local tracing and to push excessive references to the global deque. We use a large heap to avoid spontaneous collections during the tree allocation. We run this with twenty invocations on the same i7 Haswell machine, using from zero to seven GC worker threads, and measure the tracing time. Note that zero means no

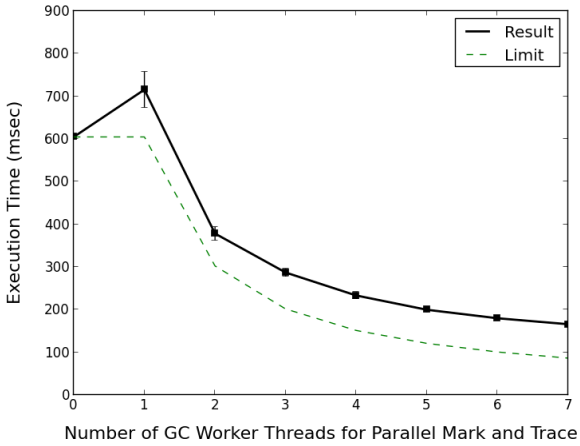

Figure 5. Performance scaling for our fast implemented libraries-based parallel mark and trace collector.

parallel GC while one to seven reflect the number of GC worker threads (with one additional controller thread). Seven workers with one controller is the full capacity of our machine (four cores with eight SMT threads). Figure 5 shows the results along with a line indicating hypothetical perfect scaling (which assumes workloads are equally divided among threads with no overhead compared to a single-threaded collector).

With parallel GC disabled, single-threaded marking and tracing takes 605 ms, while with one worker thread, the benchmark takes 716 ms. The overhead is due to sending object references back to the global deque through an asynchronous channel, and stealing references from the shared deque when the local work queue is empty. With two and three worker threads, the scaling is satisfactory, with execution times of 378 ms and 287 ms (52.8 % and 40.0 % compared with one worker). When the number of worker threads exceed four, the scaling starts to fall off slightly. With seven worker threads, the execution time is 166 ms, which is 23.2 % of one worker thread. The performance degradation is most likely from two sources: (*i*) GC workers start to share resources from the same core after every core hosts one worker, (*ii*) having one central controller thread to receive object references and push them to the global deque starts to be a performance bottleneck. These results could undoubtedly be improved with further tuning. However, as it is one tricky part of the implementation, and we worked towards a working (and safe) implementation with limited time and limited lines of code by using existing libraries, the approach itself is interesting and demonstrates the performance tradeoff due to improved productivity. We believe the performance scaling is good, and that having a language that provides higher level abstractions can benefit a parallel GC implementation (and possibly a concurrent GC) greatly.

6.2.3 GCBench

We compare our Immix implementation in Rust with the BDW collector on the gcbench micro benchmark. We enable thread-local allocators and parallel marking with eight GC threads on BDW. We run on the same machine for the comparison, and use a moderate heap size of 25 MB (which is roughly 2× minimal heap size).

In a run of 20 invocations (as shown in Table 3), the average execution time for BDW is 172 ms, while the average for our implementation is 97 ms (79 % faster). We do not find the result surprising. Our GC implements an Immix allocator which is mainly a bump pointer allocator, while BDW uses a free list allocator. Immix outperforms freelist allocators by 16% in large benchmarks [17]; we expect the performance advantage is even bigger in micro benchmarks that allocate in a tight loop. We ran our `alloc` micro benchmark for BDW, and we find that the performance difference between our allocator and the BDW allocator is similar, confirming our belief. Our GC implementation is different from the BDW collector in a few other respects, which contribute to the performance difference: (*i*) Our GC is conservative with stacks but precise with the heap, while the BDW collector is conservative with both; (*ii*) Our GC presumes a specified heap size and reserves contiguous memory space for the heap and metadata side tables during initialization, while the BDW collector allows dynamic growing of a discontiguous heap.

We also compared the two collectors with a multi-threaded version of gcbench (as `mt-gcbench` in Table 3). Both collectors use eight allocator threads, and eight GC threads. The results show that our implementation performs 2× faster than BDW on this workload. Our implementation outperforms the BDW collector on `gcbench` and `mt-gcbench` (respectively by 79 % and 2×), which suggests our implementation in Rust delivers good performance compared to the widely used BDW collector.

	BDW	Immix(Rust)	% of BDW
`gcbench`	172 ± 0.8 ms	97 ± 0.3 ms	56%
`mt-gcbench`	1415 ± 3.1 ms	466 ± 1.9 ms	33%

Table 3. Performance comparison between our Immix GC in Rust and BDW on gcbench and multi-threaded gcbench.

We conclude that using Rust to implement GC does not preclude high performance, and justify this with the following observations: (*i*) our implementation in Rust performs as well as our C implementation using the same algorithm in performance-critical paths, and (*ii*) our implementation in Rust outperforms the widely-used BDW collector on `gcbench` and `mt-gcbench`. This result shows the capability of Rust for high-performance GC implementations, as a language with memory, thread and type-safety.

7. Conclusion

Rust is a compelling language that makes strong claims about its suitability for systems programming, promising both performance *and* safety. We explored Rust as the implementation language for a high performance garbage collector by implementing the Immix garbage collector in both C and Rust, and used micro benchmarks to evaluate their performance alongside the well-established BDW collector.

We found that the Rust programming model is quite restrictive, but not needlessly so. In practice we were able to use Rust to implement Immix. We found that the vast majority of the collector could be implemented naturally, without difficulty, and without violating Rust's restrictive static safety guarantees. In this paper we have discussed each of the cases where we ran into difficulties and how we overcame those challenges. Our experience was very positive: we enjoyed programming in Rust, we found its restrictive programming model *helpful* in the context of a garbage collector implementation, we appreciated access to its standard libraries (something missing when using a restricted language such as restricted Java), and we found that it was not difficult to achieve excellent performance. Our experience leads us to the view that Rust is very well suited to garbage collection implementation.

Acknowledgments

We gratefully acknowledge the comments and thoughtful feedback that we received from Kathryn McKinley, Eliot Moss, Hans Boehm and our anonymous reviewers. Data61 is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This work is also supported by National Science Foundation grant no. CCF-1408896 and Australian Research Council Discovery grant no. DP140103878.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, Nov. 1999. doi: 10.1145/320384.320418.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open source research community. *IBM Systems Journal*, 44(2):399–417, May 2005. doi: 10.1147/sj.442.0399.

[3] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, Arizona, June 2008. doi: 10.1145/1375581.1375586.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. doi: 10.1109/icse.2004.1317436.

[5] H.-J. Boehm. Threads cannot be implemented as a library. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005. doi: 10.1145/1065010.1065042.

[6] H.-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX Conference on Hot Topics in Parallelism*, Berkeley, California, May 2011. URL http://www.usenix.org/events/hotpar11/tech/final_files/Boehm.pdf.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, Sept. 1988. doi: 10.1002/spe.4380180902.

[8] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, California, June 1992. doi: 10.1145/113445.113459.

[9] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *ACM Symposium on Parallelism in Algorithms and Architectures*, Las Vegas, Nevada, July 2005. doi: 10.1145/1073970.1073974.

[10] D. Frampton. *Garbage collection and the case for high-level low-level programming*. PhD thesis, Australian National University, June 2010.

[11] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Washington, DC, Mar. 2009. doi: 10.1145/1508293.1508305.

[12] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent GC for servers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. doi: 10.1145/512529.512546.

[13] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, Oregon, Oct. 2006. doi: 10.1145/1176617.1176753.

[14] Rust. The Rust Language. URL https://www.rust-lang.org.

[15] Rust RFC 1543. Add more integer atomic types, May 2016. URL https://github.com/rust-lang/rfcs/pull/1543.

[16] R. Shahriyar, S. M. Blackburn, X. Yang, and K. M. McKinley. Taking off the gloves with reference counting Immix. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, Indianapolis, Indiana, Oct. 2013. doi: 10.1145/2509136.2509527.

[17] R. Shahriyar, S. M. Blackburn, and K. S. McKinley. Fast conservative garbage collection. In *ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications*, Portland, Oregon, Oct. 2014. doi: 10.1145/2660193.2660198.

[18] C. Wimmer, M. Haupt, M. L. van de Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4): 30:1–30:24, Jan. 2013. doi: 10.1145/2400682.2400689.

Appendices

Here we present the key Rust code in our implementation, notably fast-path allocation and parallel mark and trace.

A. Allocation Fastpath

```
pub struct ImmixMutatorLocal {
  id         : usize,

  // use raw pointer here instead of AddressMapTable
  // to avoid indirection in the fast path
  alloc_map  : *mut u8,
  space_start: Address,

  // cursor and limit will be invalid after GC.
  // we avoid using Option<Address>. instead,
  // we reset both cursor and limit to Address::zero()
  // so that alloc will go to slow path
  cursor     : Address,
  limit      : Address,
  line       : usize,

  space      : Arc<ImmixSpace>,
  block      : Option<Box<ImmixBlock>>,

  global     : Arc<ImmixMutatorGlobal>
}

impl ImmixMutatorLocal {
  ..

  #[inline(always)]
  pub fn alloc(&mut self,
               size: usize, align: usize)
               -> Address
  {
    let start
      = self.cursor.align_up(align);
    let end = start.plus(size);

    if end > self.limit {
      // thread-local slow path
      self.try_alloc_from_local(size, align)
    } else {
      self.cursor = end;
      start
    }
  }

  // one byte(u8) per every word-aligned address:
  // 6 bits to encode referneces in first 48 bytes
  // 1 bit to indicate the start of an object
  // 1 bit to indicate whether it is larger than
  //   48 bytes
  // we use a side table to store the byte
```

```
  #[inline(always)]
  pub fn init_object(&mut self,
                     obj: Address, encode: u8) {
    let index = (obj.diff(self.space_start)
                 >> LOG_PTR_SIZE) as isize;
    unsafe {
      *self.alloc_map.offset(index) = encode;
    }
  }
}
```

B. Parallel Mark and Trace

```
extern crate crossbeam;

#[cfg(feature = "parallel-gc")]
use self::crossbeam::sync::chase_lev::*;

#[cfg(feature = "parallel-gc")]
pub fn start_trace(roots: &mut Vec<ObjectReference>,
                   immix_space: Arc<ImmixSpace>,
                   lo_space: Arc<FreeListSpace>)
{
  // create work deque: one Worker with several
  // Stealers. Worker can push to deque, while
  // Stealer can only pull
  let (mut worker, stealer) = deque();

  // push roots to the shared deque
  while !roots.is_empty() {
    worker.push(roots.pop().unwrap());
  }

  loop {
    // since the deque allows only one Worker,
    // we create an asynchronous channel for
    // stealers to pass back references to the
    // controller, then the controller
    // with the Worker will push them to deque
    let (sender, receiver)
      = channel::<ObjectReference>();

    // launch parallel GC threads
    let mut gc_threads = vec![];
    for _ in
      0..GC_THREADS.load(atomic::Ordering::Relaxed)
    {
      let new_immix_space = immix_space.clone();
      let new_lo_space = lo_space.clone();
      let new_stealer = stealer.clone();
      let new_sender = sender.clone();
      let t = thread::spawn(move || {
        start_steal_trace(new_stealer, new_sender,
                          new_immix_space,
                          new_lo_space);
      });
      gc_threads.push(t);
    }

    // controller gives up its Sender,
    // thus only stealers own Sender.
    // when all stealers quit (Senders dropped),
    // the loop ends
    drop(sender);

    // main loop for the controller
    loop {
      // fetch from the channel
      let recv = receiver.recv();
      match recv {
        // push obj reference to deque
        Ok(obj) => worker.push(obj),
        // job finishes
        Err(_) => break
      }
    }

    // a sanity check:
    // since we use an asynchronous channel, it is
    // possible that stealers find an empty deque
    // and quit before the controller receives and
    // pushes more work to the deque. however, this
    // never happened for us. defining a reasonable
    // PUSH_BACK_THRESHOLD will allow stealers enough
    // time to work through their local queue before
    // trying to fetch work from global deque.
    // for robustness, we need a backup solution here,
    // and it has little implications on performance
    // unless triggered
    match worker.try_pop() {
      // leftover work in the deque (the rare case)
      // we will relaunch gc threads
      Some(obj_ref) => worker.push(obj_ref),
      // parallel gc finishes
      None => break
    }
  }
}

#[cfg(feature = "parallel-gc")]
fn start_steal_trace
  (stealer: Stealer<ObjectReference>,
   job_sender: Sender<ObjectReference>,
   immix_space: Arc<ImmixSpace>,
   lo_space: Arc<FreeListSpace>)
{
  let mut local_queue = vec![];

  // load invariant fields used in the loop
  let line_mark_table = &immix_space.line_mark_table;
  let alloc_map = immix_space.alloc_map.ptr;
  let trace_map = immix_space.trace_map.ptr;
  let immix_start = immix_space.start();
  let immix_end = immix_space.end();
  let mark_state = objectmodel::MARK_STATE
    .load(Ordering::Relaxed) as u8;

  loop {
    let obj = {
      // fetch work from local queue if possible
      if !local_queue.is_empty() {
        local_queue.pop().unwrap()
      } else {
        // otherwise, steal from global deque
        let work = stealer.steal();
        match work {
          // global deque is empty, the thread quits
          Steal::Empty => return,
          // lost a race to steal, retry
          Steal::Abort => continue,
          // get work load, proceed tracing object
          Steal::Data(obj) => obj
        }
      }
    };

    // as steal_trace_object() is inlined,
    // passing arguments is no-op
    steal_trace_object(obj, &mut local_queue,
                       &job_sender, alloc_map,
                       trace_map, line_mark_table,
                       immix_start, immix_end,
                       mark_state, &lo_space);
  }
}

// functions are inlined so we omit
// parameters/arguments
#[inline(always)]
#[cfg(feature = "parallel-gc")]
pub fn steal_trace_object(...) {
  let addr = obj.to_address();

  if addr >= immix_start && addr < immix_end {
    // mark object in immix space
    mark_as_traced(trace_map, immix_start,
                   obj, mark_state);
    // and associated lines
    line_mark_table.mark_line_live(addr);

    let mut base = addr;
    loop {
      let value
```

```
    = objectmodel::get_ref_byte(alloc_map,
                                immix_start, obj);
  let ref_bits =
    lower_bits(value, REF_BITS_LEN);
  let short_encode =
    test_nth_bit(value, SHORT_ENCODE_BIT);

  // fast path trace for common patterns
  match ref_bits {
    // first word is reference
    0b0000_0001 => {
      steal_process_edge(base, ...);
    },
    // first two words are references
    0b0000_0011 => {
      steal_process_edge(base, ...);
      steal_process_edge(base.plus(8), ...);
    },
    // first 4 words are references
    0b0000_1111 => {
      steal_process_edge(base, ...);
      steal_process_edge(base.plus(8), ...);
      steal_process_edge(base.plus(16), ...);
      steal_process_edge(base.plus(24), ...);
    },
    // more patterns
    ...
    // slow path
    _ => trace_object_slow(base, ref_bits)
  }

  // for objects that are smaller than 48 bytes,
  // its reference locations can be encoded within
  // 1 byte
  // we finished tracing its fields
  if short_encode {
    return;
  } else {
    // for larger objects, we adjust base pointer
    // and trace again
    base = base.plus(REF_BITS_LEN * 8);
  }
 }
} else {
 // mark and trace a large object
 lo_space.mark_trace(obj)
}
}

pub const PUSH_BACK_THRESHOLD : usize = 50;

#[inline(always)]
#[cfg(feature = "parallel-gc")]
pub fn steal_process_edge(...) {
 // load the objectreference from the field
 let obj = unsafe{addr.load::<ObjectReference>()};

 // push the object reference to work queue
 // if it is not zero and not traced
 if !obj.to_address().is_zero()
   && !is_traced(trace_map, immix_start,
                 obj, mark_state) {
   // prioritize using local queue
   if local_queue.len() < PUSH_BACK_THRESHOLD {
     local_queue.push(obj_addr);
   } else {
     job_sender.send(obj_addr).unwrap();
   }
 }
}
```