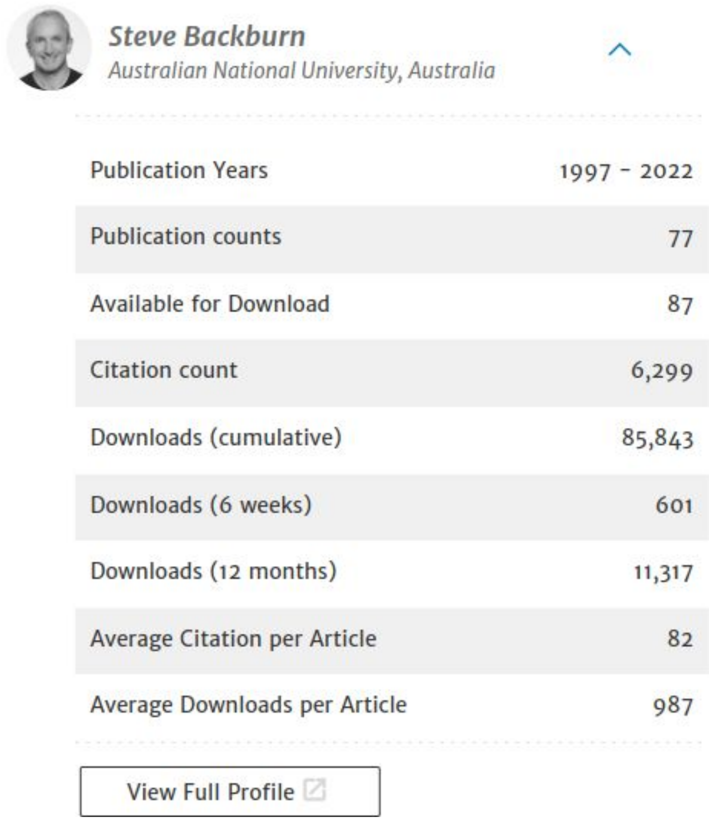
Rust as a Language for High Performance GC Implementation

paper review

Meta data

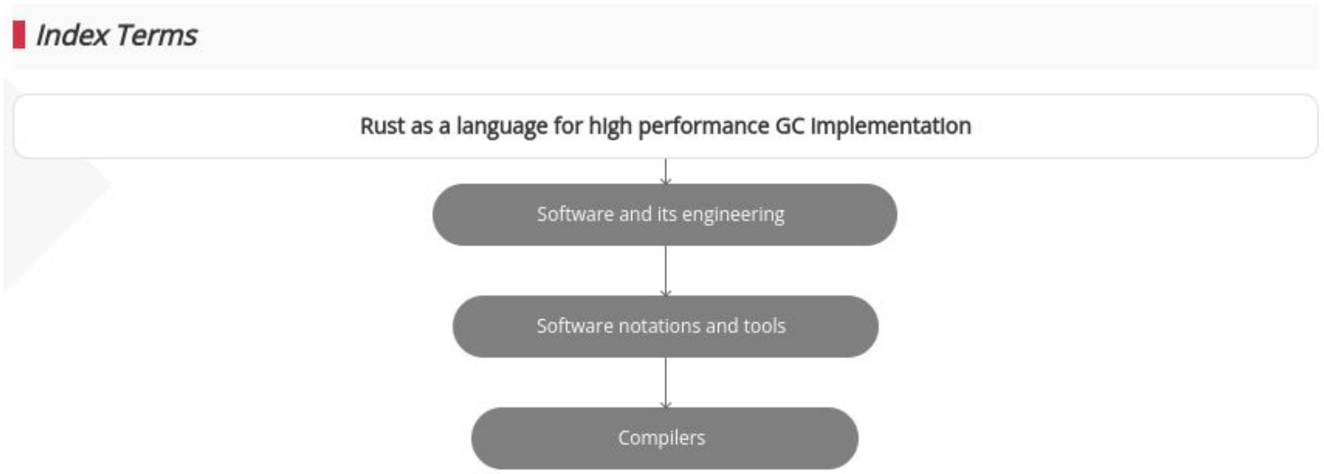
- Authors: Yi Lin, Stephen M. Blackburn, Antony L. Hosking, Michael Norrish.
- Conference: ACM SIGPLAN International Symposium on Memory Management.
- 8 citations.
- 4094 Downloads.
- 10 pages.


Steve Backburn
Australian National University, Australia

Publication Years	1997 - 2022
Publication counts	77
Available for Download	87
Citation count	6,299
Downloads (cumulative)	85,843
Downloads (6 weeks)	601
Downloads (12 months)	11,317
Average Citation per Article	82
Average Downloads per Article	987

Paper layout.

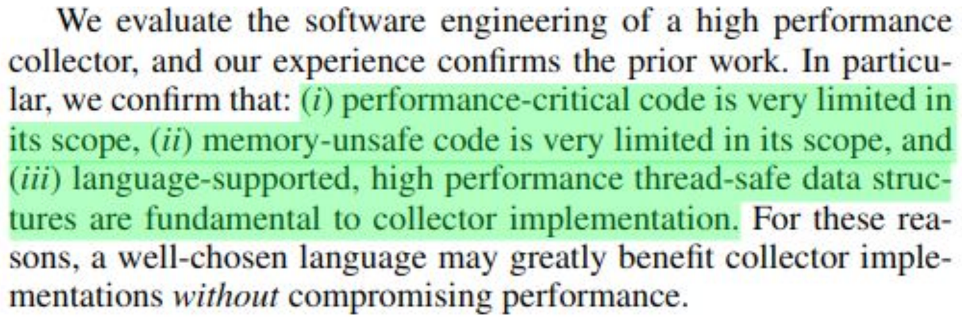
1. Abstract.
2. Introduction.
3. Rust Background.
4. Using Rust.
5. Abusing Rust.
6. Evaluation.
7. Conclusion.
8. Appendices.

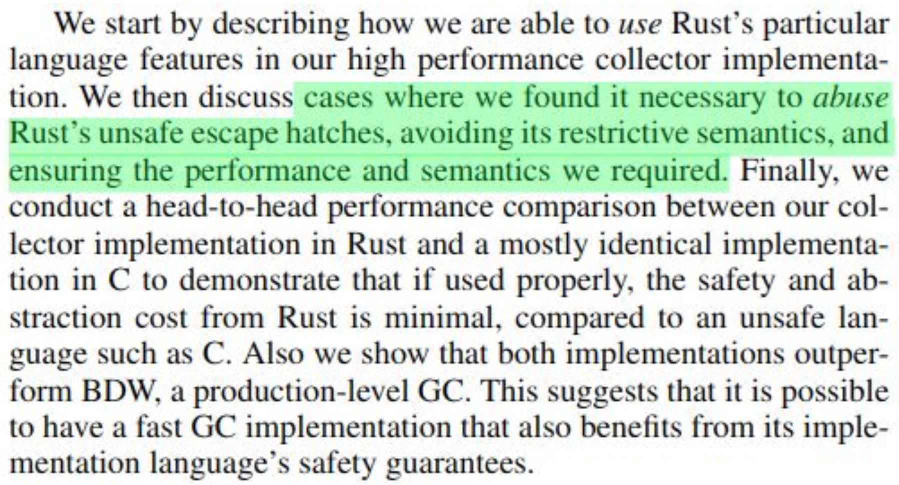

Index Terms
Rust as a language for high performance GC implementation
Software and its engineering
Software notations and tools
Compilers

My personal assessment by criteria

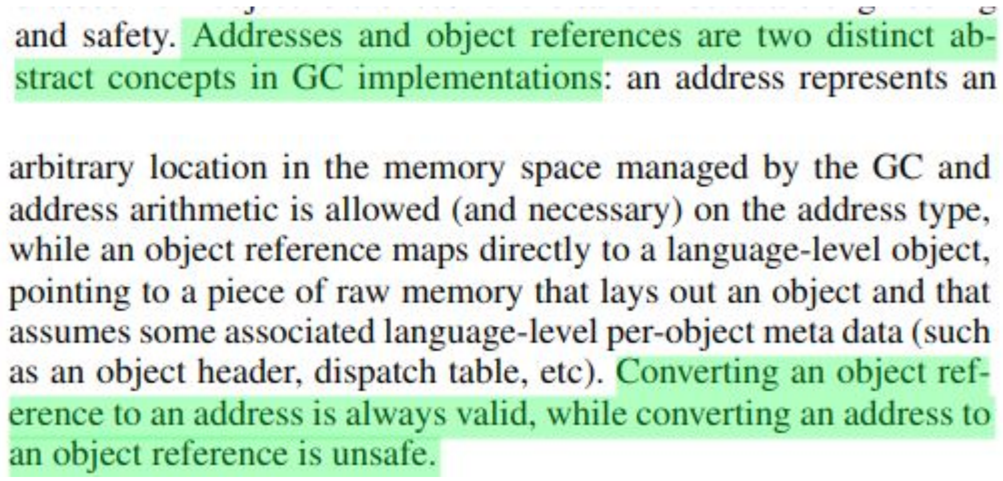
Problem statement.	👍
Innovation.	😐
Contribution.	😐
Logical correctness.	👎
Proof of statements.	👎
Readability.	👍

Intro


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.

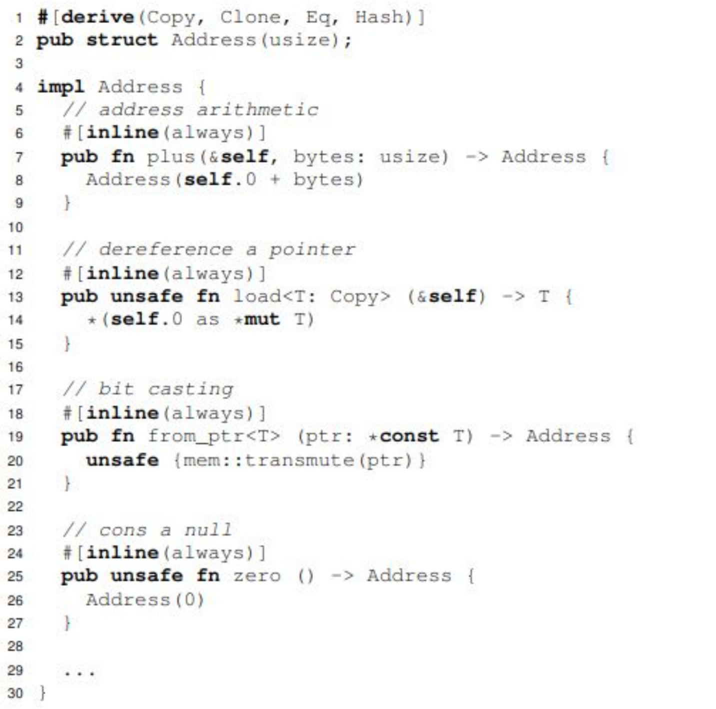

We start by describing how we are able to *use* Rust's particular language features in our high performance collector implementation. We then discuss cases where we found it necessary to *abuse* Rust's unsafe escape hatches, avoiding its restrictive semantics, and ensuring the performance and semantics 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.

Using Rust



and safety. Addresses and object references are two distinct abstract concepts in GC implementations: an address represents an

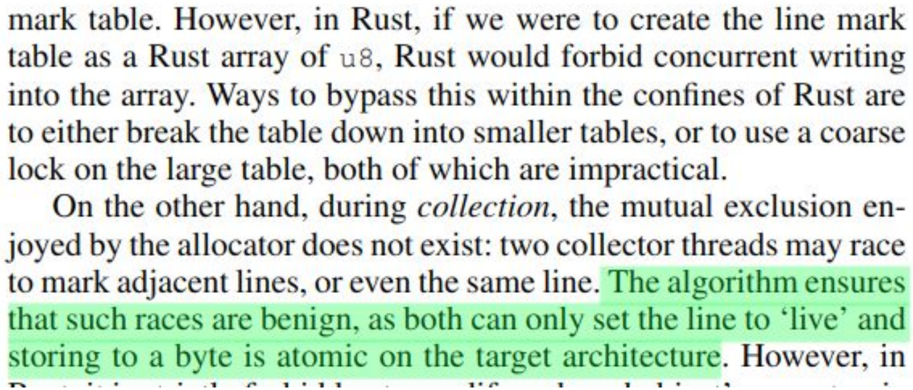
arbitrary location in the memory space managed by the GC and address arithmetic is allowed (and necessary) on the address type, while an object reference maps directly to a language-level object, pointing to a piece of raw memory that lays out an object and that assumes some associated language-level 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.



```
#[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)
  }

  ...
}
```

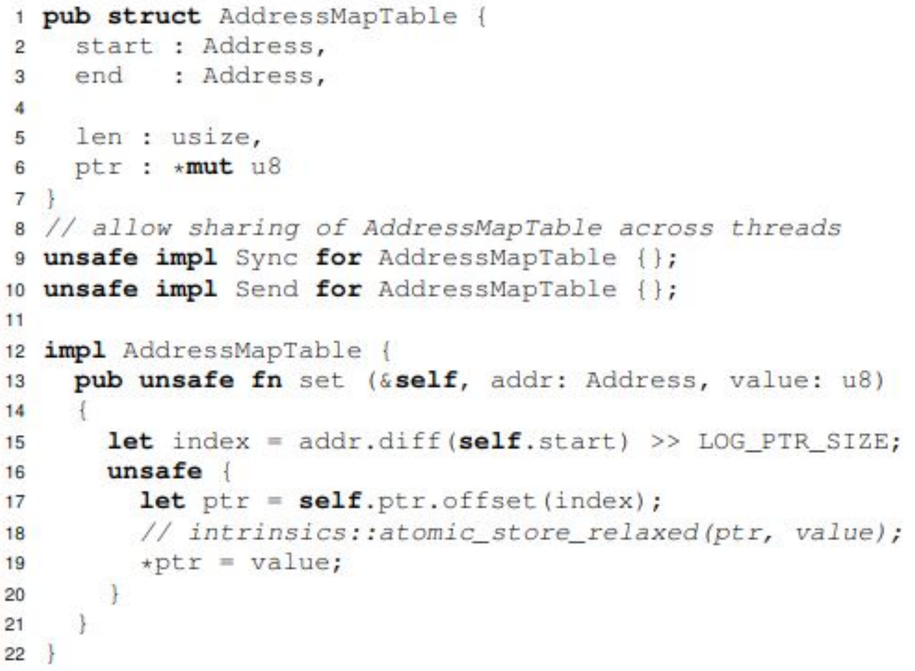
Figure 1. An excerpt of our `Address` type, showing some of its safe and unsafe methods.

Abusing Rust



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
pub struct AddressMapTable {
  start : Address,
  end   : Address,

  len : usize,
  ptr : *mut u8
}
// allow sharing of AddressMapTable across threads
unsafe impl Sync for AddressMapTable {};
unsafe impl Send for AddressMapTable {};

impl AddressMapTable {
  pub unsafe fn set (&self, addr: Address, value: u8)
  {
    let index = addr.diff(self.start) >> LOG_PTR_SIZE;
    unsafe {
      let ptr = self.ptr.offset(index);
      // intrinsics::atomic_store_relaxed(ptr, value);
      *ptr = value;
    }
  }
}
```

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.

Evaluation

	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.

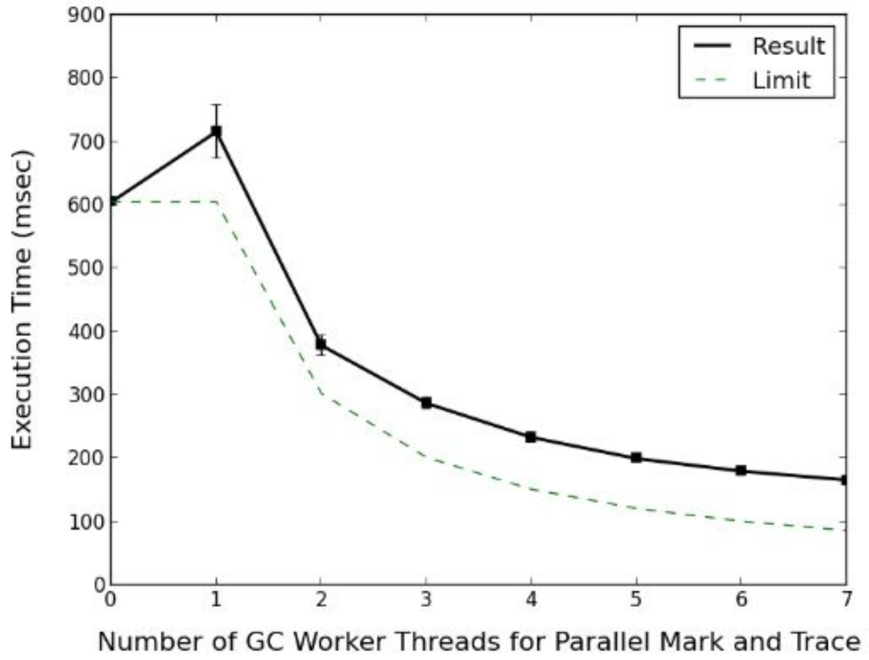


Figure 5. Performance scaling for our fast implemented libraries-based parallel mark and trace collector.

Conclusion

As for me, the article does not offer anything fundamentally new. Nevertheless, it is pleasant to read and it is understandable in most moments. It is good as an article on the habrahabr.