Enveloping Implicit Assumptions of Intrusive Data Structures within Ownership Type System

Keunhong Lee
School of Computing
KAIST
khlee@an.kaist.ac.kr

Jeehoon Kang
School of Computing
KAIST
jeehoon.kang@kaist.ac.kr

Wonsup Yoon
School of Computing
KAIST
wsyoon@an.kaist.ac.kr

Joongi Kim
Lablup Inc.
joongi@lablup.com

Sue Moon
School of Computing
KAIST
sbmoon@kaist.edu

Abstract

Intrusive data structures (IDSes) are heavily used in system programming, where achieving high performance is one of the most important design goals. Yet, they are not supported in today's ownership type system that offer memory-safety without garbage collection. Instead, IDSes force programmers to choose either unsafety or runtime overhead. This limitation stems from the implicit assumptions pertaining to the memory layouts and access patterns created by IDSes.

In this paper, we propose a new technique, referred to as *ownership pooling*, which defines ownership for IDSes. Ownership pooling consists of three new types, `FieldOf`, `OwnershipPool`, and `Shared`, and their conversion rules.

We implemented the proposed types within Rust's type system and compared its performance capabilities against the existing memory-safe implementations and the C++ implementation without memory safety as baseline. The performance of our implementation shows far better performance than that of the existing memory-safe ones and comparable to that of C++ implementation without memory safety.

1 Introduction

Type-safe languages[1] provide improved safety guarantees by eliminating memory bugs such as use-after-free, memory leak, and null dereference. Despite their safety guarantees, they are not as widely adopted as C/C++ in the systems community, because most of them rely on garbage collection that causes performance degradation. Rust, a recently developed memory-safe language, offers memory safety without the runtime overhead of garbage collection. Rust is used in microkernels such as Redox and Tock [6, 10], a web layout engine [14], and network virtualization frameworks [11, 12].

The key component of Rust's memory safety without garbage collection is its *ownership type system*. Ownership type systems require a language to define a unique, non-cloneable handle for every object and corresponding ownership transfer rules for every operation. The ownership type system then enforces the condition stating that the constraints of the ownership type be met over every operation. Rust's ownership type system was built for standard data structures and synchronization primitives. However, an ownership type system has not been built for all types of data structures. One prominent exclusion is the intrusive data structures (IDSes).

IDSes are common. Especially, most data structures in the Linux kernel code are intrusive. They do not allocate their own metadata (*e.g.,* the `prev` and `next` pointers in linked lists) and require the elements to contain the metadata. Thus, the insertion or removal of an element takes place without memory allocation or deallocation. They require fewer pointer indirections, as pointer offsets are used instead. Due to these performance advantages, IDSes have been widely used in operating systems, database systems, and game engines.

Despite their importance in system programming, no ownership type is clearly defined for IDSes due to their implicit assumptions regarding the memory layout and access pattern. In this work, we review their operations and extract implicit assumptions. By articulating these assumptions in new types, we encapsulate the ownership of IDSes within the confine of Rust's type system. First, we propose the `FieldOf` type to capture in-memory representations of IDSes. Subsequently, we use the `Shared` type to represent reference-counting pointers without claiming ownership. Lastly, we introduce the `OwnershipPool` type to declare an effective owner of the elements of IDSes. On top of the three types,

[1] In this paper, type-safe language refers to a strongly-typed and memory-safe language.

PLOS'19, October 27, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365403

we built such IDSes as linked lists, red-black trees, and concurrent stacks. In our evaluation, we show that our IDS implementations perform better than the existing memory-safe solutions and comparable to C++ implementation without memory safety.

2 Background of Type Systems

2.1 Types and Conversion Rules

A type system consists of a set of types and corresponding conversion rules defined over every expression. In C/C++, `int` and `double` are the types and `(int) + (double) -> (double)` is one of the conversion rules for addition expressions. Types also dictate constraints that must be preserved during type conversions. In the above example, the type `int` declares that the variable's 4-byte memory be treated as a signed 32-bit integer while `double` dictates that the 8-byte memory is treated as a 64-bit floating number.

However, C++ does not guarantee that a memory segment adheres to conversion rules of a specific type. We present the following example.

```
1 double a = 1.0;
2 long b = *((long*)&a);
```

In the first line, the variable `a` is declared as a double-precision floating number and 8-byte memory is allocated accordingly. In the second line, the memory is treated as a 64-bit integer. Thus, the value of `b` becomes *undefined*, depending on the floating number representation and the host system's endianness.

2.2 Type-Safe Language

A type-safe language imposes strict conversion rules so that a variable's type is preserved through operations and every operation output is well-defined. Let us examine how type-safe programming languages enforce conversion rules among types. We use two types, `Nullable` and `NonNull`, found in most type-safe (and strongly-typed) programming languages[2]. The type invariants of the two types follow their names: `Nullable` may be a null pointer, while `NonNull` should never be a null pointer. Their conversion rules dictate that the invariants are never violated.

It is straightforward to consider that a conversion from `NonNull` to `Nullable` is safe, as shown below.

```
1 NonNull x;
2 Nullable y = (Nullable)x; // always safe
```

The conversion in the opposite direction is not safe and is only allowed under the conditions defined below.

```
3 if y != Null {
4   NonNull z = (NonNull)y; // conditionally safe
5 } else {
6   Nullable z = y;
7 }
```

[2] `Optional` type plays a similar role in some languages. *e.g.*,`std::optional` in C++17 or `java.util.Optional` in Java8

Note that the single line of type casting, line #2, is a safe conversion only because `x` is declared as `NonNull`. For any variable `y`, line #4 alone is not a safe conversion and `if-else` in lines #3–7 enforces `NonNull`'s invariant. We say that an operation is *safe* when a type's invariant is enforced through an operation itself, and is *unsafe* when it is not. The above conversion contains a single (potentially) *unsafe* statement, but the code block becomes *safe* due to the if-then-else construct.

2.3 Memory Safety

Memory-safety is the major reason behind the broad adoption of type-safe languages, as such languages eliminate potential memory bugs, including null dereference, use-after-free, double free, and memory leak. We demonstrate how memory-safety is enforced by adding two additional conversion rules to the above examples.

First, we add a rule that `Nullable` should never be dereferenced. A `Nullable` variable may reference a valid object. If this rule is enforced, such a variable should be converted beforehand to `NonNull` before being dereferenced. The compiler of a type-safe language enforces this conversion rule.

The second rule to add is that `NonNull` should always point to a valid memory segment. The previous rules should enforce the pointer so as not be null in dereferencing cases, but there is no guarantee that the pointed address is not only not null but valid. The easiest way to enforce this constraint is to use a garbage collector. This is why most type-safe languages rely on garbage collection, leading to performance degradations.

2.4 Rust and the Ownership Type System

Rust, a type-safe language designed for system programming, relies on an *ownership type system* to ensure the validity of pointers and prevent memory leaks. Here, we briefly explain the basic concepts of an ownership type system. *Ownership* is a type by which every variable must have exactly one owner. In Rust, ownership is implemented as a unique handle onto the memory segment of a variable. As the ownership type system tracks ownership at compile time, Rust programmers do not worry about races, as ownership ensures exclusive access to a variable and because the Rust compiler automatically frees a variable without the potential risk of use-after-free. The following C++ example and the succeeding Rust example demonstrate the difference between Rust's memory management and simple RAII (Resource Acquisition Is Initialization) objects [13].

```
1 vector<A> vec;
2 {
3   A a;
4   vec.push_back(a); // A's copy constructor
5   a; // a is still accessible
6 }
7 // A's destructors invoked twice; vec's once
```

According to the above C++ example, inserting an element into a vector invokes a copy constructor. At the end of the code, a is destroyed, and its copied object pushed into vec is destroyed along with the containing vector's destructor. At this stage, we can write the same code in Rust as shown below:

```
1 let mut vec = vec![];
2 {
3   A a = A{};
4   vec.push_back(a); // a's ownership moved to vec
5   // a; // a is not accessible
6 }
7 // A's and vec's destructors invoked once each
```

In Rust, inserting an element at line #4 moves the element's ownership to the containing vector. Thus, a is no longer accessible and its destructor is not invoked after the scope ends. A's destructor is invoked when the containing vector is destroyed. The above comparison demonstrates clearly how the ownership type system enforces the "safety" of a variable by enforcing single ownership through insertions and removals in and out of data structures.

3 Ownership Pooling for an Intrusive Data Structure

IDSes have operations that are not represented by existing ownership type systems. The challenge lies in the complex memory layout and implicit rules to follow when accessing them, such as pointer offsetting. In the current version of Rust, IDS ownerships are not handled by the type system and are implemented using locks, incurring major runtime overhead.

In this section, we introduce the ownership pooling technique that explicates memory layout information and defines ownerships for IDSes. It includes new types, constraints, and conversion rules that comply with existing ownership type systems. Next, we begin with a new type that captures the implicit assumptions pertaining to the memory layout.

3.1 `FieldOf`: Captures memory layout

The memory layout is a crucial piece of information related to the operation of an IDS. Unlike non-IDS cases, IDSes use this information to convert an entry to an element, and vice versa. First, we describe how this information is handled without a type system in an intrusive doubly-linked list.

```
1 struct Element {
2   int data;  ListEntry entry;
3 } elem;
4 struct Element e = {0, {0, 0}};
5 list.push_back(&e.entry);
6 struct ListEntry* tail_m = list.tail();
7 struct Element* tail = (char*)tail_m - OFFSET;
```

In this example, we insert an element at the tail of an intrusive list and retrieve the tail element. Below we point out line by line (#4-#7) how IDSes differ from non-IDSes.

1. Metadata is initialized along with e, not via an explicit API.
2. `push_back()` takes `&e.entry` as an argument, not e or &e.
3. `tail()` returns `ListEntry*`, not `Element*`.
4. `(tail_m - OFFSET)` always points to a valid `Element`.

The last point is not an enforcement, but rather an assumption that a programmer makes about the memory layout. Thus, programmers using IDSes must pay close attention to the memory layout; this assumption is not clearly enumerated in any type.

We introduce the `FieldOf` type, which explicitly denotes the memory layout information of an IDS. It forces the programmer to outline the memory layout and prevent undefined behavior. We use it as a replacement for entry pointers such as `ListEntry*`.

`FieldOf` is a 3-tuple of the element type, the IDS entry type, and the offset[3]. It represents the constraint stating that the pointer of an element type plus the offset points to the IDS entry type.

```
1 struct Element {
2   int data;  ListEntry entry1;  ListEntry entry2;
3 } elem;
4 FieldOf<Element, ListEntry, 4> *e1 = &elem.entry1;
5 FieldOf<Element, ListEntry, 20> *e2 = &elem.entry2;
```

Lines #6-7 above replace naive pointers with specific pointers with memory layout information. Following conversion rules enforce the above constraint.

1. `((char*)Element* + 4)` and `(FieldOf*)` are interchangeable
2. `(ListEntry*)` and `(FieldOf*)` are interchangeable

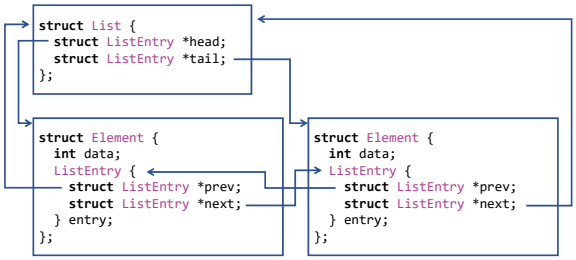
Note that the third offset parameter not only dictates the memory layout but also disambiguates the fields with the same entry type.

3.2 `OwnershipPool`: Declares an effective owner of elements' data fields

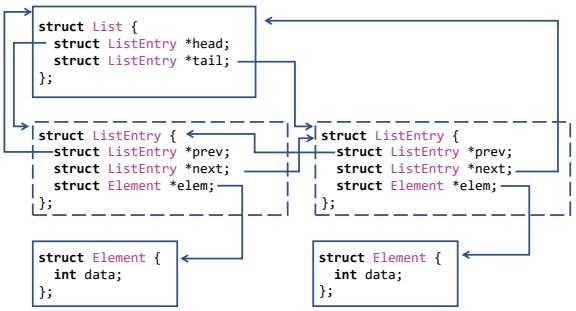
At this point, we discuss the effective ownership found in common use cases of IDSes. For non-IDSes, a data structure's ownership states that the container has the ownership of the enclosed elements. We access the elements of a non-IDS while iterating through the data structure.

For IDSes, the container does not have exclusive ownership of its elements, as an element contains multiple IDS entries and no IDS can claim exclusive ownership of an element. For instance, the `task_struct` structure, which manages all threads and processes of Linux, contains as its fields more than 20 IDS metadata entries, each of which is used

[3]This tuple of types is represented as 'Traits' in Rust.



(a) Memory layout of an intrusive doubly-linked list



(b) Memory layout of a non-intrusive doubly-linked list

Figure 1. Memory layout of intrusive and non-intrusive lists. The solid box represents the memory chunk allocated by programmers and the dashed box represents allocation inside the list library.

for managing lock-wait lists, scheduling queues, or process hierarchy.

```
1 // OwnershipPool ownership_pool;
2 for (auto it = list.iter(); it != end; ++it) {
3   struct ListEntry* ptr = *it;
4   struct Element* e_ptr = (char*)ptr - OFFSET;
5 //  if (ownership_pool.is_owning(e_ptr)) {
6       e_ptr->data += 1; // possible races
7 //  }
8 }
```

Though exclusive ownership is not guaranteed, looping through an list's element is easily found in the Linux kernel, e.g., iterating through task lists inside a scheduler. Line #6 includes the potential risk of races; however it does not incur any race in actual use cases. From a careful observation, we learned that safety does not come from the list itself, but from an understanding of the abstract task that the list is designed to perform. For a scheduler, there are a number of distinct queue structures managing multi-level priority scheduling, a least recently used (LRU) list, or a sibling list for sibling processes. However, each list and queue refers to the `entry` field assigned to itself and does not refer to other data fields at all. In terms of ownership, the ownership of `e.data` does not belong to one of the scheduling queues but belongs to the scheduler containing all of the queues. In a way, the schedule is the effective owner of the data fields (`e.data`).

We propose a new type, `OwnershipPool`, which represents the effective ownership of elements. Exclusive access to an element's data field is guaranteed by the constraint in line #5, as noted in the previous example. We propose the following type constraints and conversion rules for the effective ownership of elements.

1. An element relinquishes its data ownership to a pool upon joining the pool.
2. Subsequent joins to other pools must fail.
3. A reference to a pooled element must not be dereferenced.
4. Among the elements of a single pool, only one element can be upgraded at any time for data modification.

3.3 Shared: Denoting shared ownership without accessibility

Though an element is composed of dozens of entries shared among multiple IDSes, the whole element occupies a single memory allocation unit and individual entries cannot be allocated nor deallocated separately. Nonetheless, each individual entry owner must ensure the liveness of the enclosed memory segment.

```
1 {
2   struct Element e = {0, {0, 0}};
3   list.push_back(&e.entry);
4 } // e is destroyed here.
5 list.tail(); // use-after-free
```

The element (e) is destroyed at line #4 and its memory segment is deallocated. The list encounters use-after-free when the deallocated element is accessed. We choose to use reference counting to ensure the liveness of a memory segment, increasing the reference count by 1 as each IDS entry field claims its corresponding liveness[4].

Additionally, the reference (`&e.entry`) used to invoke the insertion operation, overclaims the ownership of `e` while the list operation never accesses the other parts of `e`. Despite the fact that the reference is marked as read-only, one cannot claim exclusive write access (i.e., ownership), as doing so would run the risk of undefined behavior of other readers. Thus, we propose a new type, `Shared`, a reference counting pointer without read access. While `Shared` does not have read/write access by default, a `Shared` variable is promoted to a readable/writable reference according to the type conversion rules of `OwnershipPool`.

[4]The other choice is to create a destructor that removes an element from its enclosed list. We may implement such self-removal algorithm for a doubly linked list, but this approach is not general for every data structure such as a singly linked list.

OwnershipPool's conversion rules require runtime verification of the inclusive relationship between a Shared and an OwnershipPool. Our implementation on Shared and OwnershipPool provides the efficient verification of the inclusive relationship.

```
template <Element>
struct Shared {
  atomic_int refcount;  atomic_int pool_id;
  Element elem; }
```

```
struct OwnershipPool {
  atomic_int id; }
```

A Shared structure reserves an additional integer along with the reference count. This integer marks which ownership pool to which it belongs. We use a monotonically increasing integer to ensure an ownership pool's uniqueness constraint. Below are the detailed conversion rules between a Shared and an OwnershipPool extended from that of the OwnershipPool type.

1. pool.register(shared) succeeds when shared's pool_id is not set. shared.pool_id is set to pool.id upon successful registration.
2. A pair of (pool, shared) returns &shared.elem.data when pool.id == shared.pool_id.

3.4 Ownership verification on IDS side

We separated the ownership of the data field and the other metadata fields to protect the metadata from being corrupted by programmers. However, a metadata entry incurs competition for its ownership among the IDSes using the entry. We demonstrate this point using the code below.

```
list1.push_back(&e.entry);
list2.push_back(&e.entry); //Error
```

The second insertion invalidates the list invariant of list1 and thus should be forbidden. A list modifies the internal metadata (prev, next) of e.entry, requiring the ownership of the entry field. However, the ownership is transferred to list1 and the second insertion cannot obtain the ownership. We leverage the actual implementation of the ownership representation to the implementation details of an IDS. In general, OwnershipPool can be used to represent such ownership relationship, however there are additional optimization opportunities depending on the data structure algorithm. For instance, a data structure without removal operation can represent its ownership with a simple null-check to determine whether this metadata entry is already used by another IDS.

4 Evaluation

In this section, we evaluate our IDS implementation against existing memory-safe implementations and naïve implementation without memory-safety. We choose a doubly linked as the most common and popular use cases of IDSes. The Rust standard library, denoted by *Standard*, offers non-IDS implementations of lists. The Rust library, intrusive_collections [4], denoted by *Intrusive*, contains straightforward implementations of an IDS list. To represent the baseline performance (operations per unit time), we use Boost's C++ IDS implementation. In order to maximize its performance, it does not use any safety features, thus representing the *upper bound* of the performance as every operation in this implementation is unsafe and incurs no runtime overhead.

Our evaluation scenario is set in a multi-threading environment. Lists run in a single thread but their elements are shared among other threads, which is a common scenario in many implementations, such as Linux's struct task_struct and Tokio's struct Entry. Existing memory-safe implementations use an atomic reference counted object(Arc) with a lock-based ownership verification, RwLock. The Arc object ensures the liveness of an object, and RWLock ensures exclusive access by acquiring the lock. For our locking mechanisms, we use the following three: POSIX mutexes (Rw), spin locks (Park) [5] and sequential locks (Seq) [3].

4.1 Evaluation Workload

For our evaluation workload, we use the following four types of operations assuming a list with 10M elements.

1. **Creation**: generates a doubly linked list or a red-black tree. All the elements are initialized randomly and pushed one by one.
2. **Delete-insert**: iterates over the previously initialized elements. It deletes each element from the data structure's head and inserts it into its tail.
3. **Read**: iterates through the list and calculates the sum of the values of all the elements.
4. **Write**: iterates through the list and increases the value of each element by one.

Our evaluation server has two Intel Xeon CPUs E5-2670 v3 (12 cores, 2.30 GHz, 30MB cache) with 64GB of main memory. Our implementation uses the Rust version nightly-2019-03-14, and the baseline implementation uses gcc 8.2.0 and Boost 1.68.0. For a fair comparison, we use jemalloc[9] as a backend allocator for all implementations.

4.2 Performance Benefits of Our IDS

Figures 2 presents the evaluation results. The performance of the baseline Boost C++ implementation is denoted by the horizontal line at $y = 1$. For all workload and lock types, the proposed method outperforms *Standard*. Between *Intrusive* and ours, the former with the Park lock mechanism shows comparable performance to the proposed approach with regard to creation and delete-insert workloads. Otherwise, our method greatly outperforms *Intrusive* for read and write workloads, where the latter is x3 to x10 slower than the baseline. Our implementation uses a comparison operation

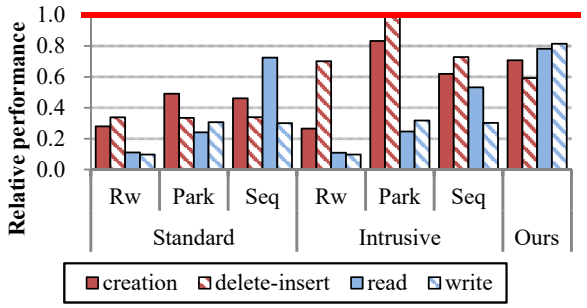


Figure 2. Relative performance against the baseline (Boost, red line) implementation. All values are an average from 20 trials. Higher is better. The data size is fixed to 8B.

while the other two use a bit flag to store the locking state. In modern CPUs, comparisons without modification are usually pipelined and fast.

5 Case Study: Tokio Timer

In this section, we apply our ownership pool to a real-world library, specifically to Tokio's timer library. Tokio is the most popular asynchronous I/O library suite implemented in Rust [7]. In Tokio's timer library, there are multiple different IDSes whose entries move across thread boundaries and we show how our IDS implementations perform in a multi-threaded environment.

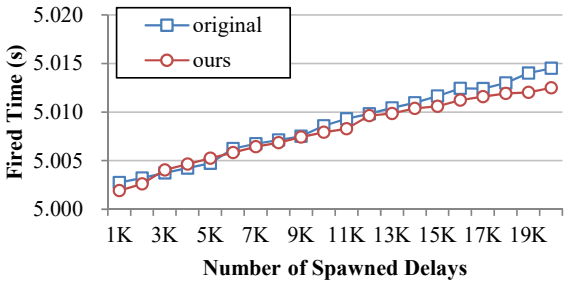


Figure 3. Mean fired time of `Delays`. All values are an average of 20 trials. Lower is better.

We spawn 1K to 20K `Delays` that are fired after 5 seconds and measure when they are actually fired. Figure 3 shows that our Tokio implementation has comparable or slightly better performance than the original implementation. The results illustrate the strengths of our IDS implementation: it has almost zero overhead for both single- and multi-threaded environments, without exposing any unsafe interface.

6 Related Work

Generalization of IDSes The Boost library provides a generalized implementation of IDSes with C++ templates, though it nonetheless cannot offer any safety guarantees given the characteristics of C++. Amanieu's implementation [4] proposed a *memory-safe* abstraction for IDSes; however it treats an element as sharing a single ownership handle. Accordingly, no element can claim full ownership to write unless a mutex is acquired.

Alternatives of Ownership Pooling There are several alternatives that can replace ownership pooling in limited use cases. One is *rooting* [2], which centralizes the ownership of a set of objects into a single object. Rooting is induced from a region-based memory management by stating that the arena/region will take ownership of the allocated memory chunks. For example, a `vector` with 100 elements acts as an allocator by returning non-overlapping indices of the elements. The ownership still belongs to the vector and each element is referenced by its index. In the notation of ownership pooling, `vector` becomes an *OwnershipPool* and the indices become `Shareds`. Compared to ownership pooling, rooting-based memory management entangles the allocation and notation of ownership so that an element cannot change its owner unless its index is invalidated and reclaimed from a new vector.

Application of memory-safe IDSes Several systems [1, 8, 11] claim strong isolation arguments assuming a *memory-safe* program. In the case of NetBricks, a user must write a network function *without* using any IDS, while most high-performance network systems rely on IDSes. Thus, ownership pooling will benefit performance-critical systems with memory-safety arguments based on the soundness of the type system.

7 Conclusion

In this paper, we presented a new technique, referred to as *ownership pooling*, which operates in Rust. It includes three new types, `FieldOf`, `Shared`, and `OwnershipPool`, and their corresponding constraints and conversion rules. Combined, they allow IDSes to be built in a memory-safe manner. We implemented the proposed types and evaluated their application on a doubly-linked list against non-IDS implementations and an existing Rust IDS library in a multi-threaded environment. We used three locking mechanisms for non-IDS implementations and an existing Rust IDS library. The evaluation results show that our implementation delivers as close performance to the baseline implementation of the Boost C++ IDS list. We applied ownership pooling with Tokio's time library and demonstrated performance comparable to that of the original implementation in Rust.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2019R1A2C2008439).

References

[1] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *ACM HotOS*.

[2] Without Boats. 2018. Shifgrethor III: Rooting. https://boats.gitlab.io/blog/post/shifgrethor-iii/.

[3] Amanieu d'Antras. 2016. SeqLock. https://crates.io/crates/seqlock/0.1.1.

[4] Amanieu d'Antras. 2018. intrusive-collections. https://crates.io/crates/intrusive-collections/0.7.2.

[5] Amanieu d'Antras. 2018. parking_lot. https://crates.io/crates/parking_lot/0.6.4.

[6] Redox Project Developers. [n.d.]. Redox - Your Next(Gen) OS. https://www.redox-os.org.

[7] Tokio Project Developers. 2018. Tokio: A runtime for writing reliable, asynchronous, and slim applications with the Rust programming language. https://crates.io/crates/tokio/0.1.6.

[8] Galen C Hunt and James R Larus. 2007. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review* 41, 2 (2007), 37–49.

[9] jemalloc Developers. [n.d.]. jemalloc memory allocator. http://jemalloc.net.

[10] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *ACM SOSP*.

[11] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV.. In *USENIX OSDI*.

[12] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *USENIX NSDI*.

[13] C++ Reference. 2018. RAII. https://en.cppreference.com/w/cpp/language/raii.

[14] Mozilla Research. [n.d.]. Servo, the Parallel Browser Engine Project. https://servo.org.