# Extending Rust with Support for Zero Copy Communication

Arthur Lafrance
University of California, Irvine

David Detweiler
University of California, Irvine

Zhaofeng Li
University of Utah

Xiangdong Chen
University of Utah

Vikram Narayanan
University of Utah

Anton Burtsev
University of Utah

## Abstract

In contrast to hardware-based isolation solutions, language-based systems support crossing of isolation boundaries with an overhead of a function call. Moreover, the strong type system of a safe language provides support for secure communication in the face of complex, semantically-rich interfaces, i.e., support for fault isolation and end-to-end zero-copy communication through isolation of object spaces and controlled ownership on the shared exchange heap. If historically, safety was prohibitive due to overheads of a managed runtime, today, languages like Rust achieve the performance of unsafe C hence empowering language-based systems to support practical isolation with fine-grained boundaries and frequent communication.

Unfortunately, despite providing the core foundation for isolation of object spaces, i.e., support for a special shared heap, Rust still lacks several abstractions required to support zero-copy communication mechanisms. Existing Rust systems restrict zero-copy passing of data to a set of hand-coded types, hence limiting flexibility of changing interfaces between isolated subsystems. Our work extends the Rust compiler with a static analysis pass that reasons about assignments of references on the shared exchange heap and instruments them with the code that correctly reflects ownership updates on cross-subsystem invocations. This allows us to develop an isolation scheme in which a hierarchical data structure can be passed between isolated subsystems with a single ownership update of its root element.

## 1 Introduction

Despite being able to dominate the landscape of isolation solutions for decades, hardware isolation mechanisms are increasingly at odds with the systems we run today. Primarily, hardware abstractions like segmentation and paging were designed to provide efficient virtualization of hardware, i.e. provide an abstraction of a virtual address space for an operating system process. As a resut hardware primitives were optimized for zero-overhead virtualization of long-running computations (a typical process runs for tens of millions of cycles before it context switches) that communicate only infrequently with other parts of the system.

Today, however, we try to retrofit isolation into the systems that are built out of small third-party libraries that frequently cross isolation boundaries, e.g. browser extensions [1, 44, 46, 60], web applications [2, 18, 21, 37], serverless cloud and edge platforms [3, 4, 38, 48, 58], user-defined database functions [15, 55], virtualized network functions [5, 33, 35, 43, 51, 54], device drivers [24, 26, 56], storage stacks and file systems [14, 23], kernel modules [12, 22, 25, 29–32], and more. Typically, such systems perform a small computation and call into another subsysem, i.e., every few hundred cycles – a frequency that challenges the performance of modern hardware isolation mechanisms that even in the ideal case require several hundred cycles to save general and extended registers, switch to a new stack, and change the isolation boundary [40].

In contrast to hardware isolation schemes, programming language safety allows crossing the isolation boundary with an overhead of a function call [6, 9, 11, 28, 34, 45, 59]. In a safe language the execution can continue on the same stack (safety ensures isolation of the caller and callee frames on the stack) and does not require saving and restoring general and extended registers (calling conventions save and restore registers between the caller and callee, and exception handling mechanisms allow recovery from a fault in a callee subsystem). If historically safety was prohibitively expensive due to the overhead of the managed runtime (safe languages like Go and C# show 36-42% overhead compared to unsafe C [20]), Rust demonstrated how safety can be enforced in a practical manner without garbage collection (Rust achieves overheads of only few percents [20, 45, 47]).

Along with low-overhead boundary crossings, safe languages leverage powerful type systems to implement multiple security and reliability properties. Isolation of heaps and restricted ownership creates the foundation for fault recovery [34, 45] and end-to-end zero-copy communication [7, 34, 45, 47]. Language-based systems rely on a special heap construction that combines private subsystem heaps



*PLOS '23, October 23, 2023, Koblenz, Germany*



ACM ISBN 979-8-4007-0404-8/23/10.
https://doi.org/10.1145/3623759.3624552

and a shared exchange heap used for passing objects across isolated subsystems [7, 34, 45]. A combination of heap isolation and single ownership on the shared heap creates a foundation for a true end-to-end zero-copy from user applications to operating system device drivers and hence enables high-throughput I/O workloads that traditionally require access to a dedicated physical device through a user-level device driver [16, 19, 36, 41].

Unfortunately, despite providing the core foundation for isolation and zero-copy communication, Rust still lacks abstractions required to implement generic zero-copy communication mechanisms. For example, several recent systems support zero-copy communication with only a set of hand-coded types [7, 45, 47]. Netbricks supports zero-copy communication via a single fixed interface that allows exchange of a single data structure that represents a batch of network packets [47]. RedLeaf adds support for defining *exchangeable types* that can be safely exchanged across isolation boundaries, but fails to support recursive references on the shared heap due to the lack of mechanisms in the language that can control the ownership of the objects on the shared heap (RedLeaf supports generation of getter and setter methods to control the assignment of objects in hierarchical data structures, but abandons them in favor of a small set of pre-defined types due to cumbersome, non-ergonomic syntax). Splinter fails to support zero-copy and falls back to marshaling complex object hierarchies between isolated subsystems [39].

Our work extends Rust with support for abstractions needed to support zero-copy communication on a shared exchange heap [34, 45], and specifically support for mediating assignments of recursive references on the shared heap. We extend the Rust compiler with a new static analysis pass that reasons about assignments of references on the shared heap and instruments them with code that correctly updates ownership of objects on the shared heap which is required for safe deallocation of objects which are owned on the shared heap by a crashing subsystem [45]. This allows us to develop an isolation scheme in which a hierarchical data structure can be passed between isolated subsystems with a single ownership update of its root element. The instrumented code then guarantees correct tracking of ownership if the hierarchical object is updated. This ensures the enforcement of ownership rules on the shared heap (and thus cleanup of resources on the shared heap in case of a subsystem crash).

## 2 Background: Isolation and Communication in Language-Based Systems

The first principles of language-based isolation were laid out by the early safe operating systems, i.e., the operating systems that relied on programming language safety for isolation of computations [8, 10, 13, 17, 27, 42, 52, 57]. SPIN suggested to use language safety for isolation of kernel extensions [9]. While restricting extensions to safe accesses, SPIN allowed uncontrolled exchange of references between the kernel and isolated subsystems hence leaving the heap in an inconsistent state if one of the extensions crashed. J-Kernel [59] and KaffeOS [6] developed ideas of *isolated heaps*, i.e., private subsystem and shared exchange heaps, as a way to support clean termination of isolated subsystems. Singularity extended isolated private and shared exchange heaps with a *single ownership* on the shared heap, i.e., at any given time, only one isolated subsystem was permitted to have a reference to an object on the shared heap [34]. A combination of single ownership and heap isolation eliminated state sharing across subsystems and hence provided support for clean termination and unloading of crashing subsystems. Singularity developed novel static analysis and verification techniques to enforce single ownership semantics on the exchange heap in an otherwise non-linear language, Sing#. When a reference to an object was passed between subsystems, the ownership of the object was "moved" (an attempt to access the object after passing it to another subsystem was rejected by the verifier). Along with fault isolation, single ownership on the shared heap enabled secure zero-copy communication, i.e., the move semantics guaranteed that the sender of an object was losing access to it and hence allowed the receiver to update the object's state knowing that the sender was not able to access or alter the new state.

***Heap isolation in Rust*** Recently, RedLeaf brought ideas of Singularity to Rust [45]. A unique property of Rust is that it ensures single ownership on the shared heap through its type system (i.e., borrow checker [49]), and hence avoids the need for additional verification mechanisms. Similar to Singularity, RedLeaf separates *private* (per-domain) and *shared* exchange heaps [45] (Figure 1). Objects on the private domain heap are regular Rust data structures with the restriction that they cannot have pointers into other private heaps and a guarantee that they cannot be shared across domains via cross-domain invocations. RedLeaf introduces an idea of *exchangeable types*, for objects that domains can safely exchange across boundaries of isolated subsystems. All objects that can be exchanged across domains are allocated on the exchange heap. RedLeaf introduces an abstraction of a *remote reference*, or `RRef<T>`, that is similar to the default Rust `Box<T>` mechanism for allocating objects on the heap. Object on the exchange heap are allowed to have references to other objects on the exchange heap, but are not allowed to have references into private heaps (since Rust cannot fully enforce this invariant RedLeaf develops a special interface definition language compiler that checks the types of objects on the exchange heap).

Rust does not provide support for garbage collection and allows leaking memory. This means special care must be taken to deallocate objects owned by a crashing subsystem

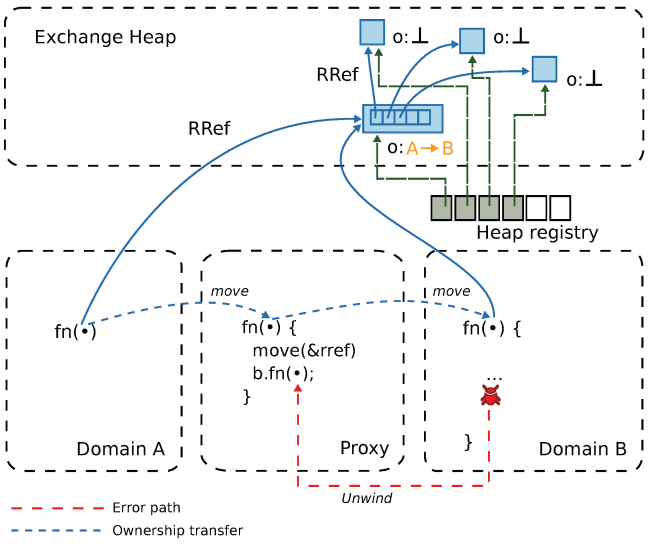


**Figure 1.** Organization of the shared and private heaps in RedLeaf

on the private and shared heaps. To support deallocation of all domain resources, RedLeaf takes the following approach. On a crash, RedLeaf reclaims private domain heap as raw memory. This is safe since no other object in the system outside of the private heap is allowed to have references into the private domain heap and eliminates the need for garbage collection. To clean the shared heap, along with each object on the shared heap RedLeaf maintains information about its owner (due to single ownership on the shared heap, only one domain (owner of the object) has access the object through an `RRef<T>`). On cross-domain invocations, the communication subsystem updates the ownership of objects moving them between domains. The system maintains a system-wide registry of all objects allocated on the shared heap. Upon a crash of a domain, RedLeaf walks through the heap registry and deallocates all objects currently owned by the crashing domain.

***Tracking ownership of hierarchical data structures*** To ensure clean up of the shared heap, RedLeaf track ownership of each object on the shared heap. RedLeaf's IPC subsystem updates the ownership of objects on cross-subsystem invocations moving them between domains. This is challenging for hierarchical data structures as they contain pointers to other data structures on the shared heap. A naive solution is for the IPC subsystem to recursively change the ownerships of all objects reachable from the root when such objects are passed. This, however, can introduce significant overheads as the object closure becomes enormous (e.g., large DOM hierarchies in web browsers).

Instead, RedLeaf argues to change the ownership of only the root object with any nested objects marked with a special owner. During deallocation, such nested objects are deallocated as part of their parent objects' `drop()` methods [45]. Unfortunately, RedLeaf cannot control assignment of references on the shared heap, owing to limitations of Rust.

Specifically, to correctly handle the clean-up of objects on the exchange heap, one must enforce the following domain-ownership invariants:

> **I1.** *Child objects should be marked as bottom (⊥) when moved into a parent* `RRef<T>` *(i.e., the reference is owned by a parent object and not by any of the domains).*

This invariant prevents double-free when the system scans for `RRef<T>`s owned by the crashing domain.

> **I2.** *Objects should be assigned an appropriate domain owner when moved out of the* `RRef<T>` *hierarchy.*

This invariant prevents memory leak by making sure the `RRef<T>` will be attributed to the crashing domain.

Rust does not provide support for mediating assignments to `RRef<T>`s. As a result, RedLeaf makes all `RRef<T>` fields private and generates public `get` and `set` methods for each field to mediate updates to the object hierarchy, at the cost of code ergonomics (generated getter and setter methods become cumbersome for complex data structures like nested structs, tuples, arrays, etc.).

## 3 Assignment Analysis

Our work extends Rust with mechanisms that allow the IPC subsystem to track and update the ownership of the hierarchical data types allocated on the shared exchange heap. We extend the Rust compiler with a static analysis pass that tracks assignments of objects allocated on the shared heap with a user-provided type (`RRef<T>` in our example), and reasons about nesting of `RRef<T>`s to correctly update `RRef<T>` ownership information on the shared heap. During the code generation phase we perform a code transformation that modifies the program during the mid-level intermediate representation (MIR) phase of compilation to correctly uphold ownership invariants for `RRef<T>` assignments.

In order to identify which assignments in the program must be rewritten, the analysis checks whether the right and left sides of the assignments are part of the object hierarchies allocated on the shared heap. Several corner-cases make this seemingly simple task challenging (we discuss them below).

***Composite types*** Composite types like structures, tuples, arrays, and enums may contain multiple `RRef<T>`s and therefore have to be handled by the analysis and the code generation path. Enumerated types, e.g., options and unions, may contain an `RRef<T>`, depending on which variant is present at runtime. For example, `MaybeRRef` may contain zero, one or two `RRef<T>`s:

```
enum MaybeRRef {
  None,
  One(RRef<i32>),
  Two(RRef<i32>, RRef<i32>),
}
```

We generate the following rewriting code that rewrites the assignment at runtime

```
// m is MaybeRRef
t.m = m;

// code injected by the code generation phase
match t.m {
   MaybeRRef::One(ref mut r) => mark_bottom(r),
   MaybeRRef::Two(ref mut r1, ref mut r2) => {
      mark_bottom(r1);
      mark_bottom(r2);
   },
   _ => {},
}
```

During the code generation pass we emit a match on the enumerated value to make the correct ownership decision at runtime. Beyond user-defined enum types, this corner case includes common rust programming patterns such as `Result` and `Option` types that might contain `RRef<T>`s.

For collection types such as array or slices, the analysis must generate code that handles every element to modify each of their ownership properly. Because arrays have a statically-known length, code can be statically generated by the analysis pass for each element. On the other hand, slices do not carry size information at compile-time, so the analysis pass must generate code that iterates over their elements at runtime and modifies the ownership of each one, similar to the runtime pattern matching-based handling of enumerations.

***Immutable borrows*** Rust allows existence of multiple immutable references. In the example below an immutable reference is assigned into the hierarchy of objects on the shared heap.

```
struct T<'a> {
   r: &'a RRef<i32>,
}

fn foo<'a>(t: &mut RRef<T<'a>>, r: &'a RRef<i32>>) {
   t.r = r; // assignment of immutable reference
   ...
}
```

Rust lifetime rules ensure that `t.r` will not outlive the original object, hence we do not need to update the ownership information (the object behind the reference is correctly owned by either domain or another reference).

***Mutable borrows*** When an `RRef<T>` is part of a type that is borrowed mutably, determining whether or not an assignment must be rewritten requires knowing whether the *source* of the borrow is contained within a shared heap object.

When the mutable borrow occurs intraprocedurally (i.e. the source of the borrow is owned within the same function as its use), this can be done easily by marking whether or not each borrow was taken from a shared heap object upon creation, so that the analysis knows whether or not it was marked as such upon assignment. Note that this is transitive: for example, given `let b1 = &mut RRef::new(3);` which borrows from a shared heap object, a second borrow `let b2 = &mut b1;` would transitively also be a borrow from a shared heap object.

When the mutable borrow occurs interprocedurally (i.e. when a function's parameter is a mutable borrow), we can no longer determine whether or not the movement of `RRef<T>`s into such parameters would create nesting based only on the body of that function; in this case our analysis requires context-sensitivity to operate correctly. We extend our analysis to accommodate this via summary-based interprocedural analysis and additional code generation upon return from a call to such a function.

Such cases occur when multiple control paths through the code create the case when on some paths assignment has to be rewritten and on some not. In the example below, the function `f` is called from two invocation contexts, i.e. two functions `inside_of_an_rref` and `outside_of_an_rref`. On one invocation path the assignment of `r` inside `f` has to be marked as bottom and in another should not.

```
struct Foo {
   r: RRef<i32>,
}

// Moves an RRef into `t`
fn f(t: &mut Foo) {
  let r = RRef::new(3);
  t.r = r;
  // whether or not `t.r` need to be marked as a bottom
  // can only be deduced based on the caller's context
}

fn outside_of_an_rref() {
  let t = Foo { r: RRef::new(3) };
  f(&mut t);
  // in this context `t.r` in `f()` does not need to be
  // marked as bottom
}

fn inside_of_an_rref() {
  let rt = RRef::new(Foo { r: RRef::new(3) });
  f(&mut *rt);
  // in this context, `rt.r` in `f()` needs to be
  // marked as a bottom as it is nested
  mark_bottom(&mut rt.r);
}

fn transitively_ambiguous(t: &mut Foo) {
   f(t);
   // still ambiguous, so the ambiguity propagates further to
   // the caller of this function
}
```

We handle such cases context-sensitively. For a given function, and for each mutably-borrowed parameter, it emits a summary of fields within the parameter into which an `RRef<T>` was assigned within the function. At each call site, our analysis of the caller uses this information to generate code that marks all such `RRef<T>`s as bottom if, based on the caller context, the assignmend into these fields created nesting. In the example, this is true for `inside_of_an_rref`, but not for `outside_of_an_rref`.

Note that this applies transatively as well: if a caller context still cannot resolve this ambiguity and allow the analysis to know for certain whether or not the movement of `RRef<T>`s created nesting, the context-sensitive ambiguity is propagated further to the caller's callers.

***Dynamic dispatch*** Dynamic dispatch via trait objects presents a somewhat generalized case of interprocedural analysis. Because the actual method implementation that

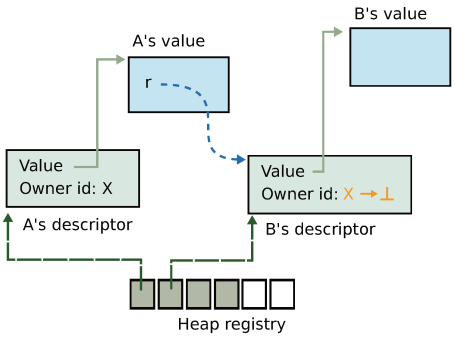


**Figure 2.** Internals of the `RRef<T>` type. `RRef<A>` contains a recursive refrence to `RRef<B>`. Both `RRef<T>`s are initially owned by domain X, but then ownership of `RRef<B>` is updated to bottom after assignment is done.

runs depends entirely on the type of the trait object at runtime, we must handle dynamic dispatch as the *union* of the result of interprocedural analysis performed on each individual implementation, because we cannot know statically which method implementation will run. Thus, our analysis must conservatively emit code that, upon return from the dynamically-dispatched method call, marks as bottom any `RRef<T>` that would become nested within *any* trait implementation. Note that because in RedLeaf the entire source is always available during compilation, we are able to conclusively detect which `RRef<T>`s this applies to across all possible trait implementations in the program.

### 3.1 Atomicity and Correctness in the Face of Crashes

To understand the logic of the cleanup algorithm in the face of crashes, consider the following sequence of operations that we inject around every assignment statement. Here we assign an `RRef<T>` `b` into another `RRef<T>`, `a.r`.

```
1    a.r = b;
2    mark_bottom(&mut a.r);
```

We first execute the assignment, then update `b`'s ownership to bottom.

To ensure atomicity of the above code in case of a crash (i.e. ensure that the shared heap is cleaned up correctly if the domain crashes between the assignment of the `b` and when it is marked as bottom), we develop a special implementation of the drop method.

Rust allows us to control behavior of the `drop()` method for the `RRef<T>` type. Internally each `RRef<T>` is represented as two separate types: descriptor and a value (Figure 2). The descriptor part contains the identifier of the owning domain, and an unsafe pointer to the value. The value itself may contain nested `RRef<T>`s. In our example, the value of `a` contains a reference to the descriptor part of `b`. The heap registry stores pointers to the descriptor parts of all `RRef<T>`s allocated on the shared heap and uses them during domain cleanup.

Rust allows us to overload the `drop()` method for the descriptor part of the `RRef<T>`, hence controlling when we deallocate

**Input:** $H$ is the set of all shared heap object descriptors, $D$ is the domain ID of the crashing domain
**Output:** None
$worklist \leftarrow \emptyset$
**for** $desc \in H$ **do**
  **if** $owner\_of(desc) = D$ **then**
    $rust\_drop(*desc.value)$
    $push(worklist, desc)$
  **end**
**end**
**for** $desc \in worklist$ **do**
  $free(desc)$
**end**

**Algorithm 1:** Shared heap cleanup

the value. The `drop()` implementation for the descriptor type is adjusted by moving the actual implementation (including explicitly dropping the pointed-to value) to a separate method `force_drop()`, and leaving `drop()` with the logical equivalent of:

```
impl<T> Drop for RRefDescriptor<T> {
   fn drop(&mut self) {
      if !crashed() || self.is_bottom() {
         // unconditionally drop the value
         self.force_drop();
      }
   }
}
```

Normally, when `RRef<T>` is going out of scope (such as during reassignment), the `drop()` method ignores the bottom flag (the `!crashed()` is true) and invokes the Rust's `drop()` method for the value type via the `force_drop()` function.

When a crash has occurred, however, the system performs the following two step algorithm (Algorithm 1). We first scan the heap registry for all `RRef<T>`s that are currently owned by the domain and *force drop* them via the `force_drop()` function. The `force_drop()` unconditionally drops the value of the `RRef<T>`, but leaves its descriptor part intact. When the value is dropped its `drop()` method triggers invocations of `drop()` methods for recursive `RRef<T>`s, i.e. it calls our overloaded `drop()` method of the descriptor part, which if the flags allow triggers recursive invocation of the `force_drop()`, hence dropping the value. After the force drop pass is done, we *free* as raw memory all descriptor parts that are owned by the crashing domain.

During the first pass, the `drop()` method obeys the bottom flag to avoid double free and leaks. For example, if the domain crashes between lines 1 and 2 in the listing above, both `a` and `b` are reachable from the heap registry (both are still reported as owned by the crashing domain), but the `drop()` method for `b` does not drop the value as the bottom flag is not set (logically, the bottom flag is the presence of a "no domain" owner). Note, it is possible that `b` will be dropped before `a`. The drop method of `a` will try to recursively drop `b` (if assignment in line 2 is already performed) calling the `drop()` method for its descriptor part. However, since the descriptor part is not deallocated by the `force_drop()` method, the descriptor part is still on the heap and the invocation of the `b`'s `drop()` method is safe.

If the domain crashes after marking `b` as the bottom (e.g., after line 2), `b` is no longer owned by the crashing domain and hence it will not be reached from the heap registry during the scan and will not be force dropped. Instead, `a`'s `drop()` method will recursively invoke the `drop()` method for `b` which in turn drop the value since the bottom flag is set.

During the second pass we drop descriptors as raw memory. This operation is correct since the descriptor, whose value pointer no longer refers to anything, is simply plain-old-data. At this point no other value can refer to any of the descriptors.

Similarly, when `RRef<T>` is being moved out of another's tree, the operation order is exactly reversed:

```
1    mark_owned(&mut a.r, current_domain());
2    let b = a.r;
```

We first mark `RRef<T>` as owned by the current domain and then move it from the hierarchy. If domain crashes in between the two operations, we have the case identical to the above, i.e., the `RRef<T>`, `a.r`, is reachable recursively through `a`, but at the same time is owned by the current domain.

# 4 Implementation

We implement our assignment analysis as a mid-level intermediate representation (MIR) pass. MIR [50], a recent addition to the Rust compilation infrastructure, is a control flow graph-based, simplified form of Rust, whose main benefit is being better suited for dataflow analysis (e.g. type and borrow checking), as opposed to the tree-based high-level intermediate representation (HIR). Assignment statements in MIR are made of *Places*: an expression that identifies a location in memory and *Rvalues*: an expression that produces a value (e.g., `let <place> = <rvalue>;`).

For each function in the program, the MIR pass iterates through all assignment statements in the control-flow graph. Analyzed `RRef<T>` assignments are then collected into a list of *insertion points* at which ownership updates will be inserted.

***Code generation*** In MIR, a function call is represented as a basic block terminator that connects two basic blocks. In order to insert calls to the trusted runtime, basic blocks enclosing the insertion points need to be split. For each insertion point, the pass creates a new basic block and moves subsequent statements to it. The two basic blocks are then joined with the `Call` terminator pointing to the desired function in the runtime (e.g., `mark_bottom()`).

# 5 Evaluation

We run our experiments on CloudLab [53] c220g5 servers configured with two Intel Xeon Silver 4114 10-core Skylake CPUs running at 2.20 GHz, 192 GB RAM, and a dual-port Intel X520 10GbE NIC. The machines run NixOS 23.05 with the Linux 6.1 kernel configured without any speculative execution attack mitigations (`mitigations=off`) reflecting the trend of recent Intel CPUs addressing a range of speculative execution attacks in hardware. In all the experiments, we disable hyper-threading, turbo boost, CPU idle states, and frequency scaling to reduce the variance in benchmarking.

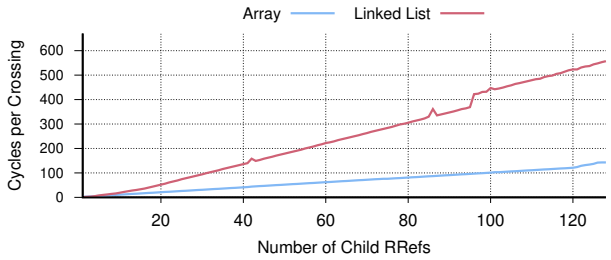


**Figure 3.** Overheads of Ownership Tracking

## 5.1 `RRef<T>` Passing

To put into perspective the performance improvement enabled by assignment analysis, we construct a synthetic test in which nested `RRef<T>`s of common data structures are passed between domains (Figure 3). For all invocations, a proxy responsible for updating the ownership of all `RRef<T>`s passed in the invocation is interposed between the caller and callee domains.

The naive implementation of the proxy traverses the `RRef<T>` hierarchy passed by the calling domain and assigns ownerships for all objects in the closure recursively. In contrast, assignment analysis only manipulates the ownerships of top-level `RRef<T>`s. We run two experiments. In the first experiment we pass an array of `RRef<T>`s that is reachable from a root `RRef<T>`. We vary the size of the array from 1 to 128 `RRef<T>` elements. In the second test, we pass a linked list of `RRef<T>`s and again vary the size of the list from 1 to 128.

Overall, the naive proxy incurs overheads as the number of recursive `RRef<T>`s increases. Passing a linked list incurs an overhead of 5 cycles per node due to pointer chasing, whereas passing a simple fixed-size array of `RRef<T>`s costs an average of 1.1 cycles per element.

## 5.2 Network Functions

We devise an application test to measure the performance improvement in the context of network function virtualization (NFV). We implement a packet processing pipeline similar to Netbricks [47], but with support for isolation of heaps, and hence support for fault isolation of individual network functions. The pipeline uses DPDK [16] to send and receive packets. The received packets are passed through a series of Rust network functions that are implemented as isolated domains which exchange packets on the shared exchange heap. We test three configurations: non-isolated Rust, i.e., Rust that uses regular references and function call invocations, a proxy that recursively updates ownership of all objects passed on the shared heap, and our new system that updates ownership of only the root element. We vary the packet batch size passed between NFs from 1 to 32.

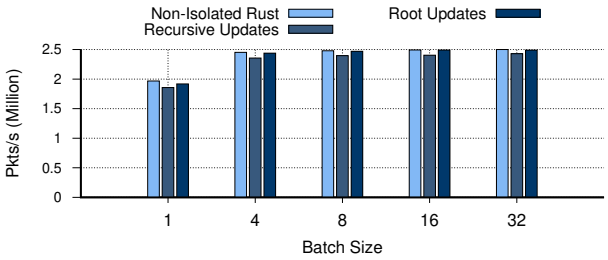


**Figure 4.** Throughput on varying batch sizes

The network functions encompass tasks commonly done in networking equipment: TTL decrementer, NAT, ACL firewall, and Maglev load balancer [19]. The NAT function rewrites the source IP and port of outgoing packets based on a dynamic mapping. Ports are allocated and recorded in a hash table for each new flow. In Maglev, consistent hashing is utilized to evenly distribute flows across all servers. For each new flow, Maglev selects a backend server via a hash table lookup. The size of the *lookup table* is proportional to the number of backend servers. The selected backend is then inserted into the *flow tracking table*, to ensure that the same servers are chosen for established flows even when the set of available servers have changed.

On average, the naive proxy results in 4% lower throughput than normal Rust while the improved proxy with assignment analysis has an overhead of only 1%. Arguably, the overheads of this highly-optimized scenario that passes at most 32 recursive `RRef<T>`s are small. Other operating system interfaces might be configured to pass a significantly larger number of buffers on the shared heap. For example, large disk accesses might pass hundreds or even thousands of buffers.

## 6 Conclusions

Despite active development, Rust is still lacking mechanisms to support safe systems that require process-like isolation guarantees and efficient zero-copy communication. Our work develops a static analysis pass inside the Rust compiler that allows us to reason about assignments of references on a special exchange heap and correctly account for the ownership of objects on the exchange heap. While seemingly a niche problem, we argue it is an important piece of a puzzle that can enable fast, secure and reliable systems that utilize language mechanisms for isolation of untrusted subsystems.

## Acknowledgments

We would like to thank PLOS'23 reviewers for various insights helping us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers NSF numbers 2313411, 1837127 and 2341138.

## References

[1] WebAssembly Specification. https://webassembly.github.io/spec/core/.

[2] OpenArena Live. https://openarena.live, 2019.

[3] Akamai. Serverless Computing with Akamai Edge Workers. https://www.akamai.com/products/serverless-computing-edgeworkers, 2015.

[4] Alexander Gallego. Redpanda Wasm engine architecture. https://redpanda.com/blog/wasm-architecture, 2021.

[5] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. XOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS'12, pages 49–60, New York, NY, USA, 2012.

[6] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[7] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, pages 156–161, 2017.

[8] Fred Barnes, Christian Jacobsen, and Brian Vinter. RMoX: A Raw-Metal occam Experiment. In *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 182–196, September 2003.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–283, 1995.

[10] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The Development of the Emerald Programming Language. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*, pages 11–1–11–51, 2007.

[11] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19, November 2020.

[12] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[13] Hank Bromley and Richard Lamson. *LISP Lore: A Guide to Programming the Lisp Machine*. Springer Science & Business Media, 2012.

[14] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N Bairavasundaram, Kaladhar Voruganti, and Garth R Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC'09)*, pages 25–25, 2009.

[15] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1988.

[16] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[17] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[18] Dylan Schiemann. Zoom on Web: WebAssembly SIMD, WebTransport, and WebCodecs. https://www.infoq.com/news/2020/08/zoom-web-chrome-apis, 2020.

[19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 523–535, March 2016.

[20] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and*

*Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[21] Evan Wallace. WebAssembly cut Figma's load time by 3x. https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/, 2017.

[22] Feske, N. and Helmuth, C. Design of the Bastei OS architecture. 2007.

[23] Linux FUSE (filesystem in userspace). https://github.com/libfuse/libfuse.

[24] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.

[25] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 193–206, 2003.

[26] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. ACM, 2000.

[27] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation. *Addison-Wesley Longman Publishing Co., Inc.*, 1983.

[28] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC'02)*, pages 45–58, 2002.

[29] Hardy, N. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[30] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[31] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[32] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, page 22. ACM, 2004.

[33] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. MSwitch: A Highly-Scalable, Modular Software Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR'15, New York, NY, USA, 2015.

[34] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[35] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 445–458, Seattle, WA, April 2014.

[36] Intel Corporation. Storage Performance Development Kit (SPDK). https://spdk.io.

[37] Jordon Mears. How we're bringing Google Earth to the web. https://web.dev/earth-webassembly/, 2019.

[38] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers, 2018.

[39] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 627–643, Carlsbad, CA, October 2018.

[40] Zhaofeng Li, Tianjiao Huang, Vikram Narayanan, and Anton Burtsev. Understanding the overheads of hardware and language-based ipc mechanisms. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*, 2021.

[41] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 429–444, April 2014.

[42] Peter W Madany, Susan Keohan, Douglas Kramer, and Tom Saulpaugh. JavaOS: A Standalone Java Environment. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[43] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 459–473, Seattle, WA, April 2014.

[44] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 699–716, 2020.

[45] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 21–39, November 2020.

[46] Nathan Froyd. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly.

[47] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 203–216, Savannah, GA, November 2016.

[48] Pat Hickey. Lucet Takes WebAssembly Beyond the Browser. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime., 2019.

[49] The Rust Project. References and borrowing. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html. Accessed: 2023-08-05.

[50] The Rust Project. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html, 2016.

[51] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 13–24, San Jose, CA, June 2013.

[52] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, 1980.

[53] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[54] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 323–336, San Jose, CA, April 2012.

[55] Mark Sullivan and Michael Stonebraker. Using Write Protected Data Structures To Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB'91, pages 171–180, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[56] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European Workshop*, pages 102–107, 2002.

[57] Daniel C Swinehart, Polle T Zellweger, Richard J Beach, and Robert B Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):419–490, 1986.

[58] The Istio Project. WebAssembly in the Istio Proxy (Envoy). https://istio.io/latest/docs/concepts/wasm/.

[59] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[60] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.