DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency

Haoran Ma[†⋆] Yifan Qiao[†] Shi Liu[†⋆] Shan Yu[†] Yuanjiang Ni[Ψ] Qingda Lu[Ψ] Jiesheng Wu[Ψ]
Yiying Zhang[‡] Miryung Kim[†] Harry Xu[†]

UCLA[†] *UCSD*[‡] *Alibaba Group*[Ψ]

Abstract

Despite being a powerful concept, distributed shared memory (DSM) has not been made practical due to the extensive synchronization needed between servers to implement memory coherence. This paper shows a practical DSM implementation based on the insight that the ownership model embedded in programming languages such as Rust automatically constrains the order of read and write, providing opportunities for significantly simplifying the coherence implementation if the ownership semantics can be exposed to and leveraged by the runtime. This paper discusses the design and implementation of DRust, a Rust-based DSM system that outperforms the two state-of-the-art DSM systems GAM and Grappa by up to **2.64**× and **29.16**× in throughput, and scales much better with the number of servers.

1 Introduction

The concept of distributed shared memory (DSM) received significant attention during the early years of distributed computing systems. This era witnessed a plethora of pioneering efforts, as exemplified by seminal works such as [10, 16–18, 31, 36, 49, 50, 56, 61–63, 80]. DSM offers the power of parallel computing using multiple processors and machines and, more crucially, streamlines the development of distributed applications with a unified, contiguous memory view.

The initial enthusiasm for DSM was tempered by significant performance bottlenecks, primarily due to the low network speeds prevalent during its nascent stages. Recent advances in hardware and networking technologies [3, 7, 12, 19, 23, 29, 33, 38, 40, 42, 46, 51, 54, 64, 66, 74, 78] have revitalized the DSM explorations. Several new DSM systems [14, 45, 60, 77, 81, 88] were proposed in recent years to take advantage of these enhanced networks. However, these systems are still far from achieving satisfactory performance, exhibiting poor scalability and substantial slowdown compared to their single-machine counterparts. This is mainly due to the intensive synchronization operations needed to ensure memory coherence across servers.

State of the art. The majority of existing DSM systems [6, 14, 45, 88] adopt an approach to achieve data consistency by adhering to the following invariant: for each data block to be accessed, the block is either located on a single node with potential read and write access, or it is replicated across multiple nodes with each having read access only. Prior to a server attempting to access a block, a DSM system checks the state of the block, invalidates copies of that block on all other servers, and then transmits the block to the requesting server. This synchronization process necessitates multiple network round trips. Even with RDMA, the incurred latency is still orders of magnitude higher compared to a single local access, significantly degrading overall performance. Effectively reducing the number of synchronizations is, therefore, crucial for minimizing DSM overhead and rendering it feasible for real-world deployment.

A practical strategy to minimize synchronization overhead involves implementing high-level protocols to guarantee exclusive access for each server. For instance, Apache Spark [91] utilizes an immutable data structure known as a resilient distributed dataset (RDD) for distributed access. However, RDD only facilitates coarse-grained distributed access, limiting each server to accessing a distinct partition of an RDD. While increasing access granularity enhances performance, it comes at the expense of reduced generality—Spark is tailored for bulk processing of batch data and is incapable of supporting distributed applications requiring object-level accesses, such as social networks where objects of various types and sizes (*e.g.*, images, connections, *etc.*) are created and manipulated upon each user request.

Insights. Our main observation is that synchronization overheads in existing DSM systems are introduced primarily due to the use of a generic approach that overlooks semantic information from programs. For example, many real-world concurrent programs are engineered with a single-writer-multiple-reader (SWMR) discipline to ensure correctness during concurrent operations. Leveraging such information

⋆ Part of the work was done when Haoran Ma and Shi Liu interned at Alibaba Group.

can potentially eliminate the need to check the state of remote data blocks before accessing them, leading to dramatically improved performance. A major challenge is, however, how to expose such semantics in a sensible way so that the DSM system can see and act upon it.

One approach to convey such semantics, as demonstrated by AIFM [73] and Midas [68], involves exposing APIs that developers can invoke to specify program regions accessible only by a single writer. However, this process is cumbersome and error-prone, demanding a profound understanding of potential executions and involving substantial program writing. Our key insight in this endeavor is that the SWMR programming paradigm aligns seamlessly with *ownership types*, which have already been integrated into programming languages like Rust [75]. Rust is widely employed in the system community for dependable and secure implementation of low-level systems code.

Rust's ownership type inherently upholds SWMR properties in any compiled Rust program. The fundamental concept behind the ownership type is that each value is ensured to have a single unique variable as its owner throughout the execution. While multiple references to a value are allowed, only the owner and mutable references can modify the value. Moreover, only one of these references is permitted to be used for modifying the value at any given point.

When developing a DSM system on top of an ownership-based language like Rust, SWMR semantics are inherently embedded in any Rust program *by design*. Effortlessly extracting such information becomes possible with basic compiler support, sparing developers from the need for code rewriting. Utilizing the SWMR semantics from the program leads to a considerably simplified process for accessing data in DSM. In the case of a write access, the ownership type ensures exclusive access to the data. Consequently, DRust can move the data to the requesting machine, performing the write there without explicitly invalidating its copies on other machines. In the case of a read access, data can be efficiently replicated to (and cached in) each requesting machine, benefiting from the compiler-provided assurance of freedom from concurrent writes.

This paper presents DRust, an efficient Rust-based DSM implementation that enables object-level concurrent accesses by leveraging the SWMR semantics made explicit by Rust's ownership type. DRust automatically turns a single-machine Rust program into a DSM-based distributed version *without requiring code rewriting*. While extracting the ownership semantics appears straightforward, leveraging it to implement a distributed coherence protocol correctly and efficiently presents two main challenges.

The first challenge is *how to manage memory correctly and efficiently.* Rust's ownership type system is inherently designed for a single-machine environment, where the memory address of an object remains constant post-creation. This assumption is disrupted in a distributed environment, where objects may be migrated or duplicated on different machines. Such actions can lead to the risk of dangling pointers, potentially breaking memory coherence.

To tackle these issues, DRust builds a global heap spanning multiple servers based on the idea of partitioned global address space [21]. Each object in the heap has a unique global address in the address space, which can be used for accessing the object from any server. DRust re-implements Rust's memory management constructs to allocate objects in the global heap. Given that a server can have cached objects (to accelerate reads), DRust carefully crafts an ownership-based cache coherence protocol upon the global heap abstraction to achieve both memory coherence and efficiency (§4.1.1).

In a nutshell, our coherence protocol leverages the ownership semantics to eliminate the need for explicit cache invalidation. It allows multiple readers to fetch a copy of the object from its host server and cache it, but disallows any change to the global address and the value of the object. When a write access occurs, it must first borrow the ownership, at which point DRust moves the object in the global heap to a new address on the server issuing the write. The address change of the object automatically invalidates cache copies that use the stale address and triggers the subsequent readers to update the cache by fetching the object from its latest address.

The second challenge is *how to support transparency in programming.* Rust's standard libraries and programs were originally built for running on a single machine, and they cannot deal with distributed resources in a cluster. For example, a Rust program running on server A cannot spawn a thread on another server B, let alone synchronize threads between A and B. To enable a Rust program to run *as is* under DRust, we provide distributed threading utilities by restructuring critical elements of the Rust standard library, including threading, communication channels, and shared-state locks (§4.1.2). Our adapted libraries offer the same interfaces, making them compatible with single-machine Rust programs, but internally invoke our distributed scheduler, which determines where to run the thread and facilitates cross-server synchronization. We built them atop the ownership-based memory model, enabling the DRust runtime to safely pass references of objects between threads and automatically fetch the value from the global heap upon dereferencing.

With our programming abstractions, a Rust application can start on a single server and gradually spawn its threads to other servers. Under the hood, DRust employs a runtime to manage distributed physical compute and memory resources for the application. The runtime runs as a process on each node in the cluster, and they work cooperatively for cross-server memory allocation and thread scheduling. The runtime prioritizes the current server for object allocation and thread creation, but it will schedule the resource allocation request to another server under memory pressure (§4.2.1). To make cluster-wise decisions such as deciding the target server for global memory allocation and thread creation, DRust

has a global controller that is launched together with the application. The global controller communicates with DRust runtime on each node to collect resource usage information and applies adaptive policies to achieve load balance (§4.2.2).

Results. We evaluated our system on four real-world applications in an eight-node cluster. Our evaluation demonstrated an average of 2.02× and 9.48× (up to 2.64× and 29.16×) speedup compared with two state-of-the-art DSM systems GAM and Grappa, respectively. Furthermore, DRust incurred a mere 2.42% slowdown compared to the original Rust program on a single machine with sufficient resources. DRust is available at https://github.com/uclasystem/DRust.

2 Background in Ownership

Over the past decades, numerous programming languages have been designed to provide safe memory management and data sharing. At the core of such a design is often a tradeoff between memory abstraction level and management efficiency. The ownership concept, and the Rust programming language built upon, are considered promising solutions that achieve a sweet spot between abstraction and efficiency. This section provides an overview of these techniques and explains how ownership can benefit DSM implementations.

Ownership Type. The ownership model has a long history in pursuit of memory-safe language designs and type systems [8, 9, 27, 43, 58, 83]. It has also inspired many systems for safe and efficient resource management [13, 41, 59, 89]. At a high level, ownership enhances a language's type system in a way that guarantees the memory and thread safety of a program with type checking done at compile time. The ownership model encompasses a range of concepts, among which the most important are *lifetimes* and *borrowing*.

An ownership-based type system uses lifetimes to control the allocation/deallocation of objects. It enforces that each object must have one and only one owner at a time. This allows the compiler to statically track an object's lifetime via its owner, and immediately deallocate the object once its owner goes out of scope, preventing memory leaks without using garbage collection that can introduce disruptive pauses to program execution.

To access an object, a program can create a reference from its owner, but the reference must "borrow" the permission from the owner, and "return" it to the owner after the access. Specifically, the type system allows the creation of multiple *immutable references* to an object from its owner for concurrent reads but prohibits any write with these references. It allows only one mutable reference to the object only when no other (mutable or immutable) references exist. Through borrowing, the ownership type disallows simultaneous writers and hence prevents data races. In addition, references must return the borrowed permission when they go out of scope. For any program that demonstrates type soundness, the type checker guarantees that references to an object can only reside within the object's lifetime; the object can be

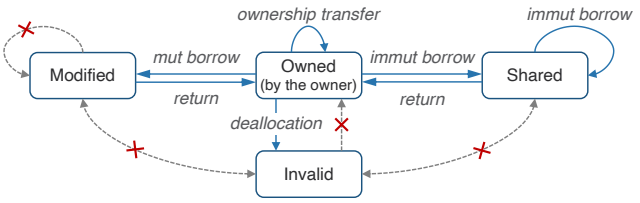


Figure 1: State machine for Rust's ownership-based memory model.

safely and automatically deallocated when its owner goes out of scope, by which time it has already lost all its references.

Finally, ownership can be transferred from one owner to another—*e.g.*, at a function call, the creation of a thread, or message passing (*i.e.*, via `channel`). However, the type system enforces that ownership transfer must occur in the absence of "borrowing". In other words, no other references can exist in scope when transferring the ownership, preventing data races during ownership transfers.

The guarantees provided by the ownership model with respect to object lifetime and data sharing can be summarized with the following four invariants:

1. **Singular Owner**: each value has one single owner at any time (which must also belong to one single thread).
2. **Safe Borrowing**: All references are created from the owner; permission borrowing and returning guarantees that references that can be used to access the object must be valid.
3. **Single Writer**: Each object allows one mutable reference at most, and it cannot coexist with any other references in the same scope.
4. **Multiple Reader**: Multiple references are permitted only when all of them are immutable.

The last two invariants are commonly called the single-writer-multiple-reader (SWMR) property in the DSM literature [57].

Rust Language. Rust offers a practical implementation of ownership and is designed with a range of zero-cost abstractions for efficient fine-grained resource management. Figure 1 depicts the state machine for Rust's ownership-based memory model. At a high level, this model restricts that the owner is always in the O (owned) state, and transitions between M (modified), S (shared), and I (invalid) must go through the O state[1]. Clearly, a distributed implementation of this approach avoids broadcasts or snooping, and only requires peer-to-peer message passing.

Listing 1 exemplifies a simple accumulator implemented in Rust (Lines 1–7). The `Accumulator` struct keeps an integer `val` and exposes an interface `add` to increment the value. Rust uses a smart pointer type `Box<T>` to store values on the heap; this pointer serves as the initial owner of the referenced value, as shown in Line 10 and 11. Line 13 instantiates `Accumulator a`, where the ownership is implicitly transferred from `val` to `a.val` during its initialization. Rust allows the creation of mutable and immutable references to access the value. For

[1] A transition from M to S is also possible as an optimization in Rust.

```rust
pub struct Accumulator { pub val: Box<i32>, }
impl Accumulator {
  pub fn add(&mut self, delta: &i32)->i32 {
    *self.val += *delta;
    *self.val
  }
}
fn main() {
  // Allocates two integers in the heap.
  let val: Box<i32> = Box::new(5); // val is an owner.
  let mut b: Box<i32> = Box::new(0); // b is an owner.
  // Ownership is transferred from val to a.val
  let mut a = Accumulator{val};
  { // Only one mutable reference is allowed.
    let mutr: &mut i32 = &mut *b;
    // No other reference is allowed now.
    /* let another_r = &*b; */ // COMPILE ERROR!
    *mutr = 10; // b == 10
  }
  { // Multiple immutable references are allowed.
    let (b_r1, b_r2): (&i32, &i32) = (&*b, &*b);
    // Mutable reference is prohibited now.
    /* let b_mutr = &mut *b; */ // COMPILE ERROR!
    // Passing by references won't transfer ownership.
    let sync_add = a.add(b_r1); // a.val == 15
    let sync_add = a.add(b_r2); // a.val == 25
  }
  {// Ownership of a and b is moved to the new thread.
    // No reference should or can borrow a or b now.
    let async_add = thread::spawn(move ||
      a.add(&*b) // a.val == 35
    ).join(); // lifetime of a and b ends
    // Current thread cannot access a and b anymore.
    /* println!("{}", a.val); */ // COMPILE ERROR!
  }
}
```

Listing 1: A simple accumulator implementation in Rust.

example, Lines 14–19 create a singular mutable reference (`&mut`) to `b` and set its value to 10. Similarly, Lines 20–27 create two immutable references (`&`) to `b` and add them to `a` via two function calls. Note that passing references as arguments in function calls does not transfer their ownership.

Finally, Rust allows spawning new threads for concurrent programming, as shown in Lines 28–35. A new thread is created via `thread::spawn`, where the use of `move` captures `a` and `b` in the current scope and transfers their ownership to the newly spawned thread. Rust performs shallow copying for inter-thread communication, where only the pointers stored in `a` and `b` are transferred to the child thread while the actual values on the heap are not moved. Rust guarantees memory safety of `a` and `b` by tracking their ownership. At Line 32, when the child thread finishes its closure (*i.e.*, not necessarily after `join`), and `a` and `b` exit the scope (to which their ownership belongs), their lifetimes terminate and Rust deallocates them from the heap.

3 Motivation

DSM was proposed to eliminate the barrier of distributed programming by offering the same memory consistency model as single-machine shared memory. The core of its design is a software-based cache coherence protocol, which mimics a hardware-based approach on multi-core CPUs and synchronizes memory states on different servers by sending control messages between them. However, it is notoriously hard to implement cache coherence efficiently at the software level due to the high communication latency between physically disjointed servers.

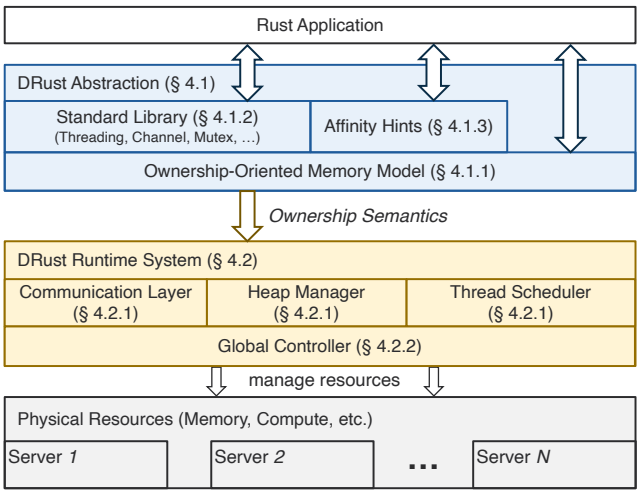


Figure 2: Design overview of DRust.

High Synchronization Overheads for Coherence. To gain a high-level understanding of how much improvement can be achieved by improving the cache coherence protocol, we performed an analysis by running a real-world application DataFrame [67] with a state-of-the-art DSM system GAM [14] with a fast network. We first ran Dataframe on a single server with 16 CPU cores and 64GB memory. We then ran it with GAM on eight servers connected by a 40Gbps Infiniband network by evenly distributing the same amount of resources to eight servers (*i.e.*, each server uses 2 CPU cores and 8GB memory). Our experiments show a **2.4**× slowdown when Dataframe runs on eight servers.

A detailed examination reveals that such a slowdown stems primarily from its complicated coherence protocol. GAM runs a directory-based protocol, which assigns each DSM cache block a home node. Upon each object read/write, the home node tracks the state of its cache block and updates all cache copies for the state change, incurring extensive computation and network overhead. We broke down the average time spent on each component when accessing one object in the DSM. Reading a 512-byte (*i.e.*, GAM's default cache block size) uncached object in GAM takes 16μs, while the actual time to read the object over the network is only 3.6μs. In other words, maintaining cache coherence takes **77%** of the total time. This large memory access overhead significantly increases operation latency, hindering the practical deployment of distributed shared memory. With the single writer invariant inherent in the ownership model, we expect that most of this overhead can be eliminated, leading to significant ($>2\times$) speedups for each access.

4 Design

DRust is an efficient DSM framework atop the Rust programming language. As shown in Figure 2, it consists of

```
1  // Unmodified Rust code.
2  pub struct Accumulator { pub val: Box<i32>, }
3  impl Accumulator {
4    pub fn add(&mut self, delta: &i32)->i32 {
5      *self.val += *delta;
6      *self.val
7    }
8  }
9  fn main() {
10   // Allocates two integers in the distributed heap.
11   let val: Box<i32> = Box::new(5);
12   let b: Box<i32> = Box::new(10);
13   let mut a = Accumulator{val};
14   // a.val and b will be fetched to local.
15   let local_add = a.add(&*b); // a.val == 15
16   // Only refs to a and b are shipped to remote.
17   let remote_add = thread::spawn(move ||
18     a.add(&*b)).join(); // a.val == 25
19 }
```

Listing 2: DRust seamlessly transforms an unmodified accumulator implemented in Rust into a distributed version.

Rust-based programming abstractions for DSM (§4.1) and a runtime (§4.2) that manages distributed physical resources.

DRust is compatible with standard Rust. Listing 2 illustrates how the accumulator (shown in Listing 1) runs on DRust distributively without requiring code rewriting. The program starts running on a single machine A and the DRust runtime gradually allocates its memory and spawns new threads on different machines. Specifically, Lines 10–13 create `Accumulator a` and `b` where `a.val` and `b` are in the global heap. We use a global allocator to allocate objects in the global address space and hence these objects may be allocated on a different server. Line 15 synchronously adds `b` to `a` by fetching both values `a.val` and `b` to A's local memory (if they are allocated somewhere else). Line 17 spawns a new thread and ships the function closure to perform `add` asynchronously. This thread will be scheduled on a different server B if A's compute power has been saturated. In this case, DRust performs shallow copying and only ships the pointers stored in `a` and `b` to B without actually moving objects in the global heap. The newly-created thread relies on the DRust runtime to detect data locations and fetch objects upon dereferencing.

4.1 DRust Programming Abstraction

DRust provides each thread with a local stack and abstracts distributed memory as a shared global heap. Each server allocates thread stacks and backs one partition of the global heap with its physical memory. DRust re-implemented core memory management constructs including `Box`, `&`, and `&mut` for transparent heap access. This approach hides the complex details of memory allocation/deallocation, moving objects, and coherence maintenance (§4.1.1). DRust supports distributed threading and synchronization by adapting Rust's standard libraries atop the core language constructs (§4.1.2). Furthermore, DRust offers affinity annotations that allow developers to build more efficient applications by expressing data affinity semantics (§4.1.3).

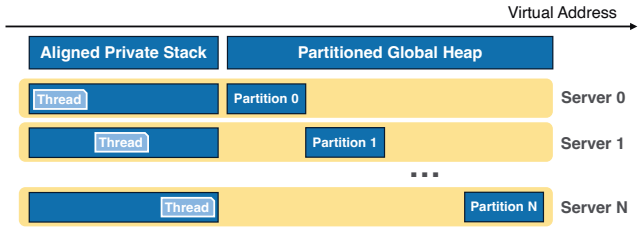


Figure 3: The address space layout of DRust. The stack is private to each thread but they share an aligned address space to ease migration, while the heap is globally shared and partitioned across servers.

4.1.1 Memory Management

Next, we discuss how DRust (re)implements the memory-related language constructs in Rust to achieve memory safety and memory coherence.

Address Space. As shown in Figure 3, DRust maintains an identical address space layout on all servers. It exposes distributed memory as a coherent shared heap to applications. Embracing the idea of partitioned global address space (PGAS) [21], it partitions the heap space and assigns each server a unique address range. The stack, in contrast, is private to each thread. However, DRust aligns the stack space on each server and pads stacks to avoid overlapping. This streamlines thread migration between servers as it allows a thread to keep its private stack address unchanged when being moved.

Coherence Protocol in a Nutshell. For efficiency, DRust employs a *call-by-reference* model for newly created threads. Upon creation of a thread, the DRust runtime only passes references or `Box` pointers to objects to the newly created thread. Upon dereferencing, objects are fetched to the server where the thread is executed.

When a *read access* of an object is issued on a server, our runtime simply fetches a copy of the object from its hosting server and places it in its *local cache*. As a result, multiple copies of the same object may exist on different servers. This allows multiple servers to read the object at the same time from their respective cached copies. Fetching a copy of the object for read does not change the object's address in the global space. When a *write access* occurs on an object, the server issuing the write must first obtain the object's write access permission through a *mutable borrow*. Our reimplementation of mutable borrow (discussed shortly) *moves*[2] *the object in the global heap to a new address* that belongs to that server. In doing so, the object's cached copies on other servers are automatically invalidated without sending explicit invalidation messages—subsequent reads on these servers must obtain an immutable reference to the object through an immutable borrow from its owner pointer, which has been updated to the new address immediately after

[2]The term "copy" is used to describe the process of adding an object into the cache without changing its global address. The term "move" means relocating the object into a server's heap partition, which requires changing its global address.

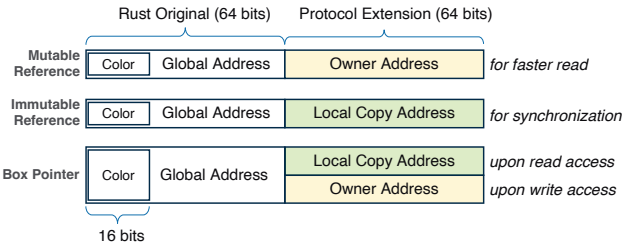


Figure 4: DRust repurposes Rust pointers and references to contain a global heap address and an extension field for its coherence protocol.

the mutable borrow returns. Upon identifying the owner's address change, each immutable borrow would direct a server to fetch a fresh version of the object from the new address as opposed to relying on a stale copy residing in its cache.

Note that this is a general protocol that covers the case that the object is on the same server that issues the write—as long as the server moves the object into a different location in the global heap, no other servers can read the stale copies of the object. However, this is not efficient as each local write requires moving the object to a new address. To address this inefficiency, DRust employs a pointer-coloring technique, inspired by the designs of many concurrent garbage collectors [1, 52]. Discussed at the end of this subsection, this technique offers a more efficient solution for handling local writes.

Pointer Layout. In order to support this protocol, each pointer must remember not only the object's global address, but also the address of the cached copy in a server's local cache (to avoid redundant remote fetches). As such, we modify Rust's pointer structure, as illustrated in Figure 4. DRust internally extends each Rust `Box` pointer and reference with an additional 64-bit field, which is used differently for read and write access. At a high level, the field records the address of the cached copy for faster read accesses; for write accesses, this field records the address of the object's owner for post-write synchronization. Additionally, DRust reserves the highest 16 bits in the global address field as "color" bits. These bits record the version number of the pointer and play a crucial role in DRust's efficient handling of local writes.

Next, we discuss how DRust reimplements Rust's ownership operations to realize the distributed coherence protocol. For ease of presentation, this subsection focuses on a simplified version of the protocol. A complete coherence protocol and its proof of memory coherence are available in [53].

Mutable Borrow. Mutable borrow creates a mutable reference that holds exclusive access to the referenced object for writing. Algorithm 1 outlines the procedures for both dereferencing and dropping a mutable reference. When performing dereferencing, DRust first checks the object's location (Line 2) and performs direct access if the object's address belongs to the heap partition of the machine A that executes the access. Otherwise, DRust *moves* it to A's heap partition (as opposed to caching it) (Line 3). The move, conducted in the following three steps, changes the object's

Algorithm 1: Access logic for mutable references.

Input: A mutable reference m containing a global address $m.g$ and the owner address $m.o$.
Output: A local memory address to be written to.

```
1 Function DerefMut(m):
2     if ¬IsLocal(m.g) then
3         m.g ← Move(ClearColor(m.g))
4     return ClearColor(m.g)
5 Function DropMutRef(m):
6     c' ← GetColor(m.g) +1
7     Write(m.o, AppendColor(m.g, c'))
```

global address. DRust (1) copies the object into A's heap at an address p, (2) updates the mutable reference with the address p, and (3) asynchronously requests the remote server that previously stored the object to deallocate the original object.

A challenge arises with its original owner `Box`, which now becomes a dangling pointer, pointing to an invalid memory location. Fortunately, the integrity of the system is maintained by the single-writer invariant (referenced as Invariant 3). This invariant ensures that while the mutable reference remains alive, no other entity, including the original owner, can access the data. To ensure correctness, when this new reference is dropped, DRust synchronously updates the original owner `Box`, redirecting it to the new address p (Line 7). As a result, the original owner always possesses the latest view of the object. Additionally, all modifications made through this mutable reference are visible in all subsequent accesses, as they necessitate borrowing permission from the updated owner `Box`. The single-writer invariant also eliminates the possibility of simultaneous updates to the owner, ensuring that updating the owner is free from concurrency issues.

Immutable Borrow. Immutable borrowing allows concurrent reads to the same object from immutable references on the same or different servers. As detailed in Algorithm 2, DRust handles the dereferencing of immutable references by first checking the object's location (Line 2). For remote objects, DRust creates a local copy in the *per-node read-only "cache"* and records its local address in the reference's extension field (see Figure 4). This preserves the original global address of the object, ensuring that any new immutable reference—whether it is derived from the owner `Box` or from another immutable reference—can always access the original object from the global heap.

As opposed to being a separate memory space, our "cache" provides a "virtual" aggregation of all local copies maintained on each server. These copies reside in the regular heap, managed by a per-node hashmap H. This hashmap maps each global address to a pair of its local address and the number of local immutable references to the local copy. To prevent redundant copies of an object on the same server, DRust checks the hashmap H before creating a new local copy

Algorithm 2: Access logic for immutable reference.

Input: A shared immutable reference r containing a global address $r.g$ and a local copy address $r.l$, and a local cache hashmap H.
Output: A local memory address for reading.

```
1  Function DEREF(r,H):
2      if ISLOCAL(r.g) then
3          return CLEARCOLOR(r.g)
4      else
5          if r.l = Null then
6              ATOMIC {
7              if r.g ∈ H then
8                  ⟨l', cnt⟩ ← GETENTRY(H, r.g)
9                  r.l ← l'
10                 UPDATEENTRY(H, r.g, ⟨l', cnt+1⟩)
11             else
12                 r.l ← COPY(CLEARCOLOR(r.g))
13                 INSERTENTRY(H, r.g, ⟨r.l, 1⟩)
14             }
15         return r.l
16 Function DROPREF(r,H):
17     if r.l ≠ Null then
18         ATOMIC {
19         ⟨l', cnt⟩ ← GETENTRY(H, r.g)
20         UPDATEENTRY(H, r.g, ⟨l', cnt−1⟩)
21         }
```

(Line 7). If a local copy is already present, DRust increments its reference count in H and updates the extension field in the immutable reference to point to this copy (Lines 8–10). If no existing copy is found, a new one is created (Lines 12–13). Since the hashmap uses objects' global addresses as keys, if an object has been modified by another server since its last read, its global address must have changed, making cache lookup fail even if a (stale) local copy exists.

DRust actively updates the reference count of each local copy when an immutable reference is either dereferenced or dropped, as outlined in Lines 10 and 20. Utilizing these counts, the DRust runtime periodically scans the "cache" and lazily reclaims unreferenced copies (*i.e.*, those with a zero reference count) under memory pressure (§4.2.1). This mechanism, in conjunction with the safe borrowing invariant (2), prevents the local cache from memory leaks or illegal accesses.

Owner Access without Borrow. DRust treats a direct memory access via the owner `Box` as a pair of mutable/immutable borrow and return. Depending on the reference type, DRust uses the extension field of `Box` accordingly and executes the read/write dereferencing logic. A special case arises when a mutable owner is immutably borrowed and becomes immutable until all borrowed references return. In this case, the owner can only cache the object during the borrow and delay the move until the borrow finishes. This would not create any correctness issues because the owner cannot be used for write access during this period.

Algorithm 3: Utility functions for pointer coloring.

```
1  Function GETCOLOR(g):
2      return g ≫ 48
3  Function CLEARCOLOR(g):
4      return g & ((1 ≪ 48) − 1)
5  Function APPENDCOLOR(g, c):
6      return CLEARCOLOR(g) | (c ≪ 48)
```

Ownership Transfer. Similar to Rust, DRust does not move the actual value during the transfer and only copies the `Box` pointer. DRust additionally checks and resets the pointer's extension field and frees the cached copy in the executing machine's cache to avoid cache leakage.

Memory Deallocation. Like Rust, DRust tracks the lifetime of an object via its owner. Given that ownership transfer is implemented by only evicting the cached copy of the object (without changing its global presence), the memory safety of DRust's global heap is preserved by the singular owner invariant (1). In other words, DRust still guarantees that when an object's owner goes out of scope, the object must be unreachable (and dead) and can be safely deallocated.

Consistency Model. Our protocol, together with Rust's ownership model, offers *sequential consistency* for cross-server memory accesses in safe Rust programs (*i.e.*, following the original Rust, no guarantees can be provided when Rust `Unsafe` is used), which is a strong consistency order. Therefore, it allows any safe Rust program to preserve its memory consistency on DSM. Sequential consistency necessitates a coherent memory system, requiring not only the SWMR invariant but also the *data-value* invariant [57]. In simple terms, the data-value invariant requires that the latest write to a value is immediately visible to subsequent readers. As discussed earlier, DRust's protocol moves an object upon a write and updates the owner immediately. Therefore, the latest value is globally visible after each mutable borrow finishes. Subsequent read accesses, either in the Owned state or the Shared state, are hence guaranteed to see the moved object and read its latest value.

Optimizing for Local Writes. A special case is that a server issues a write to an object that resides in its own heap partition. While the coherence protocol still guarantees safety, requiring moving an object in its local heap each time it is written clearly brings inefficiencies. To optimize for local writes, DRust adopts a pointer-coloring method, inspired by the design of concurrent garbage collectors in a managed runtime system such as JVM [1, 52]. Several utility pointer coloring functions are shown in Algorithm 3 which are used when dereferencing and dropping a reference. We reserve the first 16 bits of a global address as a "color". The color

value stored in the object's owner gets incremented upon the expiration of a mutable reference, as detailed in Lines 6–7 in Algorithm 1. Any subsequent immutable borrow would look up the cache with the object's global address. Even if the actual address remains the same, its color changes if a write has occurred. As such, the lookup would not return any stale copy from the local cache.

The 16-bit `color` field may overflow when the pointer keeps being borrowed for local writes on the same server. DRust implements a *move-on-overflow* strategy that moves the object to a new address and resets its color to zero once the maximum color value is reached (2^{16}), thereby preventing overflow and maintaining system integrity and performance.

Writing Unsafe Code in DRust. Rust allows developers to bypass compiler safety checks and write *unsafe* code for low-level operations such as accessing raw pointers and mutating shared variables at their own risk [44, 70]. Since DRust relies on SWMR semantics enforced by Rust's ownership types, DRust ensures consistency and memory safety only in the "safe" Rust code. DRust does not cache objects in unsafe code but allows developers to implement their own cache. Developers must ensure that they do not violate consistency in unsafe code blocks where type safety is not enforced. This caution mirrors practices in other managed languages, like native code in Java and unsafe code in C#. To assist developers, DRust offers primitives such as `dalloc`, `dread`, and `dwrite` for managing data on the global DSM heap.

4.1.2 Adapting Rust Standard Libraries

To further reduce the barrier for programs to run distributively, we reimplement several standard Rust libraries atop DRust's core memory constructs covering four categories: threading for distributed computation (`std::thread`), inter-thread channel for communication (`std::sync::mpsc`), reference-counted pointers for ownership sharing (`std::sync::Rc` and `std::sync::Arc`), and shared-state locks for concurrency control (`std::sync::Mutex` and `std::sync::atomic`).

Threading. DRust's threading library enables Rust threads to run distributively with two major adaptations. First, it enables distributed thread launching by re-implementing the `spawn` interface. Internally, it captures the thread body as a closure during compile time and forwards it to the runtime. During execution, the runtime launches the thread according to each server's load (details in §4.2.1). Second, DRust performs implicit ownership transfers between the parent and the child threads at the start or the end of the child thread execution. Thanks to the distributed ownership transfer support provided by DRust's memory model, the implementation in the threading library is hidden from developers and preserves type soundness and memory safety. Additionally, DRust is compatible with advanced thread utilities such as `thread::scope`, which allows for the spawning of scoped threads that can borrow non-static data. These utilities ensure that all threads are joined at the end of their scope and can internally utilize DRust's functions for spawning and joining threads, thus extending their applicability to the distributed setting.

Inter-Thread Channel. DRust extends Rust's channel to connect two distributed threads for message passing. DRust internally builds a network-based message queue for cross-server messages. Benefiting from the shared global heap, `Box` pointers and references can be safely copied and remain valid across servers. Therefore, the sender can push an object into the channel *as is* without serialization, even if it may contain `Box` pointers. DRust forwards the object binary bytes to the receiver over the network, and the receiver can recover the object from the binary by direct type conversion without deserialization.

Ownership Sharing. Rust allows multiple owners to share an object via reference-counted smart pointers, which count the number of live owners. In this case, smart pointers only have read access, and the object lifetime terminates when all owners die and the reference count hits zero. DRust does not require special treatment for `Rc` as it only allows ownership sharing inside a single thread. For `Arc` which shares ownership among multiple threads, DRust handles it in a similar way to immutable references with on-demand local caching and lazy eviction.

Shared-State Concurrency. Rust supports shared-state concurrency, primarily through its atomics and mutexes, where threads commonly share an atomic-typed value or one mutex via ownership sharing (*i.e.*, `Arc`). Unfortunately, the ownership model cannot type check concurrent read/write to shared states. Hence, Rust relies on an *unsafe* implementation in its standard library. §4.1.1 already provides a general discussion on writing unsafe code in DRust, and here we focus on DRust's implementation for distributed shared states.

Shared states create a unique challenge for DRust, as they may be replicated on multiple servers and those states must be synchronized among these servers. For example, an `Arc<AtomicBool>` may be replicated across different servers and used independently, causing multi-version issues if not synchronized properly. DRust addresses this inconsistency by allocating the actual value on the global heap and storing only the `Box` pointer in `atomic` types. This design allows atomics to be freely moved or replicated across servers while keeping a single version of the actual value. To synchronize concurrent operations on atomics, DRust adapts methods of atomic types to forward the operation as a message to the server storing the actual value, which serializes all operations with unsafe logic similar to the original Rust to guarantee atomicity and memory consistency. Similarly, DRust implements `Mutex` by allocating its metadata and owned object on the global heap and leaving only `Box` pointers in the mutex struct. Concurrent operations on mutexes are serialized on the server storing the mutex.

```
1  pub struct Node { val: i32, next: Option<TBox<Node>>, }
2  pub struct List { pub head: Option<Box<Node>>, }
3  impl List {
4    pub fn sum(&self) -> i32 {
5      let mut total: i32 = 0;
6      if let Some(r) = &self.head {
7        let mut node = &**r; // Fetch whole list to local.
8        loop { // Iterate every list node.
9          // Accessing node is guaranteed local.
10         total += (*node).val;
11         if let Some(next) = &node.next {
12           node = &**next;
13         } else { break; }
14       }
15     }
16     total
17   }
18 }
```

Listing 3: A linked list implementation with `TBox` in DRust. The use of `TBox` ties list nodes one by one. Iterating a list will fetch all nodes together (if they are on another server), and henceforth accessing any node is guaranteed local.

4.1.3 Affinity Annotations

To further improve performance, DRust allows developers to provide optional data affinity semantics via annotations. These annotations are useful for many datacenter applications that make extensive use of object-oriented data structures that require *pointer-chasing* to access. For instance, Memcached [55] uses a chained hash table where each hash bucket stores its KV pairs with a linked list. To find one KV pair from a bucket, Memcached has to iterate the linked list following the node pointers. However, frequent pointer chasing is unfavorable in a distributed setting, where each pointer dereference incurs additional runtime checks and potential cross-server traffic. It would be beneficial for the runtime to colocate them on the same server and schedule the computation there.

Data-Affinity Pointer. To expose data affinity for clustered placement, DRust includes a new pointer type `TBox` for developers to "tie" a heap object to its owner. `TBox` shares the same interfaces as the ordinary `Box` and can be used as a drop-in replacement for `Box`. However, `TBox` enforces that the pointed-to object must reside on the same server as its owner. In other words, when its owner object is copied or moved, the object referenced by `TBox` will be copied or moved as well. `TBox` can be used in a nested manner to allow a large union of objects to be co-located. The DRust runtime fetches (*i.e.*, copies or moves) them together in a single batch, leading to fewer network round-trips and higher throughput. `TBox` can also be assigned to and owned by a stack variable, in which case the referenced object is pinned onto the heap partition of the server that hosts the stack and cannot be moved. Dereferencing a `TBox` is thus guaranteed to be a local access—DRust optimizes it by skipping the runtime check.

Listing 3 presents a linked list implementation using `TBox`. Our linked list uses `TBox` (Line 1) to specify the data affinity between consecutive nodes. As a result, all list nodes are chained with `TBox`, forming an affinity group. Line 4–17 define a `sum` function that calculates the total sum of all node values. Assuming the list is non-empty, Line 7 dereferences the pointer to the head node, and the DRust runtime checks the location of the head node and fetches the entire list of nodes together if they are not local. Next, accessing each node inside the loop body (Line 8–14) is guaranteed local and hence skips runtime checks. Compared to using `Box` directly, `TBox` makes both data fetching and accessing more efficient.

```
1 fn main() {
2   let val: Box<i32> = Box::new(5);
3   let mut a = Accumulator{val};
4   let remote_add = spawn_to(a.val, move ||
5     a.add(10)).join(); // a.val == 15
6 }
```

Listing 4: A distributed accumulator can leverage `spawn_to` to offload a thread to the server where `a.val` locates.

Data-Affinity Thread. To expose the affinity between data and computation for thread scheduling, DRust extends its threading library with a `spawn_to` interface. `spawn_to` mirrors the ordinary `spawn` interface to spawn a new thread but takes an additional `Box` pointer argument, which indicates where the thread should be created. The runtime checks where the `Box` points to and creates the new thread on that same server. A common practice to use `spawn_to` is to pass the mostly-accessed object as the location indicator. Listing 4 presents how the distributed accumulator (shown in Listing 2) can use `spawn_to` to offload a thread to the same server as `a.val` resides. Line 5 hence performs local dereference to `a.val` inside `a.add()`.

4.2 DRust Runtime System

DRust's runtime system is the core component that manages memory and compute resources. It includes a runtime library (§4.2.1) that is linked to each application and launched on each server and a cluster-wise global controller (§4.2.2).

4.2.1 Application-Integrated Runtime

The runtime library consists of a communication layer to support inter-server coordination and data transfer, a heap allocator to manage the heap partition and the read-only cache, and a thread scheduler to launch and schedule application threads.

Communication Layer. The DRust runtime uses its communication layer to support the cache coherence protocol and cross-server memory accesses. The communication layer consists of a control plane and a data plane, both built with RDMA. The control plane leverages *two-sided verbs* to send and receive small control messages, and the receiver can perform the coherence logic upon receiving the message to minimize the end-to-end latency. The data plane, in contrast, is specialized for bulky data transfer with one-sided verbs. It fetches an object as a whole with a single RDMA message upon pointer dereferencing without interrupting the target server, minimizing both latency and CPU usage.

Heap Allocator. The DRust runtime provides standard memory allocation interfaces and always returns global addresses to the upper-level language abstractions. It prioritizes local

memory allocation as long as the local heap partition has sufficient space. This strategy improves data locality by colocating an object with its creating thread.

When the local heap partition is depleted, DRust queries the global controller and allocates memory on the most vacant server. For remote memory allocation, it forwards the request to the target server by sending a message through the communication layer and returns the allocated address to the user. Memory deallocation follows a similar logic but it bypasses the controller and finds the server directly via the object's global address. The allocator does not reserve separate space for the local cache. Instead, it manages the cache as part of the local heap partition and always allocates cached entries in the local heap partition. Under memory pressure, the allocator will scan the local cache and evict entries that are no longer used (*i.e.*, reference count hits zero).

Thread Scheduler. The DRust thread scheduler runs in the user space and schedules threads locally to maximize CPU utilization. It also provides thread migration functionalities, facilitating the global controller to balance load between busy and vacant servers.

The scheduler represents a newly created user thread as a closure, which includes a function pointer and a set of initial arguments (*i.e.*, references). It collaborates with the global controller to allocate a unique stack space for a thread (see Figure 3), and starts the thread by executing the closure.

The scheduler adopts the method of cooperative multitasking and context switches between threads *non-preemptively*. A running thread yields its control flow proactively when developers call `await` or reactively upon long-latency operations. Similarly to other systems [60, 65, 82], our scheduler handles context switches as function calls, which allow DRust to save only a few registers per thread.

The scheduler supports creating/migrating a thread to another server as well. To migrate a thread, DRust sends its function pointer, the saved register state, and its stack to the target server. Because each thread reserves its stack address range globally, DRust can copy the stack across servers without changing its address. Therefore, the thread can be easily resumed by directly calling the function pointer with the saved register state on the target server. DRust generates code for state transmission during the compile time for the scheduler to call upon thread migration.

4.2.2 Global Controller

The controller runs as a daemon process on the machine where the program is launched. It manages cluster resources and coordinates memory allocation and thread migration. It periodically pings each server to probe and record its resource usage (CPU and memory). It controls resource allocation in cooperation with the DRust runtime on each server. When allocating memory or creating a thread, the runtime will first query the controller, which chooses a target server following adaptive policies (discussed later), and then coordinate with the runtime on the target server to perform the actual operation. The controller also maintains a global table to track the location of each thread; the table is queried and updated by the scheduler when migrating a thread.

During program execution, servers may run into imbalanced loads when objects get relocated or new threads are created. DRust balances the load of each server by *migrating* threads from the busy server to less occupied ones, following an adaptive policy to minimize cross-server memory accesses. If a server is about to run out of memory (>90% memory usage), the controller keeps migrating the thread that consumes the most local heap memory until the pressure is resolved. If the server is under compute congestion (>90% CPU usage), the controller migrates threads that frequently access remote objects. The thread is then moved to the server it accesses the most unless the target server is also overloaded, in which case it moves to a vacant server instead.

4.2.3 Fault Tolerance

In DRust, the global heap can be replicated to tolerate failures. Replication creates copies for each heap partition at the same virtual address on backup servers. Threads, in contrast, are not replicated for efficiency and are only executed with the primary global heap. To maintain a synchronized view between the primary heap partition and its backup copy, a thread must additionally write back to the backup partition after each mutable borrow. However, our insight is that the thread can batch modifications to an object and delay the write-back until the object ownership is transferred to another server, which is the time point that the object becomes visible to threads on other servers. When a server with a primary heap partition fails, the controller will automatically promote its backup server to the primary and add a new backup server.

5 Implementation

The majority of DRust was implemented in Rust except for its communication library which is in C. We implemented DRust's core language constructs as three Rust types (*i.e.*, `struct`): `Ref<T>`, `MutRef<T>`, and `DBox<T>`. They serve as the counterpart for the original Rust `&T`, `&mut T`, and `Box<T>`, respectively. We implemented the coherence protocol with traits on these types, including `Copy`, `Clone`, and `Drop`, which are automatically embedded into the program source code and executed when references/pointers are created or destroyed. To support unmodified Rust programs, we changed the Rust compiler and added additional compilation passes to transform Rust references and `Box` pointers into corresponding types in DRust.

Our communication layer links `libibverbs` directly for fast and kernel-bypassing RDMA networking. We implemented a low-level C library that covers basic connection establishment and exposes high-level Rust interfaces for various RDMA verbs, including `RDMA_READ`, `RDMA_WRITE`, `RDMA_SEND`, `RDMA_RECV`, `ATOMIC_FETCH_AND_ADD`, and `ATOMIC_CMP_AND_SWP`.

We primarily utilize one-sided `READ` and `WRITE` verbs for data transfers between servers, as they outperform the two-sided `SEND`/`RECV` counterparts—one-sided operations bypass the CPU and OS at the receiver side, whereas two-sided operations require the receiver to pre-post `RECV` verbs and await notification upon message arrival. For instance, when a remote object is accessed via mutable references, DRust copies the object to local memory using the `READ` verb. Upon dropping the reference, DRust updates the original owner `Box` to reflect the new address, a process executed using the `WRITE` verb. Conversely, two-sided `SEND`/`RECV` verbs are utilized for control message exchanges, such as establishing connections across servers. Atomic verbs `ATOMIC_FETCH_AND_ADD`, and `ATOMIC_CMP_AND_SWP` are primarily utilized for implementing shared states (e.g., atomic types and mutexes). DRust uses the RC (reliable connection) transport type to ensure reliable transmission and strict message ordering.

Our heap allocator implementation piggybacks Rust's original allocator and aligns its virtual address range with the heap partition range. Our thread scheduler was built upon Tokio [82] for its efficient user thread and cooperative scheduling integration. The global controller is responsible for managing all threads in the cluster and padding their stacks to avoid address overlapping.

6 Limitations

DRust's design has three limitations. First, although DRust permits the use of unsafe code, its consistency guarantees are only applicable to safe Rust code. In unsafe code blocks, developers are responsible for ensuring consistency themselves. Second, DRust's superior performance relies on SWMR semantics exposed by applications. In cases where data is mostly under shared states (such as `Mutex`), DRust degenerates into a traditional DSM system; all concurrent accesses to the same data have to be centralized and serialized by the server responsible for the shared states. However, such scenarios contradict Rust's recommended programming practices. Finally, the current implementation of DRust does not support address space layout randomization (ASLR) yet, and we have temporarily disabled it. However, DRust's design is compatible with ASLR as long as DRust threads share the same randomized address space layout on each server. This can be achieved by delegating the randomization of stack and heap address allocation in DRust to its global controller, a feature that will be supported in future versions of DRust.

7 Evaluation

Setup. We evaluated our system on an 8-node cluster, where each node was equipped with dual Intel Xeon E5-2640 v3 processors (16 cores), 128GB of RAM, and a 40 Gbps Mellanox ConnectX-3 InfiniBand network adapter, connected by a Mellanox 100 Gbps InfiniBand switch. All servers ran Ubuntu 18.04 with kernel 5.14. We disabled hyperthreading, CPU frequency scaling, OS security mitigations in accordance with common practices [69, 72].

Application	Dataset	Memory (GB)	Comp. Intensity (cycles/byte)
DataFrame [67]	h2oai [37]	64	110.13
SocialNet [32]	Socfb-Penn94 [71]	64	86.09
GEMM [11]	LAPACK [2]	96	300.63
KV Store [14]	YCSB [22]	48	48.15

Table 1: Applications used in the evaluation.

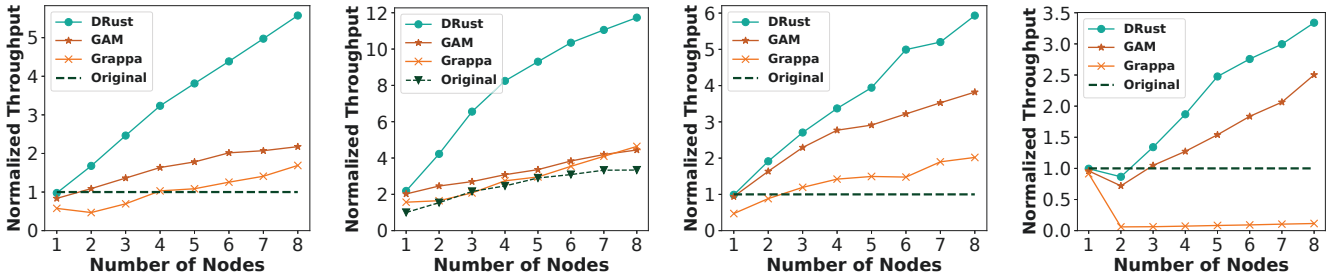
Methodology. We compared DRust with two state-of-the-art DSM systems, GAM [14] and Grappa [60]. For a fair comparison, we ported the evaluated applications to each baseline system and invested extensive effort in tuning parameters to achieve their best possible performance. GAM offers ordinary object read/write interfaces, and we exported it as a library to Rust and hooked pointer dereferencing to use GAM's API without program modification. Grappa, in contrast, offers a drastically different programming abstraction that requires rewriting the program to access shared memory via *delegation*. Therefore, we re-implemented applications in C++ and re-structured them using Grappa's abstractions.

7.1 Applications

We evaluated four representative datacenter applications covering a wide range of use cases and resource demands, including data analytics, microservices, scientific computation, and key-value storage, as shown in Table 1.

DataFrame is an in-memory data analytics framework similar to Spark [91] and Pandas [90]. We built our library atop Polars [67], a native DataFrame engine in Rust offering OLAP query APIs such as `filter`, `groupby`, and `join`. DataFrame organizes the dataset as columnar format tables in shared memory, and user queries will manipulate table columns by reading/writing rows and transforming them into new tables. DataFrame exploits data-level parallelism by internally partitioning columns by row into an array of small chunks where each chunk can be processed independently. We additionally applied `TBox` to annotate chunks from the same table column for co-location and used `spawn_to` to offload columnar operations to the data side to improve data locality and performance. Note that such annotations were not necessary for the application to run; they were added for additional performance optimizations.

SocialNet is a twitter-like web service from the DeathStarBench suite [32]. It is composed of 12 microservices with complicated call dependencies. Each microservice in SocialNet can scale independently with replicas, thereby offering higher throughput with more servers. SocialNet decouples the process of user texts, media resources, and storage into different microservices, and it employs RPCs to pass values (texts, media files, *etc.*) between them. DRust enables



(a) DataFrame (318 s) (b) SocialNet (120 ops/s) (c) GEMM (1039 s) (d) KV Store (2.7 Mops/s)

Figure 5: Application throughput when running with DRust, GAM, and Grappa, normalized to the throughput of their original implementation running on a single node. The number in the parenthesis is the original application's throughput on a single node.

SocialNet to pass only references in RPCs, eliminating the serialization/deserialization overhead and redundant data copies. Because SocialNet was implemented in C++ and deployed with Docker Swarm [28], we ported it into Rust for our evaluation. We followed its original microservice structure but changed the RPC call sites to pass references instead of values, and we followed the original orchestration configuration to spread and scale each microservice in the cluster. We did not use any affinity annotations for SocialNet.

GEMM (general purpose matrix multiplication) is a highly-optimized matrix multiplication routine from the BLAS library [11]. We ported the library using the same divide-and-conquer algorithm by recursively partitioning each matrix into small chunks for parallel processing and reducing the final results. Input and output matrices are stored in the shared memory, where each subroutine will read two input matrix chunks and write the partial results back to the output matrix. Our port strictly followed the original implementation without using additional affinity annotation.

KV Store is an in-memory key-value cache engine similar to Memcached [55]. It uses a hash table to store KV pairs in shared memory and mutexes to synchronize concurrent requests. We used YCSB benchmark [22] to generate zipf load with 90% GET and 10% SET using default skewness parameter 0.99.

7.2 Scaling Performance

In this experiment, we investigated whether DRust can speed up applications by distributing them in a cluster and how well they can scale with the number of servers used. For each application, we first ran it *as is* on a single server without using DSM and measured its throughput. Then, we ran the same application on DSM (subject to modifications when running Grappa) with the same configuration but on varying numbers of servers and measured the throughput normalized to its single-node throughput (*i.e.*, strong scaling). As GAM and Grappa cannot adaptively balance the workload across servers, we evenly distributed the application's working set and threads among all participating nodes. Ideally, an application should scale linearly and enjoy proportionally higher throughput with more nodes. However, this is usually unachievable because of the limited parallelism of real-world applications and the coherence overhead of DSM systems, and a good result for DRust will show that applications' throughput can get close to their ideal throughput.

Figure 5 shows the results for each application respectively. DRust outperforms both GAM and Grappa in all cases. On a single node, it is 1.04–2.10× faster than two baseline DSMs, while only adding a maximum overhead of 2.42% compared to the original program. When running with multiple nodes, DRust scales up applications significantly better than GAM and Grappa. On eight nodes, DRust achieves a throughput that is 1.33–2.64× higher than that of GAM, 2.53–29.16× higher than that of Grappa.

Compared to each program's single-machine performance, using DSM over DRust enables each program to easily leverage the available distributed resources and achieve a throughput that is 3.34–11.73× higher than their single-machine counterparts. Next, we discuss each application to explain the scalability difference between DRust and the baseline DSMs.

DataFrame. As shown in Figure 5a, compared with its original version, DataFrame running on eight nodes with DRust achieves 5.57× higher throughput, whereas with GAM and Grappa, the throughput improvements are 2.18× and 1.69×, respectively. In other words, DataFrame with DRust is 2.56× and 3.29× faster than GAM and Grappa on eight nodes, respectively.

A detailed examination reveals that the performance difference comes from the *shared index table* in each DataFrame operation and the *shared chunks* between dependent DataFrame operations. In each operation, DataFrame constructs an index hash table to track the mapping from each destination chunk in the output column to all its source chunks in the input column. This index table is shared by all index-builder threads and worker threads. During processing, index-builder threads will concurrently insert into the index table using the destination chunk ID as the key and an array of source chunk IDs as the value, and worker threads will look up the shared index table and retrieve source chunks for processing. As a result, the massive writes and reads to the shared table can incur heavy coherence overhead. Further, DataFrame passes chunks as references between dependent

operations and relies on the DSM system for actual data movement. However, it only performs lightweight computation over the fetched data (*i.e.*, low compute intensity as shown in Table 1), making the coherence overhead stand out.

DRust outperforms GAM and scales much better because of its light coherence protocol, which incurs negligible object move overhead for writes and no coherence overhead for reads. The use of affinity annotations also helps DataFrame colocate worker threads with their frequently accessed data, bringing 20% additional boost (details in §7.3). GAM, in contrast, has to invalidate each cache block upon each write and read, thereby bottlenecked by the extensive coherence traffic. Grappa performs the worst in all three DSM systems due to its *always-delegation* programming model, which implements every global memory read/write via a delegated function call. The cost for delegation overwhelms the actual memory access latency in this case, ruining the performance of the shared hash table. Grappa's delegation overhead actually causes a 1.23× slowdown when scaling DataFrame from a single node to two nodes.

SocialNet. Since SocialNet is microservice-based and can be deployed distributively, we added another baseline by running the original (non-DSM) code but deploying it on varying numbers of nodes. Figure 5b demonstrates the performance of all systems. SocialNet runs consistently faster with all three DSM systems compared to the original version. DRust, GAM, and Grappa achieve a 2.18×, 2.02×, and 1.57× speedup on a single node and a 3.51×, 1.33×, and 1.39× speedup on eight nodes, respectively. In the conventional setup, SocialNet requires data—such as text and media files—to be serialized into byte streams for network transmission, and then deserialized back into usable formats at the receiving end. This serialization and deserialization process is computationally intensive, particularly for large or complex data objects. In contrast, DSM systems enable SocialNet to pass references instead of the entire data values required by remote procedure calls. This approach eliminates the need for serialization and deserialization, reduces redundant data copies, and significantly enhances performance. DRust scales much better than GAM and Grappa thanks to its lightweight coherence protocol, achieving up to 2.77× and 3.16× higher throughput than GAM and Grappa, respectively.

GEMM. GEMM differs from the previous two applications in its high compute intensity and relatively infrequent shared memory accesses. In this application, matrices are transformed and divided into smaller sub-matrices for parallel processing. Each computing thread, responsible for multiplying sub-matrices, is assigned to a server. These threads cache their respective sub-matrices in the server's local memory and access them repeatedly to compute product results. This process is highly compute-intensive. As depicted in Figure 5c, DRust and GAM scale well for GEMM and achieve 5.93×, 3.82× speedup with eight nodes. In contrast, Grappa only achieves a 2.02× speedup with eight nodes due to its inability

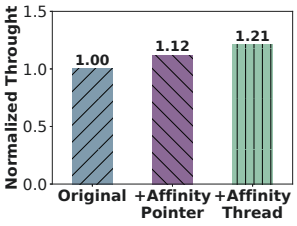


Figure 6: Effectiveness of DRust's affinity annotations.

Latency (cycles)	Average	Median	P90
DRust	395	356	536
Rust	364	332	496

Table 2: DRust's `Box` pointer only adds a small dereferencing cost compared to Rust's ordinary `Box`.

to cache sub-matrices locally, necessitating frequent remote accesses. DRust's superior performance over GAM, with a 1.55× higher speedup on eight nodes, is primarily due to its more efficient handling of initial cross-server data accesses required when a sub-matrix is first accessed remotely. Unlike GAM, which incurs significant runtime overhead due to the maintenance of state and location of data copies, DRust directly copies data to local memory, without any complex cross-server synchronization operations, thus enhancing overall efficiency.

KV Store. KV Store is the most DSM-unfriendly application in our evaluation because it exposes poor memory locality and low compute intensity, which amplifies the overhead of cross-server memory accesses. In addition, it uses mutexes to synchronize between workers and the structure of the program does not lend itself to ownership-based read/write ordering.

Figure 5d shows the results. KV Store experiences a slowdown on all three DSM systems when scaling from a single node to two nodes (13% for DRust, 25% for GAM, and 93% for Grappa). However, the impact is mitigated when more servers are enlisted—DRust and GAM achieve 3.34× and 2.50× higher throughput on eight nodes compared to the original KV Store implementation, respectively. Due to the limited ownership semantics exposed by mutexes, DRust does not scale as well with KV Store as with other applications. DRust is 1.33× faster than GAM on eight nodes, benefiting from its adaptive load balancing and a more efficient implementation of mutexes utilizing one-sided RDMA atomic verbs, whereas GAM depends on less efficient two-sided RDMA messages for synchronization. Grappa, in contrast, incurs the highest distribution overhead and poorest scalability, primarily because each PUT/GET operation requires remote delegation, and nodes handling popular objects become bottlenecked due to skewed load.

7.3 Drill-Down Experiments

Affinity Annotations. In this experiment, we evaluated the individual contributions of affinity annotations by enabling each of them incrementally for DataFrame on eight nodes. Figure 6 reports the results. Using `TBox` helps DataFrame group chunks from the same column and eliminates the runtime dereference check overhead for single-column operations (*e.g.*, `filter`), bringing a 12% throughput improvement. Adding `spawn_to` further improves the throughput by 9% by

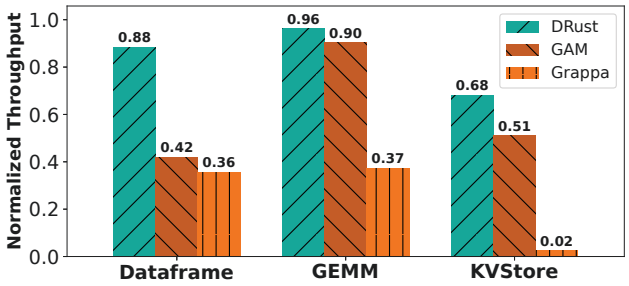


Figure 7: Comparison of cache coherence costs between DRust, GAM, and Grappa on eight nodes.

informing DRust runtime to colocate the worker thread to its input columns, which reduces cross-server memory accesses.

Runtime Dereference Checks. We measured the latency of dereferencing DRust's `Box` pointer and compared it with an ordinary Rust `Box` pointer. Both of them point to an 8-byte object in local memory and not in CPUs cache, which represents the common path for pointer dereferencing. Table 2 reports the results. DRust only adds a small overhead of ∼30 cycles. Note that this microbenchmark is extremely memory-intensive, whereas real-world applications usually employ larger object sizes and are more compute-intensive, further mitigating the runtime check overhead. For our evaluated applications, we observed a 1.02% overhead for DataFrame and a 1.14% overhead for BLAS, when they run with DRust on a single node, respectively.

Thread Migration Latency. To quantify how quickly DRust can resolve the workload imbalance, we measured the latency for the DRust runtime to migrate a thread by running GEMM on eight nodes and repeated the experiment for ten times. On average, DRust migrated 15 threads with an average of 218μs latency for each migration.

Cost of Cache Coherence. In this experiment, we ran each application again on a single node and eight nodes but fixed the total amount of CPU and memory resources. For the eight-node setting, we distributed the resources evenly to each node and measured application throughput. We expect to see a slowdown due to the cost of running the coherence protocol and cross-server memory accesses, but a good result for DRust should show that application performance remains close to its original single-node version. Figure 7 reports the results. SocialNet uses pass-by-value RPCs in its original version and is significantly slower than our DSM-based version, so it is omitted in the evaluation. DRust adds only moderate cache coherence cost with an overhead of 32% in the worst case (KV Store) and 4% in the best case (GEMM). GAM and Grappa, in contrast, incur much higher overheads ranging from 10% to 98% for different applications.

8 Related Work

Software DSM Systems. Distributed cache coherence protocols and their implementations for DSM have been extensively studied since 1980s [16–18,31,36,49,50,56,61–63,80]. Among them, Munin [10] and TreadMarks [6] proposed relaxed consistency models and simpler protocols trying to alleviate the coherence overhead. Recent DSM systems leveraged today's advanced hardware such as RDMA [14,45,60,77,81,92] to improve efficiency.

Disaggregated and Remote Memory. Memory disaggregation and remote memory techniques are another promising approach to scaling applications out of a single machine. Their key idea is to connect a host server with large memory pools [33, 40, 46] via fast datacenter network, which can be accessed by applications via OS kernel [4, 69, 76, 86] or software runtimes [34, 52, 73, 84, 85, 87]. However, they do not provide cache coherence.

Distributed Programming Abstractions. Researchers have studied and proposed new programming languages and abstractions. Munin [10] built a type system that defines types for local and global pointers and tracks whether the pointer is shared via type checking. X10 [20, 39] and UPC [30] introduce function offloading interfaces for distributed computing and additional type annotations to reduce the runtime overhead. Ray [89] and Nu [72] are two recent systems proposing new abstractions for distributed programming. Unlike DRust, they require effort to port applications to avoid fine-grained memory sharing.

Hardware-Accelerated DSM. Specialized datacenter network technologies and emerging hardware designs stand for another trend to accelerate DSM. Clio [35], StRoM [79], and RMC [5] reduce remote memory access latency by offloading tasks into customized hardware. Concordia [88], Kona [15], and CXL [23–26,47,48,92] enable block-level or cache-line-level memory coherence with their hardware-implemented protocols. DRust can benefit from advances in hardware support and achieve better scalability.

9 Conclusion

This paper presents DRust, a practical DSM system based on the ownership model. It automatically turns a single-machine Rust program into its distributed version with a lightweight coherence protocol guided by language semantics. DRust significantly outperforms existing state-of-the-art DSM systems, demonstrating that a language-guided DSM can achieve strong memory consistency, transparency, and efficiency simultaneously.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Daniel S. Berger for his feedback. This work is supported by CNS-1763172, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2147909, CNS-2128653, CNS-2301343, CNS-2330831, CNS-2403254, CNS-1764077, CNS-1956322, CNS-2106404. This work is also supported by Alibaba Group through Alibaba Research Intern Program, and funding from Amazon and Samsung.

A Artifact Appendix

A.1 Artifact Summary

DRust is an efficient, consistent, and user-friendly DSM system featuring a lightweight coherence protocol guided by language semantics. DRust allows for seamless scaling of single-machine applications to multi-server environments without sacrificing performance. Demonstrating significant improvements over existing DSM systems, DRust combines strong memory consistency, transparency, and efficiency effectively.

A.2 Artifact Check-list

- **Hardware:** Intel servers equipped with InfiniBand
- **Software Environment:** Rust 1.69.0, GCC 5.5, Linux Kernel 5.4, Ubuntu 18.04, MLNX-OFED 4.9
- **Public Link to Repository:** https://github.com/uclasystem/DRust
- **Code License:** GNU General Public License (GPL)

A.3 Description

A.3.1 DRust's Codebase

DRust comprises four main components:

- An RDMA communication library written in C
- The DRust library
- Applications integrated with DRust
- Necessary shell scripts and configuration files

A.3.2 Deploying DRust

The initial step in deploying DRust involves cloning the source code on all involved servers:

```
git clone git@github.com:uclasystem/DRust.git
```

Adjust several configurations according to your server setup and operational requirements:

1. Set the Number of Servers:
   - Define `TOTAL_NUM_SERVERS` in `comm-lib/rdma-common.h` based on the total number of available servers.
   - Similarly, adjust `NUM_SERVERS` in `drust/src/conf.rs`.
2. Configure the Distributed Heap Size by setting `UNIT_HEAP_SIZE_GB` in `drust/src/conf.rs` to the required heap size per server, e.g., 16 for 16GB.
3. Update the InfiniBand IP addresses and ports in `comm-lib/rdma-server-lib.c`:

```
const char *ip_str[2] = {"10.0.0.1",
"10.0.0.2"};
const char *port_str[2] = {"9400", "9401"};
```

4. In `drust.json`, update each server's IP address and specify three available ports.

Following configuration, build DRust as follows:

```
# Compile the communication static library
cd comm-lib
make -j lib
# Copy the static library to the DRust directory
cp libmyrdma.a ../drust/
# Compile the Rust components
cd ../drust
cargo build --release
```

Deploy the compiled binary across all servers post-build, ensuring its correct distribution:

```
scp target/release/drust user@ip:DRust/drust.out
```

A.3.3 Running Applications

DRust is bundled with four example applications: Dataframe, GEMM, KVStore, and SocialNet. Follow these steps to execute them:

1. Launch the DRust executable on all servers, excluding the main server:

```
# Start the DRust process with the specified
server index and application name.
# For example, ./../drust.out -s 7 -a gemm
cd drust
./../drust.out -s server_id -a app_name
```

2. On the main server:

```
# Start the main DRust process with the
specified application.
cd drust
./../drust.out -s 0 -a app_name
```

More details of DRust's installation and deployment can be found in DRust's code repository.

References

[1] The Z garbage collector. https://wiki.openjdk.java.net/display/zgc/Main.

[2] Lapack benchmark. https://www.netlib.org/lapack/lug/node71.html, 2023.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348, 2015.

[4] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[5] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[7] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. In *FAST*, 2014.

[8] H. G. Baker. Lively linear lisp: look ma, no garbage!. *SIGPLAN Not.*, 27(8):8998, aug 1992.

[9] T. Balabonski, F. Pottier, and J. Protzenko. The design and formalization of mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38(4), aug 2016.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[11] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[12] M. N. Bojnordi and E. Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *ISCA*, pages 13–24, 2012.

[13] K. Boos, N. Liyanage, R. Ijaz, and L. 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. USENIX Association, Nov. 2020.

[14] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[15] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. *Rethinking Software Runtimes for Disaggregated Memory*, pages 79–92. Association for Computing Machinery, New York, NY, USA, 2021.

[16] R. Campbell, G. Johnston, and V. Russo. Choices (class hierarchical open interface for custom embedded systems). *ACM SIGOPS Operating Systems Review*, 21(3):9–17, 1987.

[17] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. *ACM SIGOPS Operating Systems Review*, 25(5):152–164, 1991.

[18] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems (TOCS)*, 13(3):205–243, 1995.

[19] CCIX. Cache coherent interconnect for accelerators. https://www.ccixconsortium.com/, 2018.

[20] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, 2008.

[21] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, 2005.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[23] Compute express link 3.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf, 2022.

[24] Compute express link 1.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.0-Specification.pdf, 2019.

[25] Compute express link 1.1. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.1-Specification.pdf, 2019.

[26] Compute express link 2.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf, 2020.

[27] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 5969, New York, NY, USA, 2001. Association for Computing Machinery.

[28] Managing a Cluster of Docker Daemons using Swarm Mode. https://docs.docker.com/engine/swarm/, 2023.

[29] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[30] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: distributed shared memory programming*. John Wiley & Sons, 2005.

[31] B. D. Fleisch. Distributed shared memory in a loosely coupled distributed system. *ACM SIGCOMM Computer Communication Review*, 17(5):317–327, 1987.

[32] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 318, New York, NY, USA, 2019. Association for Computing Machinery.

[33] GenZ. Genz consortium. http://genzconsortium.org/, 2019.

[34] Z. Guo, Z. He, and Y. Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692708, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[36] D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE micro*, 12(1):10–22, 1992.

[37] Database-like ops benchmark. https://github.com/h2oai/db-benchmark, 2023.

[38] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, pages 10:1–10:7, 2013.

[39] R. Haque and J. Palsberg. Type inference for place-oblivious objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[40] Hewlett-Packard. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[41] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):3749, apr 2007.

[42] Intel. Intel high performance computing fabrics. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/, 2019.

[43] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[44] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[45] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2015.

[46] K. Keeton. The Machine: An architecture for memory-centric computing. In *ROSS*, 2015.

[47] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574587, New York, NY, USA, 2023. Association for Computing Machinery.

[48] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[49] K. Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[50] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[51] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, pages 267–278, 2009.

[52] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107, 2022.

[53] H. Ma, Y. Qiao, S. Liu, S. Yu, Y. Ni, Q. Lu, J. Wu, Y. Zhang, M. Kim, and H. Xu. Drust: Language-guided distributed shared memory with fine granularity, full transparency, and ultra efficiency. *arXiv preprint arXiv:2406.02803*, 2024.

[54] Mellanox. Connectx-6 single/dual-port adapter supporting 200gb/s with vpi. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card, 2019.

[55] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[56] R. G. Minnich and D. J. Farber. The mether system: Distributed shared memory for sunos 4.0. *Technical Reports (CIS)*, page 332, 1993.

[57] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, and N. E. Jerger. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd edition, 2020.

[58] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229240, sep 2008.

[59] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. 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. USENIX Association, Nov. 2020.

[60] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. {Latency-Tolerant} software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.

[61] J. Nieplocha, R. Harrison, M. Krishnan, B. Palmer, and V. Tipparaju. Combining shared and distributed memory models: Evolution and recent advancements of the global array toolkit. In *proceedings of POHLL'2002 workshop of ICS-2002, NYC*, 2002.

[62] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A portable" shared-memory" programming model for distributed memory computers. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. IEEE, 1994.

[63] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[64] OpenCAPI. Open coherent accelerator processor interface. https://opencapi.org/, 2018.

[65] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[66] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[67] Polars: Blazingly Fast DataFrame Library. https://pola-rs.github.io/polars/, 2023.

[68] Y. Qiao, Z. Ruan, H. Ma, A. Belay, M. Kim, and H. Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[69] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, 2023.

[70] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779, 2020.

[71] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[72] Z. Ruan, S. J. Park, M. K. Aguilera, A. Belay, and M. Schwarzkopf. Nu: Achieving {Microsecond-Scale} resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, 2023.

[73] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[74] S. M. Rumble. Infiniband verbs performance. https://ramcloud.atlassian.net/wiki/display/RAM/Infiniband+Verbs+Performance, 2010.

[75] Rust. https://www.rust-lang.org/, 2023.

[76] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[77] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[78] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A network architecture for disaggregated racks. In *NSDI*, pages 255–270, 2019.

[79] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[80] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 170–183, 1997.

[81] K. Taranov, S. Di Girolamo, and T. Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1811–1824, 2021.

[82] Tokio Team. Build reliable network applications without compromising speed. https://tokio.rs/.

[83] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for rust (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 75–80, 2015.

[84] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *OSDI*, pages 261–280, 2020.

[85] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, pages 35–53, 2022.

[86] C. Wang, Y. Qiao, H. Ma, S. Liu, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for {Multi-Applications} on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, 2023.

[87] C. Wang, Y. Shan, P. Zuo, and H. Cui. Reinvent cloud software stacks for resource disaggregation. *Journal of Computer Science and Technology*, 38(5):949–969, 2023.

[88] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[89] S. Wang, E. Liang, E. Oakes, B. Hindman, F. S. Luan, A. Cheng, and I. Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, Apr. 2021.

[90] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[91] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, page 10, Berkeley, CA, USA, 2010.

[92] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 658–674, 2023.