Gossamer: A novel programming paradigm for rack-wide applications

by

Noaman Ahmad
MS CS, University of Illinois at Chicago, 2025

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2025

Chicago, Illinois

Defense Committee:
Jakob Eriksson, Chair and Advisor
Xingbo Wu, Microsoft Research
Ajay Kshemkalyani
Luis Gabriel Ganchinho de Pina
Erdem Koyuncu
Michael Papka

Dedicated to my family, who always have my back.

ACKNOWLEDGMENT

A page or two so of shout-outs to people you appreciate. Don't forget your advisor and committee members!

NA

CONTRIBUTIONS OF AUTHORS

I would like to acknowledge Dr. Jakob Eriksson and Ben Baenen for their integral role in the completion of this dissertation.

Chapter 3 includes work done in collaboration with Dr. Jakob Eriksson, Ben Baenen and Dr. Chen Chen, and comes from a preprint paper in which I am the first author (1). The design and implementation of *Trust<T>* is a result of Dr. Jakob's, Ben's, and my work together and Dr. Chen helped with experimentation and writing.

Chapter 4 builds on work from Chapter 3. Dr. Jakob helped with its writing and Dr. Chen helped with evaluating the system.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF FIGURES (Continued)

FIGURE **PAGE**

SUMMARY

One to two page summary of the entire work. Like a long abstract.

CHAPTER 1

INTRODUCTION

As delegation does not rely on shared memory, it can serve as a powerful programming model for both concurrent and distributed computing. We present two system, *Trust<T>* and *Gossamer*, to demonstrate the versatility of this approach across different domains.

Trust<T> introduces a type- and memory-safe alternative to traditional locking in concurrent programs. Instead of synchronizing multi-threaded access to an object of type T with a lock, it can be encapsulated in a *Trust<T>* with a designated trustee thread handling all operations through message passing. This delegation-based model eliminates per-lock throughput limitations, achieving up to 22x higher throughput in microbenchmarks and 5-9x improvement for a key-value store in high-contention workloads, while remaining competitive even without contention.

Gossamer extends the concept of delegation to distributed computing. It allows an application written for a single machine to scale to a distributed cluster with minimal changes. Using Remote Direct Memory Access (RDMA) for high-performance communication, *Gossamer* supports delegating Rust closures across machines and employs lightweight user-space threads (fibers) for concurrency. Our experiments show 3x better throughput scaling than eRPC and performance comparable to the Graph500 MPI reference implementation.

Together, *Trust<T>* and *Gossamer* illustrate how delegation unifies concurrency and distribution under a safe, efficient, and scalable programming paradigm, enabling high- performance computation without the complexities of shared memory or explicit synchronization.

The main contributions of this dissertation are summarized below:

- *Trust<T>*: a model for efficient, multi-threaded, delegation-based programming with shared objects leveraging the Rust type system.
- *Gossamer*: an extension of *Trust<T>* that enables a normal delegation based application to run on and utilize the resources of a rack with minimal changes to application code.

The rest of this chapter introduces both of these and then outlines the rest of this dissertation.

1.1 ***Trust<T>***

We present *Trust<T>* (pronounced *trust-tee*), a programming abstraction and runtime system which provides safe, high-performance access to a shared object (or $\mathsf{property}$) of type T. Briefly, a *Trust<T>* provides a family of functions of the form:

$$\mathsf{apply}(\mathsf{c} : \mathsf{FnOnce}(\&\mathsf{mut}\ \mathsf{T}) \to \mathsf{U}) \to \mathsf{U},$$

which causes the closure c to be safely applied to the property (of type T), and returns the return value (of type U) of the closure to the caller. Here, FnOnce denotes a category of Rust closure types, and $\&\mathsf{mut}$ denotes a mutable reference. (A matching set of non-blocking functions is also provided, which instead executes a callback closure with the return value.)

Critically, access to the property is only available through the *Trust*<*T*> API, which taken together with the Rust ownership model and borrow checker eliminates any potential for race conditions, given a correct implementation of *apply*. Our implementation of *Trust*<*T*> uses pure delegation. However, the design of the API also permits lock-based implementations, as well as hybrids.

Beyond the API, *Trust*<*T*> provides a runtime for scheduling request transmission and processing, as well as lightweight user threads (`fibers` below). This allows each OS thread to serve both as a Trustee, processing incoming requests, and a client. Multiple outstanding requests can be issued either by concurrent synchronous fibers or an asynchronous programming style. *Trust*<*T*> achieves performance improvements up to 22× vs. the best locks on congested micro-benchmarks and up to 9× on benchmarking workloads vs. stock `memcached`.

1.2 Gossamer

Shared memory enables programmers to write multi-threaded applications, making them perform better but also increase the programming complexity. The same is true when scaling applications from a single machine to a distributed setting. Here, lack of shared memory means that not all of the compute units (threads/processes) can access all of the shared objects, which also complicates the synchronization mechanisms. Distributed applications, historically, rely on message passing to share data. Remote Procedural Calls (RPCs) is one such framework where threads can send requests to where data is located and receive responses upon completion (2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12). This is very similar to how delegation works.

Gossamer extends *Trust*<*T*> to build a rack-wide programming model. *Gossamer* aims to provide a high performance and easy to program in, framework that can take advantage of the increased resources available in a rack. Using the API provided by *Gossamer* programmers can code their applications like a normal delegation application with very few changes and the applications have the illusion of running on the same machine. Most of the distribution of work is handled behind the scenes, but programmers have the option to customize where the worker threads should run and where data should be held in the rack if they so choose. This can be achieved by mapping the server memory to a client's address space using RDMA. This means that various things that are normally only possible in shared memory can be done in a distributed setting. Some examples include spawning a fiber, joining a fiber and accessing a shared object.

One inherent problem that restricts performance in distributed settings is the higher latency caused by network traversal. *Gossamer* overcomes that by using RDMA along with taking advantage of fibers. RDMA provides sub-micro second one way latency that is comparable to that of permanent storage on single machine. While RDMA solves the problem with high latency, it does not scale well with increasing number of connections per machine. To solve this *Gossamer* establishes only one connection per hardware thread for each remote machine and uses many fibers to utilize the available throughput to full extent.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 3 introduces *Trust*<*T*>, a delegation based programming abstraction that provides safe access to shared data. Chapter

4 presents *Gossamer*, an extension of *Trust*<*T*>, that allows single-machine, multi-threaded applications built using *Trust*<*T*>, to run on a rack consisting of many machines with minimal changes. Chapter 5 discusses future work and provides a conclusion.

CHAPTER 2

BACKGROUND

2.1 Synchronizing shared memory systems

Locking suffers from a well-known scalability problem: as the number of contending cores grows, cores spend more and more of their time in contention, and less doing useful work. Consider a classical, but idealized lock, in which there are no efficiency losses due to contention. Here, the *sequential* cost of each critical section is the sum of (a) any wait for the lock to be released, (b) the cost of acquiring the lock, (c) executing the critical section, and (d) releasing the lock. Not counting any re-acquisitions on the same core, this must be at minimum one cache miss per critical section, in sequential cost. To make matters worse, this cache miss is incurred by an atomic instruction, effectively stalling the CPU until the cache miss is resolved (and in the case of a spinlock, until the lock is acquired).

Two main solutions to this problem exist. First, where the data structure permits, fine-grained locking can be used to split the data structure into multiple independently locked objects, thus increase parallelism, reduce lock contention and wait times. With the data structure split into sufficiently many objects, and *accesses distributed uniformly*, a fine-grained locking approach tends to offer the best available performance.

The second solution is various forms of delegation, where one thread has custody of the object, and applies critical sections on behalf of other threads. Ideally, this minimizes the sequential cost of each critical section without changing the data structure: there are no sequential cache misses, ideally no atomic instructions, but of course the critical sections themselves still execute sequentially.

Combining (13; 14; 15; 16; 17; 18; 19), is a flavor of delegation in which threads temporarily take on the role of *combiner*, performing queued up critical sections for other threads. Combining can scale better than locking in congested settings, but does not offer the full benefits of delegation as it makes heavy use of atomic operations, and moves data between cores as new threads take on the *combiner* role. Most recently, TCLocks (20) offers a fully transparent combining-based replacement for locks, by capturing and restoring register contents, and automatically pre-fetching parts of the stack. TClocks claims substantial benefits for extremely congested locks, and the backward compatibility is of course quite attractive. However, a cursory evaluation in §3.5 reveals that TCLocks substantially underperform regular locks beyond extremely high contention settings, and never approaches *Trust<T>* performance.

Beyond *combining*, delegation has primarily been explored in proof-of-concept or one-off form, with relatively immature programming abstractions. We propose *Trust<T>*, a full-fledged delegation API for the Rust language, which presents delegation in a type-safe and familiar form, while substantially outperforming the fastest prior work on delegation.

While delegation offers much higher throughput for congested shared objects, it does suffer higher latency than locking in uncongested conditions. To hide this latency, and make delegation competitive in uncongested settings, *Trust<T>* exposes additional concurrency to the application via asynchronous delegation requests and/or light-weight, delegation-aware user threads (*fibers*).

Lacking modularity is another common criticism of delegation: in FFWD (21), an early delegation design, delegated functions must not perform any blocking operations, which includes any further delegation calls. In *Trust<T>*, this constraint remains for the common case, as this typically offers the highest efficiency. However, *Trust<T>* offers several options for more modular operation. First, asynchronous/non-blocking delegation requests are not subject to this constraint - these requests may be safely issued in any context. Second, leveraging our light-weight user threads, we offer the option of supporting blocking calls in delegated functions, on an as-needed basis.

Finally, prior work on delegation has required one or more cores to be dedicated as delegation servers. While *Trust<T>* offers dedicated cores as one option, the *Trust<T>* runtime has every core act as a delegation server, again leveraging light-weight user threads. Beyond easing application development and improving load balancing, having a delegation server on every core allows us to implement *Trust<T>* without any use of atomic instructions, instead relying on delegation for all inter-thread communication. Beyond potential performance advantages, this also makes *Trust<T>* applicable to environments where atomic operations are unavailable.

2.2 Distributed Systems

Just as multithreaded applications exploit multiple cores within a single machine, distributed systems exploit resources across multiple machines. To support this model, a variety of distributed systems frameworks have emerged, each providing abstractions and tooling to handle communication, coordination, and fault tolerance. Below we discuss a few of these briefly.

Remote Procedure Call (RPC) is a widely used model for communication in datacenters (2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12). RPCs provide the useful abstraction of running a function on a remote machine, while hiding the underlying network communication. For most approaches to RPC, the function needs to be registered at the start and calls to it can be either synchronous or asynchronous. A synchronous call will wait for a response while an asynchronous call can provide a callback function to be executed once the response is ready and continue execution. This is similar to delegation, rather than moving data to where it is needed, the code that uses the data is sent to its location. This process typically involves serializing and deserializing of data and transport protocols that ensure reliable message delivery. By decoupling distributed computation from explicit message handling, RPC simplifies the development of networked applications, supports modular system design, and serves as a foundational mechanism in many modern distributed architectures, including microservices and cloud-based systems.

Message Passing is another dominant communication framework (22; 23; 24; 25; 26). Unlike RPC, Message Passing has explicit communication and involves sending and receiving messages. Depending on the implementation, there can be guarantees regarding the reliability of message transport or the order of messages, or it can be a send-and-forget system where the sender might not even know if the message reached its destination. MPI (27) a widely used standard that provides a range of communication primitives, from one-to-one messages to many-to-many. It also provides both blocking and non blocking versions of send/receive.

There is also ongoing research to minimize resource usage of communication as much as possible. Remote Direct Memory Access (RDMA) is one such attempt (28; 29; 30; 31). RDMA allows the user to read a remote machine's memory by utilizing specialized Network Interface Cards (NIC) while bypassing the remote machine's operating system. Additionally, there is also some work on using a custom networking stack for distributed systems as compared to a hardware accelerated RDMA system (32; 33; 34).

CHAPTER 3

***Trust*<*T*>**

Trust<*T*> is a general, type- and memory-safe alternative to locking in concurrent programs. Instead of synchronizing multi-threaded access to an object of type T with a lock, the programmer may place the object in a *Trust*<*T*>. The object is then no longer directly accessible. Instead, a designated thread, the object's *trustee*, is responsible for applying any requested operations to the object, as requested via the *Trust*<*T*> API.

While locking offers a limited throughput *per lock*, *Trust*<*T*> is based on delegation, a message-passing technique which does not suffer this per-lock limitation. Instead, per-object throughput is limited by the capacity of the object's trustee, which is typically considerably higher.

Our evaluation shows *Trust*<*T*> consistently and considerably outperforming locking where lock contention exists, with up to 22× higher throughput in microbenchmarks, and 5–9 × for a home grown key-value store, in situations with high lock contention. Moreover, *Trust*<*T*> is competitive with locks even in the absence of lock contention.

3.1 Introduction

Safe access to shared objects is fundamental to many multi-threaded programs. Conventionally, this is achieved through *locking*, or in some cases through carefully designed lock-free data structures, both of which are implemented using atomic compare-and-swap (CAS) operations. By their nature, atomic instructions do not *scale* well: atomic instructions must not be reordered with other instructions, often starving part of today's highly parallel CPU pipelines of work until the instruction has retired. This effect is exacerbated when multiple cores are accessing the same object, resulting in the combined effect of frequent cache misses and cores waiting for each other to release the cache line in question, while the atomic instructions prevent them from doing other work.

Delegation (15; 35; 36; 16; 14; 17; 18; 19; 37; 21), also known as message-passing or light-weight remote procedure calls (LRPC), offers a highly scalable alternative to locking. Here, each shared object[1] is placed in the care of a single core (*trustee* below). Using a shared-memory message passing protocol, other cores (clients) issue requests to the trustee, specifying operations to be performed on the object.

Compared to locking, where threads typically contend for access, and may even suspend execution to wait for access, delegation requests from different clients are submitted to the trustee in parallel and without contention. This dramatically reduces the cost of coordination for congested objects. The operations/critical sections are applied sequentially in both designs: by each thread using locks, or by the trustee using delegation; here delegation may benefit from improved locality at the trustee. Together, this translates to much higher maximum per-object throughput with delegation vs. locking.

---

[1] Here, we use *object* to mean a data structure that would be protected by a single lock.

However, under medium or low contention, classical delegation struggles to compete with locking: the latency and overhead of request transmission, request processing and response transmission are insignificant compared to the cost of contending for a lock, but can be substantial compared to the cost of acquiring an *uncontended* lock.

3.2 *Trust<T>*: The Basics

```rust
let ct = local_trustee().entrust( 17 );            // ct: Trust<i32>
ct.apply( |c| *c+=1 );                              //  c: &mut i32
assert!(ct.apply( |c| *c ) == 18 );
```

Figure 1: Minimal *Trust<T>* example. An entrusted counter, referenced by `ct` is initialized to 17, then incremented once. The comments on the right indicate the types of the variables.

The objective of *Trust<T>* is to provide an intuitive API for safe, efficient access to shared objects. Naturally, our design motivation is to support delegation, but the *Trust<T>* API can in principle also be implemented using locking, or a combination of locking and delegation. Below, we first introduce the basic *Trust<T>* programming model, as well as the key terms *trust, property, trustee* and *fiber* in the *Trust<T>* context, before digging deeper into the design of *Trust<T>*.

3.2.1 Trust: a reference to an object

A *Trust<T>* is a thread-safe reference counting smart-pointer, similar to Rust's *Arc<T>*. To create a *Trust<T>*, we clone an existing *Trust<T>* or `entrust` a new object, or `property` of type T, that is meant to be shared between threads. Once entrusted, the property can only be accessed by *applying* closures to it, using a trust. Figure Figure 1 illustrates this through a minimal Rust example. Line 1 entrusts an integer, initialized to 17, to the local trustee - the trustee fiber running on the current kernel thread. Line 2 applies an anonymous closure to the counter, via the trust. The closure expected by `apply` takes a mutable reference to the property as argument, allowing it unrestricted access to the property, in this case, our integer. The example closure increments the value of the integer. The assertion on line 3 is illustrative only. Here, we apply a second closure to retrieve the value of the entrusted integer[1].

```
1 let ct = local_trustee().entrust(17);
2 let ct2 = ct.clone();
3 let thread = spawn(move || {
4  ct2.apply(|c| *c+=1);
5 });
6 ct.apply(|c| *c+=1);
7 thread.join()?;
8 assert!(ct.apply(|c| *c) == 19);
```

(a) Example using *Trust<T>*.

```
1 let cm = Arc::new(Mutex::new(17));
2 let cm2 = cm.clone();
3 let thread = spawn(move || {
4  *(cm2.lock()?) += 1;
5 });
6 *(cm.lock()?) += 1;
7 thread.join()?;
8 assert!(*cm.lock()? == 19);
```

(b) The same program using standard Rust primitives.

Figure 2: Minimal multi-threaded *Trust<T>* example. Reference counting ensures that the property remains in memory until the last *Trust<T>* referencing the property drops.

[1] A note on ownership: While the passed-in closure takes only a reference to the property, the Rust syntax `*c` denotes an explicit dereference, essentially returning a copy of the property to the caller. This will pass compile-time type-checking only for types that implement `Copy`, such as integers.

In the example in 2a the counter is instead incremented by two different threads. Here, the `clone()` call on `ct` (line 2) clones the trust, but not the property; instead a reference count is incremented for the shared property, analogous to `Arc::clone()`. On line 3, a newly spawned thread takes ownership of `ct2`, in the Rust sense of the word, then uses this to apply a closure (line 4). When the thread exits, `ct2` is dropped, decrementing the reference count, by means of a delegation request. When the last trust of a property is dropped, the property is dropped as well.

For readers unfamiliar with Rust, Figure 2b illustrates the rough equivalent of Figure 2a, but using conventional Rust primitives instead of *Trust<T>*. Note the similarity in terms of legibility and verbosity.

3.2.2 Trustee - a thread in charge of entrusted properties

In our examples above, *Trust<T>* is implemented using delegation. Here, a *property* is *entrusted* to a *trustee*, a designated thread which executes applied closures on behalf of other threads. In the default *Trust<T>* runtime environment, every OS thread in use already has a trustee user-thread (*fiber*) that shares the thread with other fibers. When a fiber applies a closure to a trust, this is sent to the corresponding trustee as a message. Upon receipt, the trustee executes the closure on the property, and responds, including any closure return value. This may sound complex, yet the produced executable code substantially outperforms locking in congested settings.

A TrusteeReference API is also provided. Here, the most important function is `entrust()`, which takes a property of type `T` as argument (by value), and returns a *Trust<T>* referencing the property that is now owned by the trustee. This API allows the programmer to manually manage the allocation of properties to trustees, for performance tuning or other purposes. Alternatively, a basic thread pool is provided to manage distribution of fibers and variables across trustees.

3.2.3 Fiber - a delegation-aware, light-weight user thread

While the *Trust<T>* abstraction has some utility in isolation, it is most valuable when combined with an efficient message-passing implementation and a user-threading runtime. User-level threads, also known as coroutines or *fibers*, share a kernel thread, but each execute on their own stack, enabling a thread to do useful work for one fiber while another waits for a response from a trustee. This includes executing the local trustee fiber to service any incoming requests.

In this default setting, the synchronous `apply()` function suspends the current fiber when it issues a request, scheduling the next fiber from the local ready queue to run instead. The local fiber scheduler will periodically poll for responses to outstanding requests, and resume suspended fibers as their blocking requests complete.

3.2.4 Delegated context

For the purpose of future discussion, we define the term *delegated context* to mean the context where a delegated closure executes. Generally speaking, closures execute as part of a trustee fiber, on the trustee's stack. Importantly, blocking delegation calls are not permitted from within delegated context, and will result in a runtime assertion failure. In §3.3, we describe multiple ways around this constraint.

3.3 Core API

The *Trust<T>* API supports a variety of ways to delegate work, some of which we elide due to space constraints. Below, we describe the core functions in detail.

3.3.1 `apply()`: synchronous delegation

```
apply(c: FnOnce(&mut T)->U)->U
```

`apply()` is the primary function for blocking, synchronous delegation as described in earlier sections. It takes a closure of the form `|&mut T|` {}, where `T` is the type of the property. If the closure has a return value, apply returns this value to the caller.

Importantly, `apply()` is synchronous, suspending the current fiber until the operation has completed. Often, the best performance with `apply()` is achieved when running multiple application fibers per thread. Then, while one fiber is waiting for its response, another may productively use the CPU.

3.3.2 apply_then(): non-blocking delegation

```
apply_then(c: FnOnce(&mut T)->U,
           then: FnOnce(U))
```

```
let ct = trustee.entrust(17);              // create trust for shared counter set to 17
ct.apply_then(|c| { *c+=1; *c },           // increment counter and return its value
              |val| assert!(val==18));     // check return value once received
```

Figure 3: Asynchronous version of the example in Figure 1. The second closure runs on the client, once the result of the first closure is received from the trustee.

Frequently, asynchronous (or non-blocking) application logic can allow the programmer to express additional concurrency either without running multiple fibers, or in combination with multiple fibers. Here, `apply_then()` returns to the caller without blocking, and does not produce a return value. Instead, the second closure, `then`, is called with the return value from the delegated closure, once it has been received. Figure 3 demonstrates the use of `apply_then()` following the pattern of Figure 1.

The `then`-closure is a very powerful abstraction, as it too is able to capture variables from the local environment, allowing it to perform tasks like adding the return value (once available) to a vector accessible to the caller. Here, Rust's strict lifetime rules automatically catch otherwise easily introduced use-after-free and dangling pointer problems, forcing the programmer to appropriately manage object lifetime either through scoping or reference counted heap storage.

Importantly, as `apply_then()` does not suspend the caller, it may freely be called from within delegated context.

3.3.3 `launch()`: apply in a trustee-side fiber

```
launch(c: FnOnce(&mut T)->U)->U

launch_then(c: FnOnce(&mut T)->U,

          then: FnOnce(U))
```

The most significant constraint imposed by *Trust<T>* on the closure passed to `apply()` and `apply_then()` is that the closure itself may not block. Blocking in delegated context means putting the trustee itself to sleep, preventing it from serving other requests, potentially resulting in deadlock. In previous work (38), this problem was addressed by maintaining multiple server OS threads, and automatically switching to the next server when one server thread blocks. This avoids blocking the trustee, but imposes high overhead, resulting in considerably lower performance, as demonstrated in (21).

In *Trust<T>*, blocking in delegated context is prohibited: attempted suspensions in delegated context are detected at runtime, resulting in an assertion failure. Closures may still use `apply_then()`, but not the blocking `apply()`. [1]

The lack of *nested blocking delegation* can be a significant constraint on the developer, and perhaps the most important limitation of *Trust<T>*. Specifically, it affects modularity, as a

[1]Other forms of blocking, such as I/O waits or scheduler preemption, do not result in assertion failures. However, these can significantly impact performance if common, as blocking the trustee can prevent other threads from making progress.

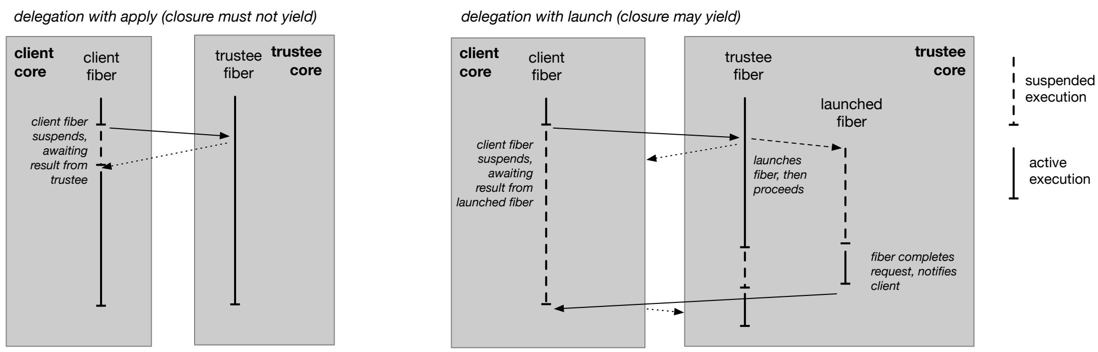


Figure 4: Operation of `launch()` vs `apply()`. `launch()` supports blocking calls, including nested delegation calls in the delegated closure, but incurs a higher minimum overhead. Solid arrows indicate requests, dotted arrows are delegation responses.

library function that blocks internally, even on delegation calls, cannot be used from within delegated context.

To address this, without sacrificing the performance of the more common case, we provide a convenience function: `launch()`, which offers all the same functionality as `apply()`, but without the blocking restriction. Figure 4 describes `launch()` from an implementation standpoint. `launch()` creates a temporary fiber on the trustee's thread, which runs the closure. If this fiber is suspended, the client is notified, and the trustee continues to serve the next request. Once the temporary fiber resumes and completes execution of the closure, it then delivers the return value and resumes the client fiber via a second delegation call. Thus, if a delegated closure fails the runtime check for blocking calls, the developer can fix this by replacing the `apply()` call, with a `launch()` call.

3.3.3.1 Atomicity and `launch()`

That said, a complicating factor with blocking closures executed by `launch()` is that without further protection, property accesses are no longer guaranteed to be atomic: while the newly created fiber is suspended, another delegation request may be applied to the property, resulting in a race condition. To avoid this risk, `launch()` is implemented only for `Trust<Latch<T>>`. `Latch<T>` is a wrapper type which provides mutual exclusion, analogous to `Mutex<T>` except that it uses no atomic instructions, and thus may only be accessed by the fibers of a single thread.[1]

3.3.3.2 Leveraging Rust for safe and efficient delegation

Using the Rust type system, we ensure that delegated closures in *Trust*$<T>$ cannot capture values that contain any references or pointers.

In principle, this is far stricter than what is necessary: the existing and pervasive Rust traits `Send` and `Sync` already describe the types that may be safely moved and shared between threads, and this continues to hold within *Trust*$<T>$.

That said, safety does not guarantee performance. A common performance pitfall when writing delegation-based software is memory stalls on the trustee, which affects trustees disproportionately due to the polling nature of the delegation channel (see §3.4.1). Frequent cache misses and use of atomic instructions in delegated closures can substantially degrade trustee throughput vs. running closures with good memory locality.

[1] In Rust terms, `Latch<T>` does not implement `Sync`.

Generally speaking, cache line contention and use of atomic instructions are a natural result of sharing memory between threads. By prohibiting the capture of references and pointers, *Trust<T>* makes accidental shared memory patterns of programming much less likely in delegated code, and encourage pass-by-value practices.

3.3.3.3 Variable-size and other heap-allocated values

Rust closures very efficiently and conveniently capture their environment, which `apply()` sends whole-sale to the trustee. However, only types with a size known at compile time may be captured in a Rust closure (or even allocated on the stack).

In conventional Rust code, variable size types, including strings, are stored on the heap, and referenced by a `Box<T>` smart pointer. For the reasons described above (see §3.3.3.2), we do not allow `Box<T>` or other types that include pointers or references to be captured in a closure: only pure values may pass through the delegation channel.

As a result, variable size objects and other heap-allocated objects must be passed as explicit arguments rather than captured, so that they may be serialized before transmission over the delegation channel. For example, a `Box<[u8]>` (a reference to a heap-allocated variable-sized array of bytes) cannot traverse the delegation channel. Instead, we encode a copy of the variable number of bytes in question into the channel, and pass this value to the closure when it is executed by the trustee. In practice, this takes the form of a slightly different function signature.

```
apply_with(c: FnOnce(&mut T, V)->U, w: V)->U
```

Here, the `w:` argument is any type `V:Serialize+Deserialize`, using the popular traits from the `serde` crate. That is, any type that can be serialized and deserialized, may pass over the delegation channel in serialized form. If more than one argument is needed, these may be passed as a tuple. Thus, to insert a variable-size key and value into an entrusted table, we might use:

```
table_trust.apply_with(|table, (key, value)|
  table.insert(key,value),(key,value))
```

We use the efficient `bincode` crate internally for serialization. As a result, while passing heap-allocated values does incur some additional syntax, the impact in terms of performance is minimal.

3.4 Key Design and Implementation Details

In this section, we delve deeper into the design and implementation of *Trust*$<$*T*$>$, from the mechanics of delegating Rust closures and handling requests and responses, to asynchronous versions of `apply()`.

3.4.1 Delegating Closures

The key operation supported by *Trust*$<$*T*$>$ is `apply()`, which applies a Rust *closure* to the property referenced by the trust. A Rust closure consists of an anonymous function and a captured environment, which together is represented as a 128-bit *fat pointer*. Thus, to delegate a closure, a request must at minimum contain this fat pointer, and a reference to the property in question.

One or more requests are written to the client's dedicated, fixed-sized *request slot* for the appropriate trustee. That is, only the client thread may write to the request slot. For efficiency, if the captured environment of the closure fits in the request slot, we copy the environment directly to the slot, and update the fat pointer to reflect this change. A flag in the request slot indicates that new requests are ready to be processed. See §3.4.3 for details on request and response slot structure.

Responses are transmitted in a matching dedicated response slot. Leveraging the Rust type system, we restrict both requests and responses to types that can be serialized. The subtle implication of this is that the return value may not pass any references or pointers to trustee-managed data.[1] While small closures with simple, known-and-fixed-size return types will generally yield the best performance, there is no limit beyond the serializability requirement on the size or complexity of closures and return types.

3.4.2 Scheduling Delegation Work

Generally speaking, a call to `apply()` appends a request to a pending request queue, local to the requesting thread. In the case of `apply()`, the calling fiber is then suspended, to be woken up when the response is ready. Pending requests are sent during response polling, and as soon as an appropriate request slot is available. The intervening time is spent running other fibers, including the local trustee fiber, and polling for responses/transmitting requests.

[1] That said, we cannot prevent determined Rust programmers from using `unsafe` code to circumvent this restriction.

There is a throughput/latency trade-off between running application fibers, and polling for requests/responses: poll too often, and few requests/responses will be ready, wasting polling effort. Poll too seldom, and many requests/responses will have been ready for a long time, increasing latency. Automatically tuning this trade-off is an area of ongoing research. That said, the current implementation performs delegation tasks in a fiber that is scheduled in FIFO order just as other fibers. After serving incoming requests, this fiber polls for incoming responses and issues any enqueued outgoing requests as applicable.

3.4.2.1 Local Trustee Shortcut

When a Trust has the current thread as its trustee, it is superfluous to use delegation to apply the closure. Instead, it is just as safe, and more efficient, to simply apply the closure directly, since we know that no other closures will run until the provided closure has run to completion. As a reminder, we know this because delegated closures may not suspend the current fiber.

3.4.3 Request and Response Slot Structure

Figure Figure 5 illustrates the internal structure of the basic request and response slot design. A header consisting of a `ready` bit and a request count, is followed by a variable number of variable-sized requests. The value of the `ready` bit is used to indicate whether a new request or set of requests has been written to the slot: if the bit differs from the `ready` bit in the corresponding response slot, then a new set of requests is ready to be processed.

By default, the slot size is 1152 bytes, and the client may submit as many closures as it can fit within the slot. Here, the minimum size of a request is 24 bytes: a 128-bit fat pointer for

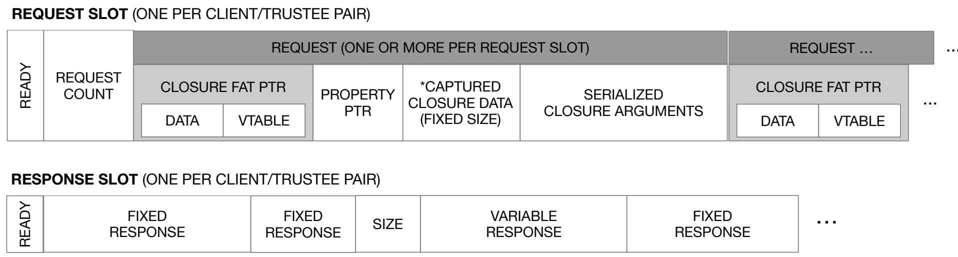


Figure 5: The fixed-size *Trust*< *T*> request slot consists of a `ready` bit, a request counter, and a variable number of variable-sized requests. The response slot contains a matching `bit`, as well as one (fixed, variable, or zero-sized) response per request in the matching request slot. There is one dedicated pair of request/response slots for each trustee/client pair.

the closure, and a regular 64-bit pointer for the property. The captured environment of Rust closures have a known, fixed size, which is found in the vtable of the closure. For typical small captured environments, this is copied into the request slot, and the pointer updated to point at the new location. Serialized closure arguments are appended next, followed by the next request.

Responses are handled in a similar fashion, though there is no minimum response size. Responses are sent simultaneously for all the requests in the request slot. The size of each response is often statically known, in which case it is not encoded in the channel. Any variable-size responses are preceded by their size.

The size of each request is always known, either statically or at the time of submission, which means we can restrict the number of requests sent to what can be accommodated by the request slot. The size of the response is not always known at the time the request is sent. In cases where the combined size of return values exceeds the space in the response slot, the trustee dynamically allocates additional memory to fit the full set of responses, at a small performance penalty.

3.4.3.1 Two-part slot optimization

In order to accommodate a broad range of application characteristics, including those with a single trustee and many clients, as well as a single client with many trustees, we introduce a small optimization beyond the basic design above. Rather than represent the request and response slots as monolithic blocks of bytes, we represent each as two blocks: a 128-byte primary block, and a 1024-byte overflow block; each request and response is written, in its entirety, to one or the other block.

This addresses an otherwise problematic trade-off with respect to the request and response slot sizes: with a monolithic request slot of, say, one kilobyte, the trustee would be periodically scanning flags 1024 bytes apart, a very poor choice from a cache utilization perspective, unless the slots are heavily utilized. A two-part design accommodates a large number of requests (where needed), but improves the efficiency of less heavily utilized request slots by spacing ready flags, and a small number of compact requests, more closely using a smaller primary request block.

3.4.4 Reference counting for Trusts

Trusts in *Trust*<*T*> act as rust smart pointers that own the property, since they can be used to access and modify the property behind them. This means that we need to make sure that when all of the trusts are dropped, the property is also dropped and the associated memory is freed so as not to have memory leakage. To achieve that trusts need to be reference counted, but a naive integer count will not suffice due to a combination of the following two issues.

- *Trust*<*T*> does not support a blocking delegation operation when in the middle of another delegation request. For example calling *apply* on a trust from within a closure that itself is used for delegation will cause the system to hang and never finish.
- If the integer used for counting references, let's call it refcount, is incremented and decremented asynchronously, i.e. with a nonblocking delegation call, it could lead to use after free bugs. Let's consider the following scenario: Thread A clones a trust with only one reference and sends an async request to increment its refcount. The cloned trust is then sent to Thread B that drops it, sending another async request to decrement its refcount. While the requests issued by the same fiber are guaranteed to be completed in the same order they are issued, there is no such guaranty across multiple threads or even fibers. It is entirely possible that the decrement request is processed first, making the refcount zero, which results in the property being dropped. Thread A however still has a valid trust to this property which it can use, expecting the property to still be available.

Not being able to use async delegation to increment and decrement the refcount leads to not being able to clone a trust from within a delegated context, which can quite restrictive. To get around that, *Trust*<*T*> uses a new way to keep track of how many trusts are active at any time. Each trust is given a unique id at the time of creation, be it a new trust or a clone of an existing one. Instead of just using an integer, *Trust*<*T*> uses a set of these ids associated with each property. Instead of incrementing or decrementing the refcount, clone and drop both issue a delegation request involving a symmetric difference operation. Symmetric difference is defined as adding an element to a set if it is not already a member, and removing it from the set if it is. This way, regardless of the order in which requests originating in clone and drop are processed, the first one will add the trust id to the set and later one will remove it, allowing us to use async delegation for both. This, in turn, enables us to clone a trust within a delegated context.

3.5 Evaluation

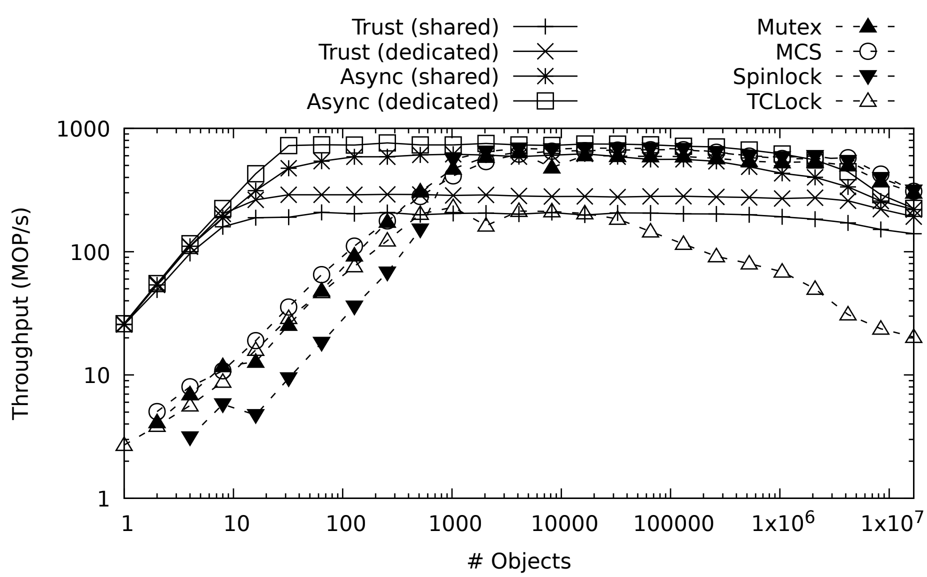


Figure 6: Fetch-and-add throughput vs. object count. Uniform access distribution.

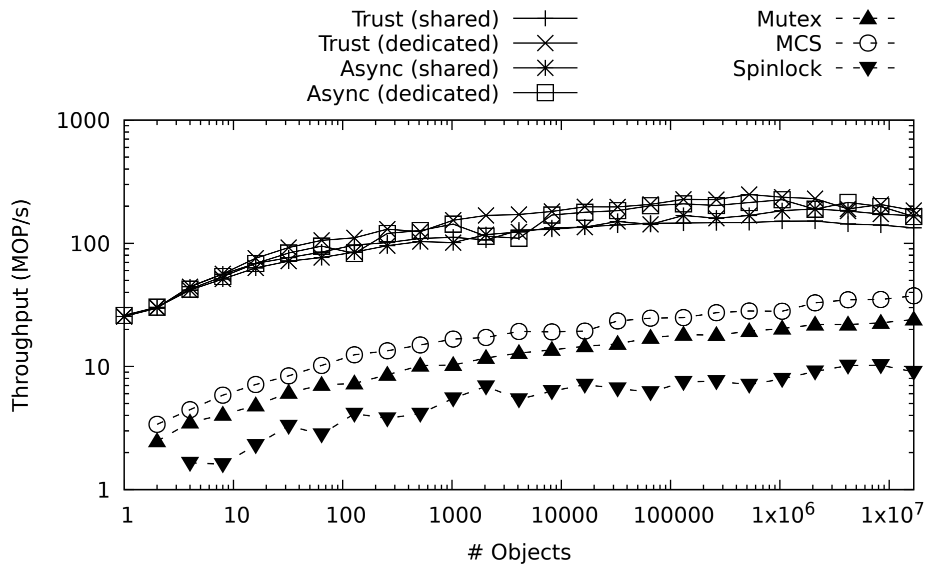


Figure 7: Fetch-and-add throughput vs. object count. Zipfian access distribution, $\alpha = 1$.

Below, we evaluate the performance of *Trust<T>* in two ways: 1) on both microbenchmarks, designed to stress test the core mechanisms behind *Trust<T>* and locking, and 2) on end-to-end application benchmarks, which measure the performance impact of *Trust<T>* in the context of a complete system and a more realistic use case.

3.5.1 Fetch and Add: Throughput

For our first microbenchmark, we use a basic fetch-and-add application. Here, a number of threads repeatedly increment a counter chosen from a set of one or more, and fetches the value of the counter. In common with prior work on synchronization and delegation (15; 35; 36; 16; 14; 17; 18; 19; 37), we also include a single `pause` instruction in both the critical section and the delegated closures. The counter is chosen at random, either from a uniform distribution, or a zipfian distribution. Each thread completes 1 million such increments. In this section, each data point is the result of a single run.

Below, we primarily evaluate on a two-socket Intel Xeon CPU Max 9462, of the Sapphire Rapids architecture. This machine has a total of 64 cores, 128 hyperthreads, and 384 GB of RAM. Unless otherwise noted, we use 128 OS threads. In testing, several older x86-64 ISA processors have shown similar trends – these results are not shown here. For locking solutions, we use standard Rust `Mutex<T>` and the spinlock variant provided by the Rust `spin-rs-0.9.8` crate, as well as MCSLock<T> provided by the Rust `synctools-0.3.2` crate. For `Trust<T>`, we show results for blocking delegation (Trust) as well as nonblocking delegation (Async). In Fig. Figure 6, we also include TCLocks, a recent combining approach offering a transparent replacement for standard locks, via the Litl lock wrapper (39) for `pthread_mutex`. To be able to evaluate this lock, we wrote a separate C microbenchmark, matching the Rust version. In the interest of an apples-to-apples comparison, we first verified that the reported performance with stock `pthread_mutex` on the C microbenchmark matched the Rust `Mutex<T>` performance in our Rust microbenchmark.

Below, the `Trust` results may be seen to represent any application with ample concurrency available in the form of conventional synchronous threads. `Async` represents applications where a single thread may issue multiple simultaneously outstanding requests, e.g. a key-value store or web application server. Applications with limited concurrency are not well suited to delegation, except where the delegated work is itself substantial, which is not the case for this fetch-and-add benchmark. We further report results with both letting all cores serve as both clients and trustees (`shared`), as well as with an ideal number of cores dedicated serve only as trustees (`dedicated`).

3.5.1.1 Uniform Access Pattern

Figure Figure 6 illustrates the performance of several solutions on the uniform distribution version of this benchmark. For a very small number of objects, no data points are reported for some of the lock types - this is because the experiment took far too long to run due to severe congestion collapse.

`Trust<T>` substantially outperforms locking under congested conditions. Between 1–16 objects, the performance advantage is 8–22× the best-performing MCSLock. For larger numbers of objects, the overhead of switching between fibers becomes apparent, as asynchronous delegation is able to reach a higher peak performance. In entirely uncongested settings, with 10× as many objects as there are threads, locking is able to match asynchronous delegation performance. TCLocks (20) was the only lock type to complete the single-lock experiment within a reasonable time. It consistently outperforms spinlocks under congestion, and remains competitive with Mutex and MCS on highly congested locks. However, TCLocks appear to trade their transparency for high memory and communication overhead, making it unable to compete performance-wise beyond highly congested settings. [1] Moreover, we struggled to apply TCLocks to memcached (which consistently crashed under high load), as well as to Rust

[1] TCLocks performance appears somewhat architecture dependent. In separate runs on our smaller Skylake machines, TCLocks were able to outperform Mutex by ≈50% under the most extreme contention (a single lock).

programs (as Rust now uses built-in locks rather than `libpthreads` wrappers). We thus elide TCLocks from the remainder of the evaluation.

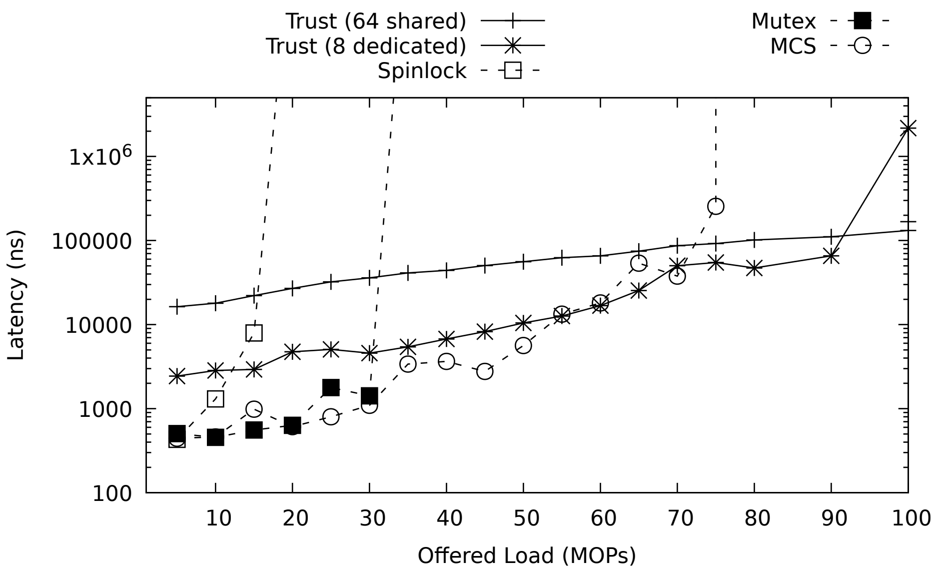


Figure 8: Mean latency vs. offered load. Uniform access distribution.

3.5.1.2 Skewed Access Pattern: Zipfian distribution

Zipf's law (40) elegantly captures the distribution of words in written language. In brief, it says that the probability of word occurrence p_w is distributed according to the rank r_w of the word, thus: $p_w \propto {r_w}^{-\alpha}$, where $\alpha \sim 1$. Similar relationships, often called "power laws", are common in areas beyond written language (40; 41; 42; 43; 44), sometimes with a greater

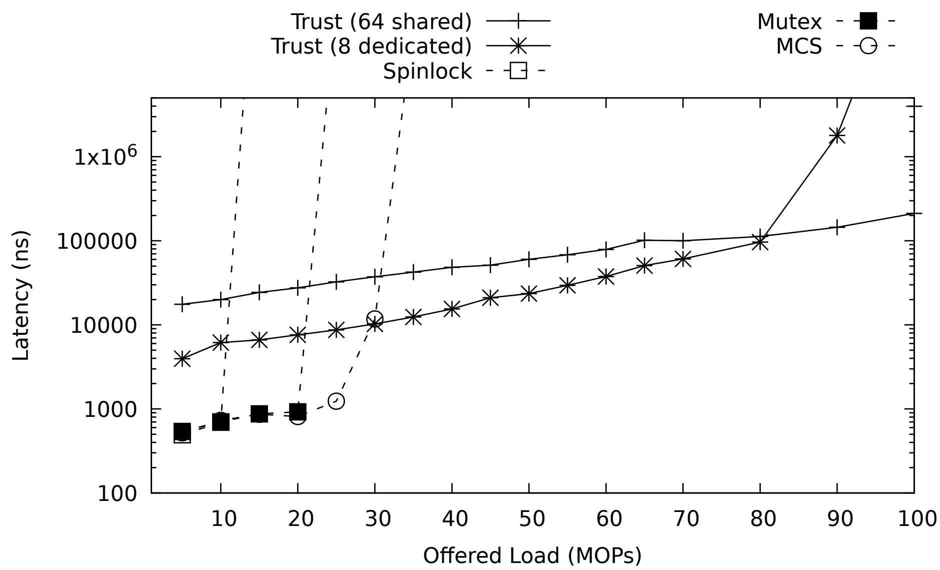


Figure 9: Mean latency vs. offered load. Zipfian access distribution, $\alpha = 1$.

value for α. The higher the α, the more pronounced is the effect of popular keys, resulting in congestion.

Figure Figure 7 shows the results of our fetch-and-add experiment, but with objects selected according to a zipfian distribution ($\alpha = 1$) instead of the uniform distribution above, representing a common skewed access distribution.

With this skewed access pattern, *Trust<T>* overwhelmingly outperforms locking across the range of table sizes. This is explained by the relatively low throughput of a single lock. In our experiments, even MCSLocks, known for their scalability, offer at best 2.5 MOPs. When a skewed access pattern concentrates accesses to a smaller number of such locks, low performance is inevitable. By comparison, a single *Trust<T>* trustee will reliably offer 25 MOPs, for similarly short critical sections. For more highly skewed patterns, where $\alpha > 1$ (not shown), the curve grows ever closer to the horizontal as performance is bottlenecked by a small handful of popular items.

3.5.2 Fetch and Add: Latency

Next we measure mean latency for a scenario with 64 objects (uniform access distribution), and 1,000,000 objects (Zipfian access distribution), while varying the offered load. We show delegation results with 8 dedicated trustee cores, and with 64 shared trustee cores[1]. We also plot the results for a spinlock, a standard Rust mutex, and an MCS lock as above.

At low load, low contention results in low latency for locking, an ideal situation for locks. However, as load increases, the locks eventually reach capacity, resulting in a rapid rise in latency. With *Trust<T>*, even low load incurs significant latency, due to message passing overhead. However, due to the much higher per-object capacity available, latency increases slowly with load until the capacity is reached. Thus, *Trust<T>* offers stable performance over a wide range of loads, at the cost of increased latency at low load. The higher latency does mean that to take full advantage of delegation, applications need to have ample parallelism available.

For both Uniform and Zipfian access distributions, we also measured 99.9th percentile (tail) latency (not shown). Overall, tail latency with locking (all types) tended to be approximately 10× the mean latency, in low-congestion settings. Delegation tail latency with a dedicated

[1] The evaluation system has 64 cores, 128 hardware threads. In the vast majority of cases, having both hardware threads of each core work as trustees results in reduced performance.

trustee, meanwhile, was 2.5× the mean, making delegation tail latency under low load only 2–3× that of locking.

It's also worth noting the difference between 8 dedicated trustees, and 64 trustees on threads shared with clients. The latency when sharing the thread with clients is naturally higher than when using trustees dedicated to trustee work. However, as load increases having more trustees available to share the load results in better performance. Using all the cores for trustees all the time also eliminates an important tuning knob in the system.

3.5.3 Concurrent key-value store

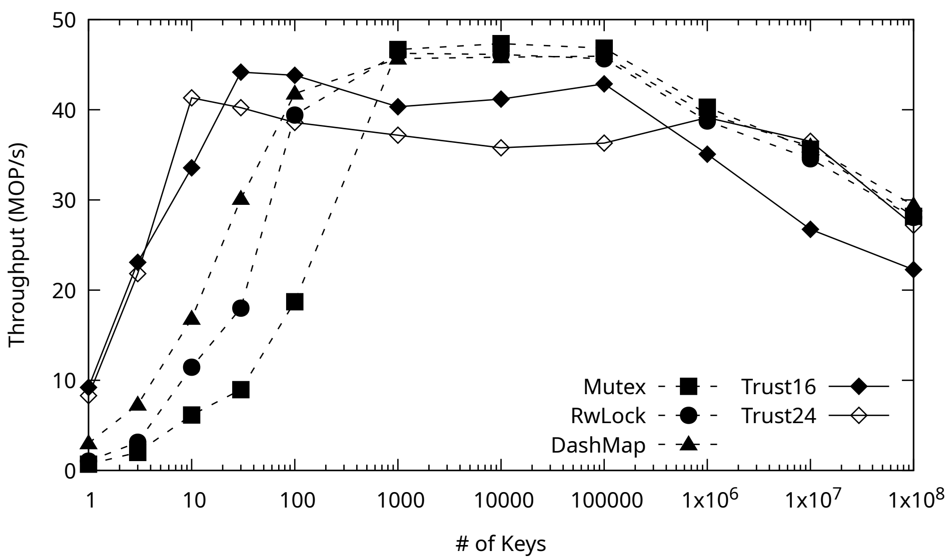


Figure 10: Key-value store throughput, with 5% writes and varying table size. Uniform access distribution

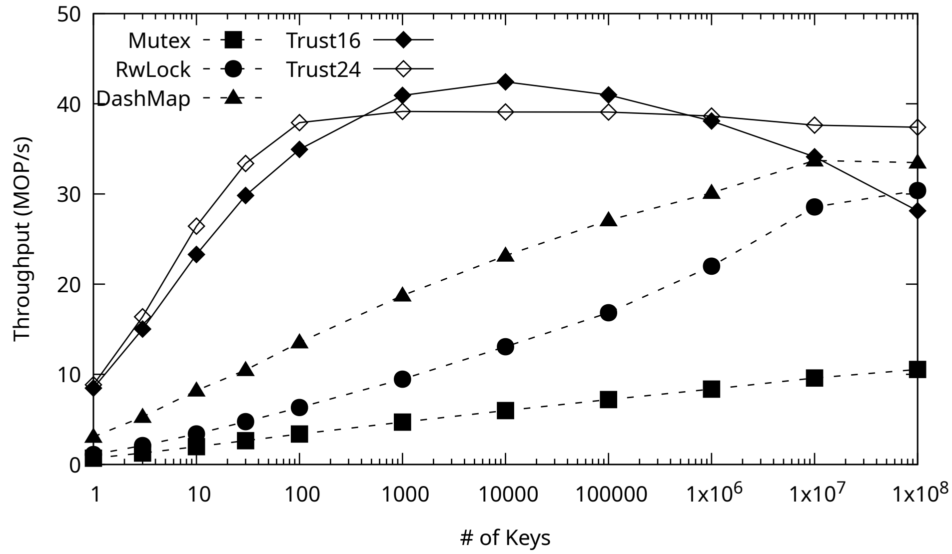


Figure 11: Key-value store throughput, with 5% writes and varying table size. Zipfian access distribution

For a more complete end-to-end evaluation, we implement a simple TCP-based key-value store, backed by a concurrent dictionary. Here, we run a multi-threaded TCP client on one machine, and our key-value store TCP server on another, identical machine. The two machines are connected by 100 Gbps Ethernet. We compare our *Trust*<*T*> based solution to Dashmap (45), one of the highest-performing concurrent hashmaps available as a public Rust crate, as well as to our own naïvely sharded Hashmap, using Mutex or Readers-writer locks and the Rust `std::collections::HashMap<K, V>`. Dashmap is a heavily optimized and well-respected hash table implementation, which is regularly benchmarked against competing designs.

We implement the key-value store as a multi-threaded server, where each worker-thread receives `GET` or `PUT` queries from one or more connections, and applies these to the back-end

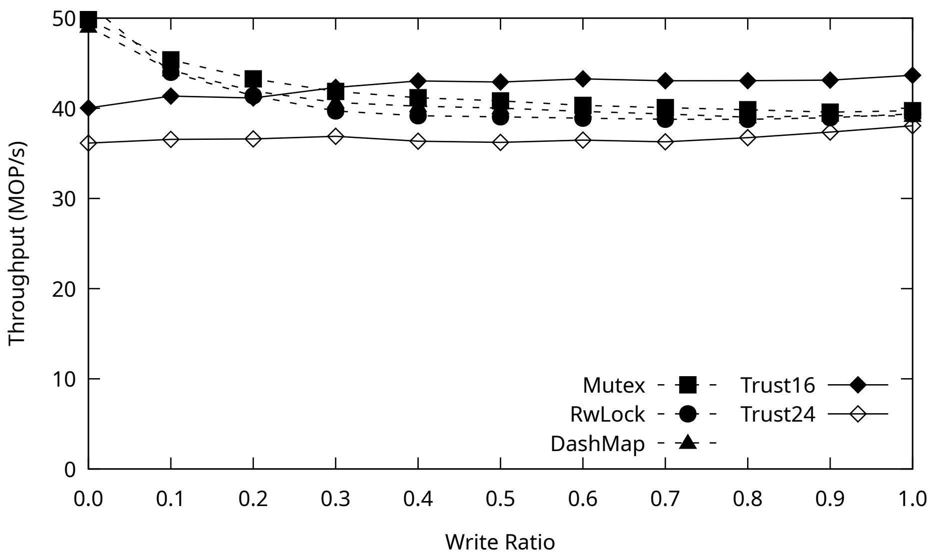


Figure 12: Key-value store throughput, with varying write percentage. Uniform access distribution

hashmap. Both reading requests and sending results is done in batches, so as to minimize system call overhead. Moreover, the client accepts responses out-of order, to minimize waiting. The TCP client continuously maintains a queue of parallel queries over the socket, such that the server always has new requests to serve. In the experiments, we dedicate one CPU core to each worker thread.

For our sharded hashmaps, we create a fixed set of 512 shards, using many more locks than threads to reduce lock contention. Dashmap uses sharding and readers-writer locks internally, but exposes a highly efficient concurrent hashmap API. For our *Trust*<*T*> based key-value store, we use 16 and 24 cores to run trustees (each hosting a shard of the table) exclusively, and the remaining cores for socket workers. They are named Trust16 and Trust24, respectively.

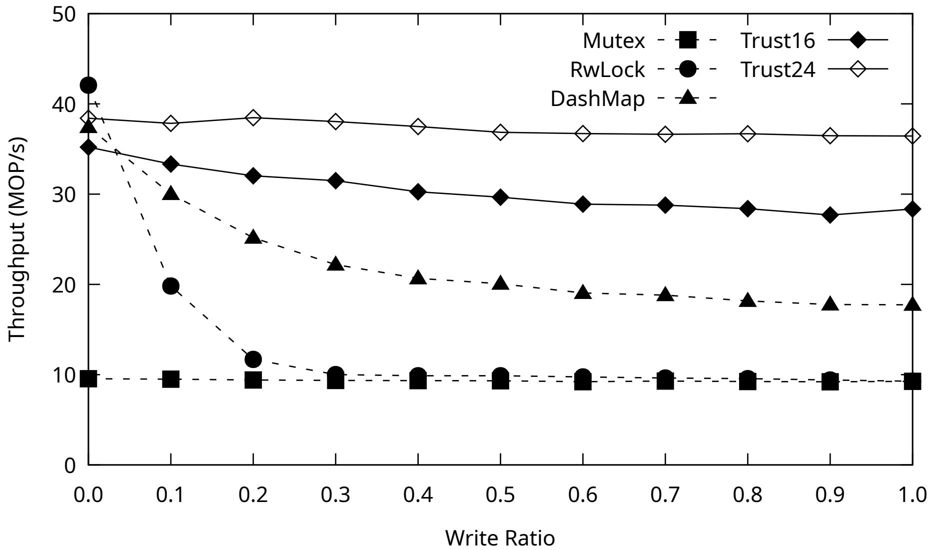


Figure 13: Key-value store throughput, with varying write percentage. Zipfian access distribution

Socket workers delegate all hash table accesses to trustees. The key size is 8 bytes and the value size is 16 bytes in the experiments. Prior to each run, we pre-fill the table, and report results from an average of 10 runs.

Figures Figure 10–Figure 11 show the results from this small key-value store application, for a varying total number of keys with 5% write requests and 95% read request, and Uniform as well as Zipfian (40) access distributions. For Zipfian access, we use the conventional $\alpha = 1$. Overall, similar to the microbenchmark results, we find that the delegation-based solution performs significantly better when contention for keys is high. However, due to the considerably higher complexity of this application, the absolute numbers are lower than in our microbenchmarks.

The relative advantage for delegation is also somewhat smaller, as some parts of the work of a TCP-based key-value store are already naturally parallel.

For the Uniform distribution and 5% writes, all the solutions perform similarly above 1,000 keys, a large enough number that there is no significant contention. With 100 keys and less, *Trust<T>* enjoys a large advantage even under uniform access distribution. With a Zipfian access distribution, accesses are concentrated to the higher-ranked keys, leading to congestion. In this setting, *Trust<T>* trounces the competition, offering substantially higher performance across the full 1–100,000,000 key range. It is interesting to note, also, that the Zipfian access distribution is where the carefully optimized design of Dashmap shines, while it offers a fairly limited advantage over a naïve sharded design with readers-writer locks on uniform access distributions. This speaks to the importance of efficient critical sections in the presence of lock congestion.

The througput of Trust16 is higher than Trust24 with 1,000–100,000 keys because it is of low cost to manage a relatively small key space, while Trust16 can dedicate more resources to handle socket connections. However, the performance of Trust16 starts to degrade with more keys, because the limited number of trustees fall short when managing larger key spaces. With 24 trustees, the performance can be maintained at a high level. The difference between Trust16 and Trust24 suggests an important direction of future research. For I/O heavy processes like key-value stores, dedicated trustees will often outperform sharing the core between trustees and clients. However, it is non trivial to correctly choose the number of trustees. Automatically adjusting the number of cores dedicated to trustee work at runtime would be preferable.

In principle, readers-writer locks have a major advantage over *Trust<T>* in that they allow concurrent reader access, while *Trust<T>* exclusively allows trustees to access the underlying data structure. To better understand this dynamic, Figures Figure 12–Figure 13 show key-value store throughput over a varying percentage of writes.

Here, we use 1,000 keys for the Uniform access distribution, and 10,000,000 keys for Zipfian access distribution. We note that these are table sizes where lock-based approaches hold an advantage in Figures Figure 10–Figure 11. For Uniform access patterns, where there is limited contention given the table size of 1,000 keys, the impact of the write percentage is muted. For lock-based designs, the performance does drop somewhat, but remains at a high level even with 100% writes.

It is interesting to note that *Trust<T>* performance increases modestly with the write percentage. One reason behind this is that in our key-value store, the closures issued by reads by necessity have large return values, while the closures issued by writes have no return values at all. This may allow the trustee to use only the first, small part of the return slot, occasionally saving two LLC cache misses per round-trip.

With the Zipfian access distribution, even with 10,000,000 keys, contention remains a bigger concern, especially for Mutex. All four designs exhibit reduced performance with increased write percentages, but again, *Trust<T>* proves more resilient. The efficiency advantage of Dashmap over our naïve lock-based designs is on full display with the Zipfian access distribution and a high write percentage. That said, the fundamental advantage of *Trust<T>* over locking in this application is clear.

3.6 Legacy Application: Memcached

We also port memcached version 1.6.20 to *Trust*<*T*> to demonstrate both the applicability and performance impact on legacy C applications. Memcached is a multi-threaded key-value store application. Its primary purpose is serving PUT and GET requests with string keys and values over standard TCP/IP sockets. Internally, memcached contains a hash-table type data structure with external linkage and fine-grained per-item locking. By default, memcached is configured to use a fixed number of worker threads. Incoming connections are distributed among these worker threads. Each worker thread uses the `epoll()` system call to listen for activity on all its assigned connection. Each connection to a memcached server traverses a fairly sophisticated state machine, a pipelined design that is aimed at maximizing performance when each thread serves many concurrent connections with diverse behaviors. The state machine will process requests in this sequence: receive available incoming bytes, parse one request, process the request, enqueue the result for transmission, and transmit one or more results.

For our port to *Trust*<*T*>, we eliminate the use of most locks, and instead divide the internal hash table and supporting data structures into one or more shards, and delegate each shard to one of potentially multiple trustees. Thus, instead of acquiring a lock, we delegate the critical section to the appropriate trustee for the requested operation. Our ported version follows the original state machine design, with one key difference: for each incoming request on the socket, we make an asynchronous delegation request using `apply_then`, then move on to the next request without waiting for the response from the trustee. That is, rather than sequentially process each incoming request, we leverage asynchronous delegation to capture additional concurrency.

A complicating factor in this asynchronous approach results from `memcached` being initially designed for synchcronous operation with locking. For any one trustee-client pair, even asynchronous delegation requests are executed in-order, and responses arrive in-order. However, this is not guaranteed for requests issued to different trustees. Consequently, the memcached socket worker thread must order the responses before they are transmitted over the network socket to the remote client. By contrast, our delegation-native key-value store in 3.5.3 sends responses out of order over the socket, and instead includes a request ID in the response.

Another difference worth mentioning is that we don't allow delegation clients (in this case, the memcached socket worker thread) to access delegated data structures at all. This means that instead of a pointer to a value in the table, clients recieve a copy of the value. This significantly improves memory locality and simplifies memory management, since every value has a single owner. However, it does incur extra copying, which may reduce performance under some circumstances.[1]

In practice, because memcached is written in C and *Trust*<*T*> is written in Rust, we cannot directly add delegation to the memcached source code. We address this in a two-step process: first, for any task that requires delegation, we create a minimal Rust function that performs that specific task. That is, a custom Rust function that becomes part of the memcached code base.

---

[1] This can become a problem when values are large. For this use case, *Trust*<*T*> includes an equivalent of Rust's `Arc<T>` which allows multiple ownership of read-only values.

Typically, such a function locates the appropriate Trust or TrusteeReference, and delegates a single closure. Second, we break out the critical sections in the C code into separate inner functions that may be called from Rust. Thus, to delegate a C critical section, we simply call the inner function from a delegated Rust closure.

Our port of Memcached to *Trust<T>* has approximately 600 lines of added, deleted or modified lines of code, out of 34,000+ lines total. This number includes approximately 200 of lines which were simply cut-and-pasted into the new inner functions for critical sections. In addition, we introduced approximately 350 new lines of Rust code, to provide the interface between the C and Rust environments.

3.6.1 Evaluation

To understand the performance of our delegated Memcached, we use the memtier benchmark client (version 1.4.0) with our delegated Memcached as well as stock memcached. For the cleanest results, but without loss of generality, we configure memcached with a sufficiently large hash power and available memory to eliminate table resizing and evictions. We also limit our evaluation to the conventional memcached PUT/GET operations. Recent versions of memcached feature a optional new cache eviction scheme, which trades less synchronization for the need for a separate maintenance thread. For stock memcached, we evaluated both the traditional eviction scheme and the new one. We show results for the new scheme, which scales much better for write-heavy workloads and is otherwise similar in our setting. For our ported version, we use the traditional eviction scheme, maintaining one LRU per shard. Eviction is not relevant here, as we provide ample memory relative to the table size.

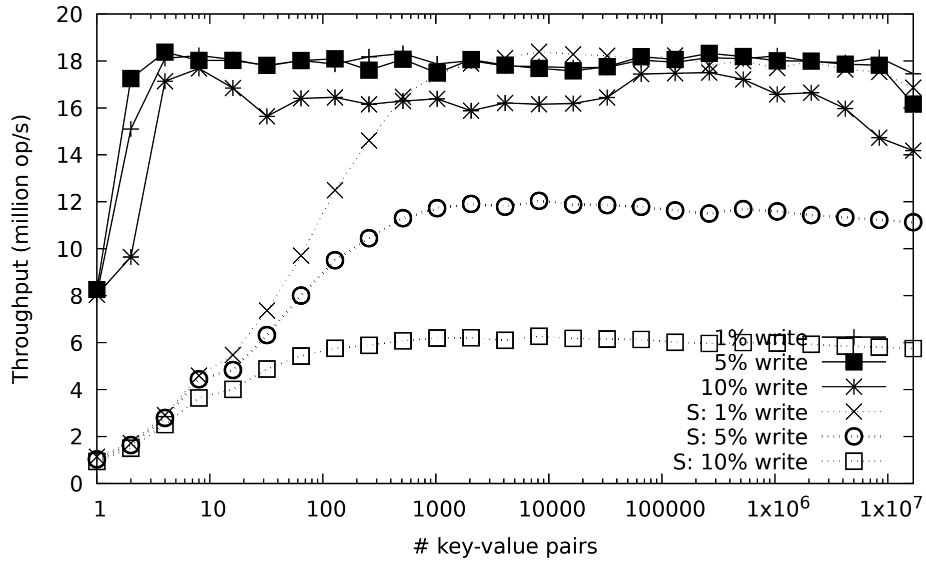


Figure 14: Memcached throughput with varying table size. Uniform access distribution. S: stock memcached.

The server and client run on separate machines, connected by 100Gbps Mellanox-5 Ethernet interfaces via a 100Gbps switch. Both client and server machines are 28-core, two-socket systems with Intel *Sandy Bridge* CPUs and 256 GB of RAM. The machines run Ubuntu Linux with kernel version 5.15.0. Unless otherwise noted, we structure the experiments as follows: start a fresh `memcached` instance. Populate the table with the indicated number of key-value pairs, then run measurements with 1% writes, 5% writes, and 10% writes. After this, we start over with a new, empty `memcached` instance. Each data point represents a single experiment, each set to last 20 seconds. For each, unless otherwise noted, we choose `memcached` and `memtier` parameters to maximize throughput. By default, this means 28 `memcached` threads pinned to hardware threads 0–27. Running with 56 hardware threads did not yield any further performance improvement.

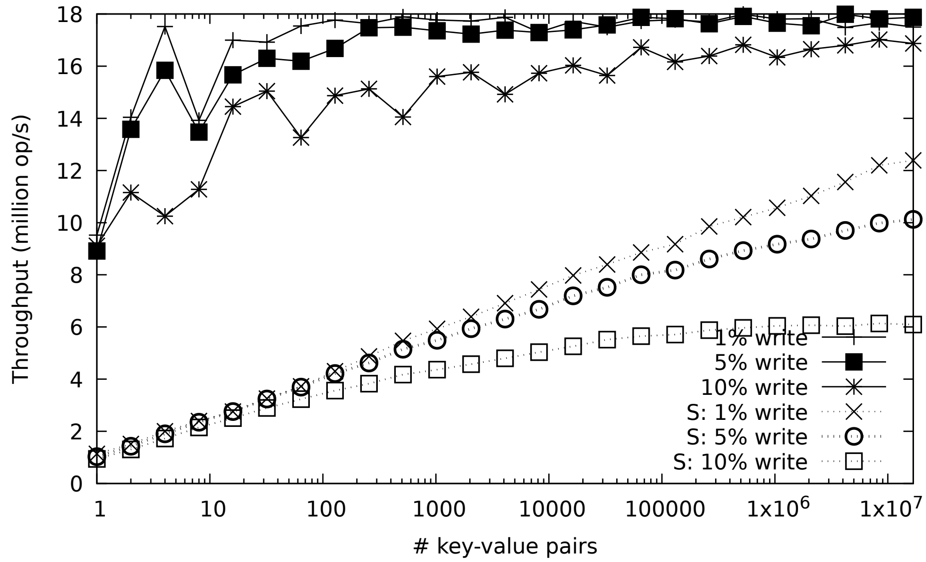


Figure 15: Memcached throughput with varying table size. Zipfian access distribution.

On the memtier side, we configure 28 threads, with four clients per thread, and pipelining set to 48.

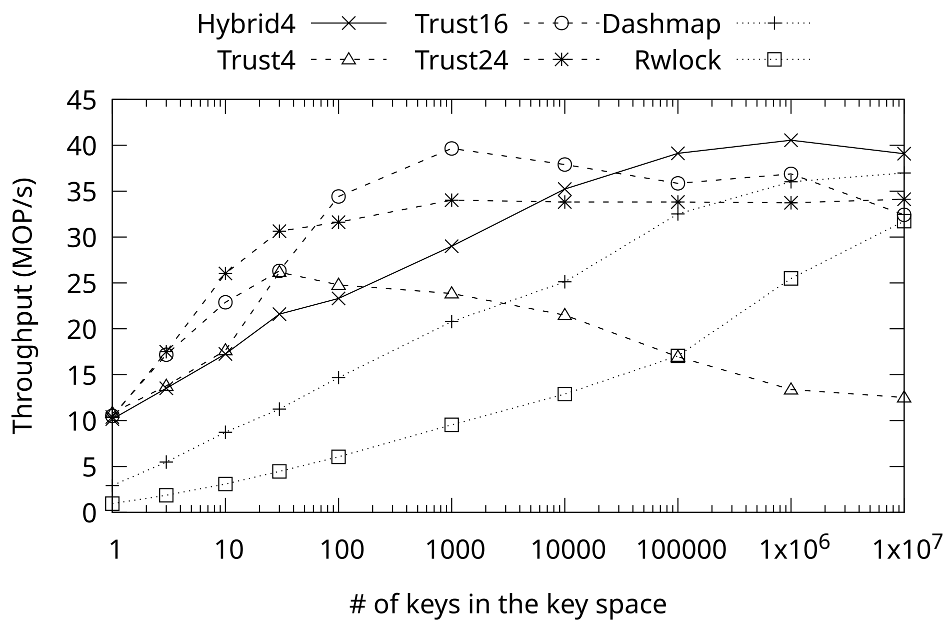
Figures Figure 14–Figure 15 illustrates the throughput of `memcached` as we vary the number of keys in the table. While the absolute numbers are significantly lower than in the microbenchmarks and the key-value store, the overall picture from `memcached` corresponds well with previous experiments.

Using *Trust*<*T*> results in performance improvements of more than 5× when accessing popular objects, whether this popularity is due to a uniform access distribution across a smaller number of keys, or a Zipfian distribution over millions of of key-value pairs. When all items are accessed infrequently, locking suffers very little contention, and has the advantage of better distributing the work across cores. Here, this results in performance competitive with delegation, at least for read-heavy workloads.

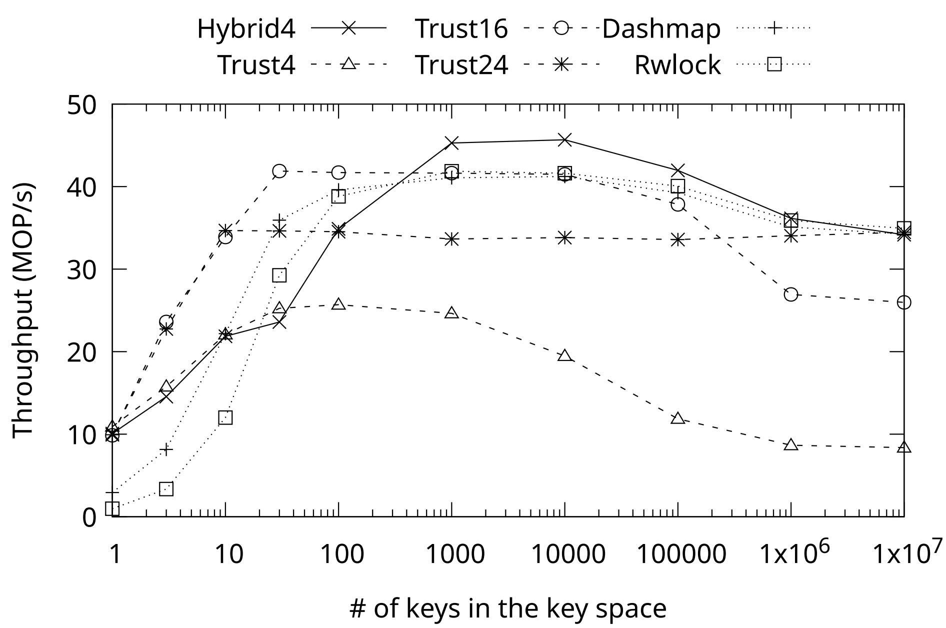
The stock version is heavily affected by writes, due to the extra work required for these operations. This includes memory allocation, LRU updates as well as table writes, all of which involve synchronization in a lock-based design. With *Trust*<*T*>, all such operations are local to the shard/trustee, and do not require synchronization. With 5% of writes, stock memcached loses ≈40% of its performance, while the *Trust*<*T*> version sees only a minor performance penalty, resulting in delegation outperforming locking in this setting for the entire range of table sizes. While not shown, this trend continues with even more writes.

3.7 Hybrid Delegation

Figure 10 shows that delegation's performance advantage depends on the workload and lock contention levels. This raises the question of whether it is possible to get the best of both worlds. In his PhD thesis, Dr. Chen (46) introduced HybridLock, a synchronization mechanism that dynamically switches between traditional locking and delegation based on workload conditions. When a HybridLock is created, it assumes low contention and starts with locking as default. If, during the runtime, it detects that the lock is highly contented, it switches to using delegation manage access to the associated property. By efficiently detecting contention and switching between the two modes, HybridLock can leverage delegation to achieve high performance in a highly contented system while keeping the overhead during low contention workload to a minimum. Figure 16 is taken from Dr. Chen's thesis and shows that HybridLock can maintain high performance across contention levels.



(a) Zipfian



(b) Uniform random

Figure 16: The throughput of key-value stores using compared systems with different key space size (5% write)

3.8 Conclusion

Synchronizing access to shared data is essential for ensuring correct behavior in multi-threaded applications. This paper introduces *Trust<T>*, a delegation based synchronization mechanism for safe, highly performance concurrent access to shared mutable state in Rust. *Trust<T>* provides evidence that delegation can be a competitive alternative to locking in real systems. The programming model integrates cleanly in Rust, making delegation an accessible option for developers. This work lays the groundwork for additional language and runtime support to unlock the performance and scalability benefits of delegation-based designs.

CHAPTER 4

GOSSAMER

Delegation has a programming model that works very well for distributed computing. Like distributed systems, delegation does not have to rely on shared memory for operations. This makes it possible to write an application that is designed for a single machine delegated system and then adapt it for a distributed delegation system without significant changes to application code.

We introduce *Gossamer*, a distributed programming model that leverages delegation to efficiently distribute computation across machines. *Gossamer* uses Remote Direct Memory Access (RDMA) for inter-node communication, offering low latency, high throughput, and direct remote memory access without CPU involvement. This enables a unified programming abstraction that functions seamlessly on both single and multi-machine setups.

With *Gossamer*, developers can write distributed applications as if they were running on a single system. Its API allows delegation of Rust closures for remote execution, and lightweight user-space threads (*fibers*) provide efficient concurrency. Experimental results show that Gossamer achieves up to 3x higher throughput scaling than eRPC in microbenchmarks and matches the performance of the Graph500 reference implementation of single-source shortest path (SSSP) using MPI.

Since we hit the upper limit to how fast a single core can perform, programmers have diverted their attention to making multi-threaded applications. This gives rise to the problem of synchronizing many threads as they access the same data for correctness and making it efficient to not lose too much performance when compared to single threaded applications. Today locks dominate this paradigm to the extent that there are hardware mechanisms in place to make the use of locking easier and more efficient. Some examples include atomic exchange or atomic increment instructions. However, a delegation based programming model can offer better performance than a locking based one (21; 1; 38).

This paper utilizes the approach presented in *Trust<T>* (1), which in turn extends the work presented in *ffwd* (21), to build a rack-wide programming framework. *ffwd* and *Trust<T>* are based on delegation where a few threads (servers) act on behalf of many threads (clients). Its aim is to provide single threaded data structure performance in a multi-threaded setting. *Gossamer* takes it one step further and aims to do the same in multi-machine setting. *Gossamer* provides a very similar API to *Trust<T>* enabling programmers to design applications like a normal delegation application with very few changes. These applications give the illusion of running on a single machine even though they may be distributed across many nodes. Most of the distribution of work is handled behind the scenes, but programmers have the option to customize where the worker threads should run and where data should be held in the rack if they so choose.

One inherent problem that restricts performance in distributed settings is the higher latency caused by network traversal. *Gossamer* overcomes that by using RDMA. RDMA provides sub-micro second one way latency that is comparable to that of permanent storage on single machine. While RDMA solves the problem with high latency, it does not scale well with increasing number of connections per machine. To solve this *Gossamer* establishes only one connection per hardware thread for each remote machine in the rack and uses lightweight user space threads called fibers to utilize the available throughput to full extent.

The primary contributions of this paper are as follows:

4.1 Background on RDMA

RDMA is a networking approach that allows one machine to directly access a remote machine's memory. It uses the user-level zero-copy transfers to minimize the involvement of remote machine's operating system and CPU as opposed to traditional TCP/IP stack that involve both on each machine heavily. There are many implementations of this concept (28; 29; 30; 31), most popular of which are InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (internet Wide Area RDMA Protocol). The results presented in this paper are obtained by using RoCE.

4.1.1 RDMA API

RDMA hosts communicate using queue pairs (QPs) that consist of a send queue and a receive queue, and are maintained by the NIC. Applications post operations to these QPs by using functions called *verbs*. Each queue can be associated with a completion queue, that can be polled to learn the status of posted operations. The completion queue can be shared between both parts of the QP. For remote access the remote machine first needs to register a memory region with the NIC. The NIC driver pins this region in physical memory. The address and a key related to this region then needs to be exchanged between the machines out of band (i.e. without using RDMA). After this exchange, the remote memory can be accessed without involving either of remote operating system or CPU. This is called *RDMA Memory Semantics*, and uses *verbs READ* and *WRITE*.

RDMA also provides *Messaging Semantics* that use *verbs SEND* and *RECV*. In this case receiver has to post a *RECV verb* before the sender can send the data. In this regard it is similar to an unbuffered socket's implementation. Just like *Memory Semantics*, *Messaging Semantics* also bypass the remote kernel but unlike *Memory Semantics*, it has to involve remote CPU to post a *RECV*. These *verbs* also have slightly lower latency than *READ* and *WRITE* (47; 48).

4.1.2 Transport types

RDMA transports are either connected or unconnected (also called datagram), and either reliable or unreliable. Connected transport require a one-to-one connection between two QPs. If an application wants to communicate with N machines, it will need to create N QPs. With unconnected transport one QP can communicate with many QPs. For reliable transport NIC uses acknowledgments to guarantee in-order delivery and return an error code on failure, while unreliable transport does not provide any such guarantee. InfiniBand and RoCE use lossless link layer, so even in case of unreliable transports, losses are pretty rare and happen because of bit error or link failure. In case of connected transport a failure will break the connection. Not all transports provide all of the verbs. Table I gives an overview of transport types and the verbs they support. Current implementations of RDMA only provide reliable connected (RC), unreliable connected (UC) and unreliable datagram.

The meaning of a success status while polling a completion queue depends on the type of transport the related send/receive queues are using. A work completion is generated for all

Verb	RC	UC	UD
SEND/RECV	✓	✓	✓
WRITE	✓	✓	✗
READ	✓	✗	✗

TABLE I: Operations supported by each transport type.

receive requests and added to the completion queue. For send requests the work completion is only generated if the signaled bit was set to true in the work request or the send request ended with an error. In case of reliable transport, a success for send requests indicates that the data has been written to the remote machine's memory. For unreliable transport however, a success only means that the local buffers can be reused safely, with no indication of the remote memory's status. A successful work completion also indicates that any unsignaled work posted earlier has finished successfully (49). *Gossamer* uses RC QPs to ensure any data loss in the network is caught. We also use the cumulative nature of work completion successes to our advantage by not using signaled send for every RDMA write and only having a signaled send after 1000 unsignaled sends. This allows us to cut down on time spent while polling the completion queue.

4.2 Gossamer Design

Gossamer allows developers to write rack-wide applications just like single machine applications. Programmers will use the delegation framework to write applications that are distributed by gossamer with very few changes to the code. Figure 17 shows a high level view of what a rack with multiple *Gossamer* applications looks like. Developers use the controller machine

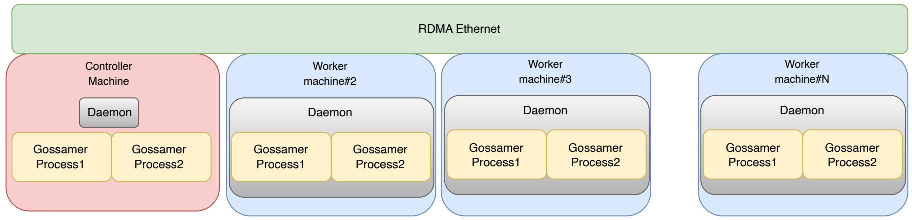


Figure 17: High-level design of Gossamer

to interact with the rack and launch applications. Here controller machine is just a normal machine running *Gossamer* processes like any other machine in the rack, the only significance is that this is the machine that users log into for terminal access. Each machine in the rack has a daemon running that is always listening for instructions from the controller machine. The daemon is responsible for determining the topology of the rack and managing the applications. A machine can have multiple applications running on it at the same time. Rack applications that are running simultaneously are isolated from each other like multiple processes on a single machine. The rest of this section describes some design details that are particularly interesting.

4.2.1 Gossamer Daemon

As shown in Figure 17, each machine has *Gossamer* daemon running. These daemons connect with each other to discover the rack size and store metadata like gossamer-id, machine-ip etc. The daemon is responsible for managing any *Gossamer* processes running on the rack at any time. To launch a *Gossamer* application, users need to run the application binary on the controller machine only. The application will contact the local daemon and tell it to start the process specifying how many system threads the whole *Gossamer* process should use. This

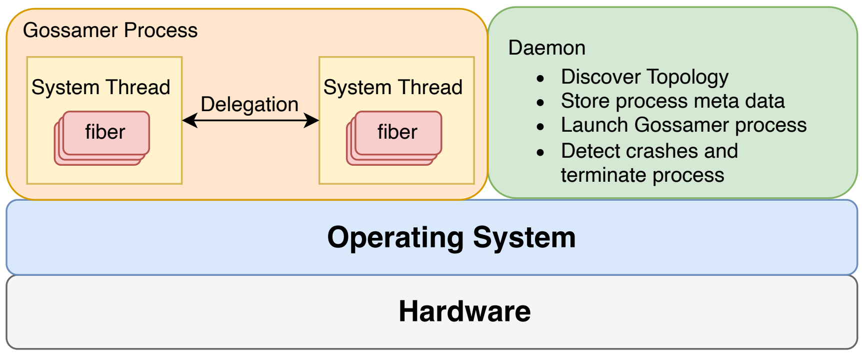


Figure 18: A Gossamer process on one machine

daemon will contact the rest of the rack and tell the daemons there to launch the process as children. The daemon is also responsible for detecting crashes on any local instance of a *Gossamer* process and terminating it across the whole rack. This means that *Gossamer* processes are fate sharing in the same way single machine processes crash if a single thread crashes.

4.2.2 Gossamer Process

To launch a *Gossamer* process, the user logs into the controller machine and starts the application there. Since parts of *Gossamer* rely on similar memory layout (as discussed in 4.2.7), each machine in the rack should be populated with the same binary ahead of time. The application needs to have its *main* function call a helper function from the *Gossamer* library and pass a function that would act as the real *main* function along with how many threads the application needs to use across the whole rack. This function will be referred to as *gsm_main* for the rest of the paper. The process consists of many fibers per system thread for concurrent operation as shown in Figure 18. Fibers are lightweight, user space threads that can spawn on any system thread that is part of the *Gossamer* process, and on any machine in the rack. Fibers that need to access shared data make delegation requests consisting of closures that modify the data as needed. From the point of view of the operating system, a single instance of *Gossamer* process behaves like any other normal process. The main difference is that instead of using system calls like *getpid()*, a *Gossamer* process contacts the daemon to get any metadata needed.

Table II shows a minimal *Trust<T>* example and how the same program looks like when using *Gossamer*.

4.2.3 Trustee, Trust and Property

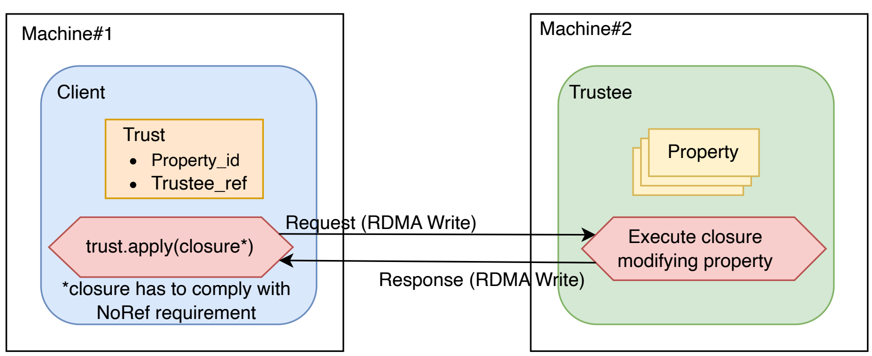


Figure 19: Trustee is a worker and clients can delegate work using a trust that holds information about property and trustee

```
fn main() {
    let pool = Threadpool::new(4);
    let property = HashMap::new();
    let trust = pool.entrust(property);
    pool.spawn(|| {
        trust.apply(|property| property.insert(‘‘Greeting’’,‘‘Hello’’));
    });
}
```

(a) A small example of delegation code with *Trust<T>*

```
fn main() {
    gossamer::main(gsm_main, 4);
}

fn gsm_main() {
    let property = HashMap::new();
    let trust = gossamer::entrust(property);
    gossamer::spawn(|| {
        trust.apply(|property| property.insert(‘‘Greeting’’,‘‘Hello’’));
    });
}
```

(b) A small example of delegation code with *Gossamer*

TABLE II: *Trust<T>* vs *Gossamer* code example

```
let property = HashMap::new();
let trust = gossamer::entrust(property);
gossamer::spawn(|| {
    trust.apply(|property| property.insert(‘‘First Greeting’’,‘‘Hello’’));
    trust.apply_then(|property| property.insert(‘‘Second Greeting’’,‘‘Hi’’),
        |_| println!(‘‘added second greeting’’));
});
```

TABLE III: A small example of delegation code

Gossamer builds on the Trust/Trustee concept from (1) (chapter 3 in this dissertation), which we introduce briefly here. A *Trustee* is a worker fiber that processes delegation requests and sends back the response. The application can entrust some data to a trustee if the delegated work needs access to it and receive a *Trust* that holds relevant metadata. This entrusted data is referred to as *Property*. The Trust can be used to delegate any work that needs access to this property. Figure 19 shows the relationship between Trust, Trustee and Property. A Trust can be cloned and shared with other fibers so that they can also start delegating work that needs access to same property. Multiple properties can be entrusted to a single trustee at the same time. Trustees can act as remote or local trustees based on where the fiber trying to access the property lives in the rack.

4.2.4 Delegation Workflow

Gossamer provides both blocking and nonblocking/asynchronous delegation operations. In case of blocking operations, the delegating fiber will wait for the response from the trustee before continuing. For nonblocking delegation, the delegating fiber will instead provide a callback closure to be executed upon completion of the request, and continue without waiting. First we will describe the workflow for blocking delegation requests and then specify how that differs from the nonblocking delegation. Each fiber issues a request that is put in a pending queue before voluntarily yielding the runtime. Each thread has a fiber that periodically checks for any incoming responses and sends any requests in the pending queue. As there will be many fibers running on each thread this will allow the polling fiber to send multiple requests as a larger batch, increasing the throughput of the system. The requests originating from a client that are destined for the same trustee are batched in a *RequestSlot* (discussed in detail in 4.2.7) that can hold a variable number of requests depending on their size. On the trustee side, one of the fibers is dedicated to polling for any incoming requests. Since any incoming requests in a RequestSlot will be contiguous in memory, it can complete all of them and then send all of the responses in a single batch as well. Similar to requests, responses are batch using a *ResponseSlot*. The fiber that polls for responses on the requesting thread will then process the responses and add the corresponding fibers back in to the ready to run queue. The main difference for async delegation is that instead of yielding after enqueuing the request, it just keeps going and enqueues any subsequent requests as well. This fiber will yield the runtime when the pending queue is full, after which the response polling fiber can run and send all of the requests to the respective trustees. Once it receives the responses, it can execute any callback closures the user might have provided. This means that there is no benefit to having multiple fibers per thread that issue requests as that is not needed for batching and the fibers will be yielding far less frequently. Table III shows a small example program that uses both types of delegation.

4.2.5 NoRef

Gossamer leverages the Rust type system to prevent the sending of references over delegation channels. One of the first challenges that arise when extending delegation to multiple machines is the fact that any pointers or references captured by a delegated closure will not be valid on a remote machine. This reference or pointer will result in undefined behavior on any machine other than where it originated. To prevent this, *Gossamer* uses a combination of rust features named *auto_traits* and *negative_impls*. In rust, traits define the behavior of data types. They inform the compiler what functionality a type has and can be used to restrict which data types can be passed to a generic function. Auto_traits is a feature that automatically implements a trait for every data type, be it an existing type or user defined. A negative implementation is used to exclude a data type from the auto implementation. *Gossamer* defines an auto trait called *NoRef*. Then all the types that contain a reference or raw pointer are given negative implementation. The function *apply*, that is used to send delegation messages as described in previous section, requires all the closures to comply with the NoRef trait as shown in Figure 19. This however limits severely what type of data can be captured by any delegated closure as many frequently used types like strings and vectors contain pointers internally. To get around that *Gossamer* provides a way to send any data type that can be serialized along with the closure as an extra parameter to the delegated closure. If the programmer makes a mistake and tries to use a closure that has captured a reference for delegation, a custom error message at compile time will inform them what the issue is and direct them to use the serialization method instead.

4.2.6 Request size and TCP fallback

As described in 4.2.4, the requests going from a client thread to the same trustee will be batched and on trustee side the requests coming from a single thread will all be contiguous in memory. This is achieved by the trustee having a pre-allocated space in memory (called requestSlot) for incoming requests from each client thread. This limits how many requests a client thread can send in a single batch depending on how much data is being sent with each request. As an example if each request captures 32 bytes of captured data, the total request size is 40 bytes. Next section goes into detail about the format of a *Gossamer* delegation request. Assuming 1 KB requestSlot, one batch can have at most 25 requests. While this allows us to optimize for small requests it also places a hard limit on how much data a request can capture i.e. the size of the requestSlot. To work around this limit and support larger requests, *Gossamer* uses TCP to send any captured data larger than that separately, while the request header goes in the requestSlot as normal. This degrades the performance severely as TCP is much slower than RDMA. We use TCP here instead of RDMA because, as mentioned in 4.1.1, RDMA needs the memory used to be pre-registered with the NIC, placing an upper limit on how much data can be sent using RDMA at once, i.e. the size of pre-registered buffer. Since there would be an upper limit anyway and larger requests are not the focus for *Gossamer*, we decided to opt for the simpler design in place at the moment.

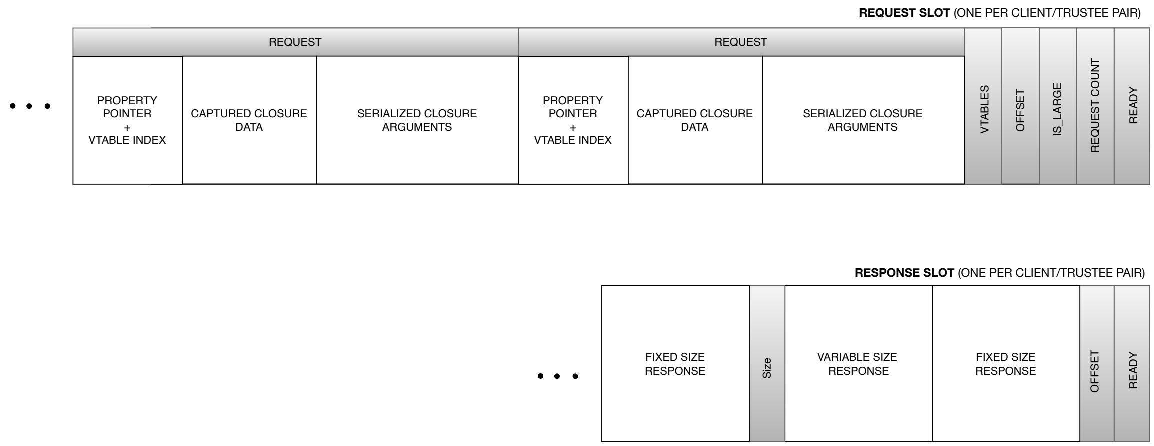


Figure 20: Request and Response slot layout

4.2.7 Request/Response Slot Format

As discussed in the previous section, the number of requests sent in batch depends on the size of the requests plus the size of any headers sent with the request. Section 3.4.3 describes the format of request and response slots used for single machine delegation channel. Here we describe further optimizations made to request slots to minimize per request metadata to reduce wasted network bandwidth. The two-slot optimization of single machine delegation channel is also disabled for remote delegation, as the cost of doing two separate RDMA write operations outweighs any benefit gained from it.

The necessary parts to process a *Gossamer* request are as follows:

- The closure. Closures in rust are fat pointers consisting of a vtable that points to where the code is in the text section along with the size of captured data, and a data pointer that points to the captured data.
- The pointer to where the property is located on trustee.
- Length of serialized data.
- The captured data.
- Any serialized data.

While we can't avoid sending the captured data, serialized data and the property pointer, *Gossamer* employs some strategies to minimize the information that needs to be sent inside the request slot. As the data pointer within the closure is not going to be valid on a remote machine, there is no need to include that in the request. Since the same binary is running on all the machines in the system, they all have access to any information known at compile time. This includes all of the information in the vtable, provided the vtable pointer. Here, we make the observation that for the vast majority of the runtime of an application, most of the requests in the request slot will be about a single closure or a few closures at most, meaning we can have a small lookup table for a few vtable pointers in the request slot instead of including one with every request. Combining that with the start point of data received, the trustee can reconstitute the closure locally, removing the need to send the closure in the request slot. If there are indeed more closures in the request slot than will fit in the lookup table, any extras can be sent in the request slot. This will reduce the data being sent in the common case with minimal overhead, but will be no more expensive in the special case. Similarly, if the serialized data has a size known at compile time the trustee already has access to it, so the only time it needs to be sent with the request is if it is not known at compile time. One thing to note here is that we rely on same vtable pointer to be the valid on all of the machines which is not the case normally due to linux address space layout randomization (ASLR), making disabling it a necessity for *Gossamer* to function.

Similarly, the responses for all of the requests in a request slot are batched in a response slot and written to the client's memory with a single RDMA write. Unlike a request however, the size of a response is not always known when submitting a request. If the response size cannot be determined statically, it is preceded by 8 bytes that contain its size. While a known response size can be used to limit the number of requests in a request slot just like the size of requests, unknown size responses can lead to scenarios where not all of the responses for a request slot fit in a single response slot. In such cases the trustee will send anything that doesn't fit in the response slot to the client using TCP in a similar fashion as discussed in 4.2.6. In addition, if the response size is zero and statically known, then nothing is sent in the slot for that particular response. Figure 20 shows the format for both the request and response slots.

In addition to the request/response, the slots also contain some metadata. They both contain a ready bit and an offset, with the request slot containing a few more things. The ready bit is there to let the other side know of a new request or response. It has to be at the end to ensure that when a client or trustee sees the ready bit the rest of the slot has already been written. This is necessary because RDMA writes data sequentially and having the start of a slot been written to, does not mean all of it is available to read. The offset has to do with another optimization for the size of RDMA writes. Depending on the size of requests and responses, a slot can be sent to the remote machine before it is 100% full, in which case writing the full slot to remote machine's memory wastes bandwidth and lowers performance. Since the ready bit needs to be at the end of the slot and in a known place, requests and responses are aligned to the end of the slot leaving the start of it empty. The offset informs the remote side where the actual data starts in the slot. The request slot also contains the request count and the vtable cache as discussed earlier. The last thing in request slot metadata is a bit that indicates if there is a single large request in the slot or multiple small ones. As discussed in 4.2.6, if the request will not fit in the slot it is sent using TCP and only the relevant pointers go in the slot. The decision to only send a single request if it is large is to simplify the design since large requests are not the focus of this work.

4.2.8 Trustee and Client memory layout

The clients and trustees on a single machine share memory, so a client can prepare the whole request slot in a pre-designated place and then flip the ready bit at the end to signal the trustee that it is ready. The trustee continuously polls this piece of memory to determine the arrival of new requests. This leads to $\mathsf{num\_threads}^2 \times \mathsf{sizeof}(\mathsf{reques\_slot})$ bytes of memory being reserved for request slots. Same for response slots as well. We can think of that as a *square* in memory with $\mathsf{num\_threads}$ side length. When we move on to a distributed setting,

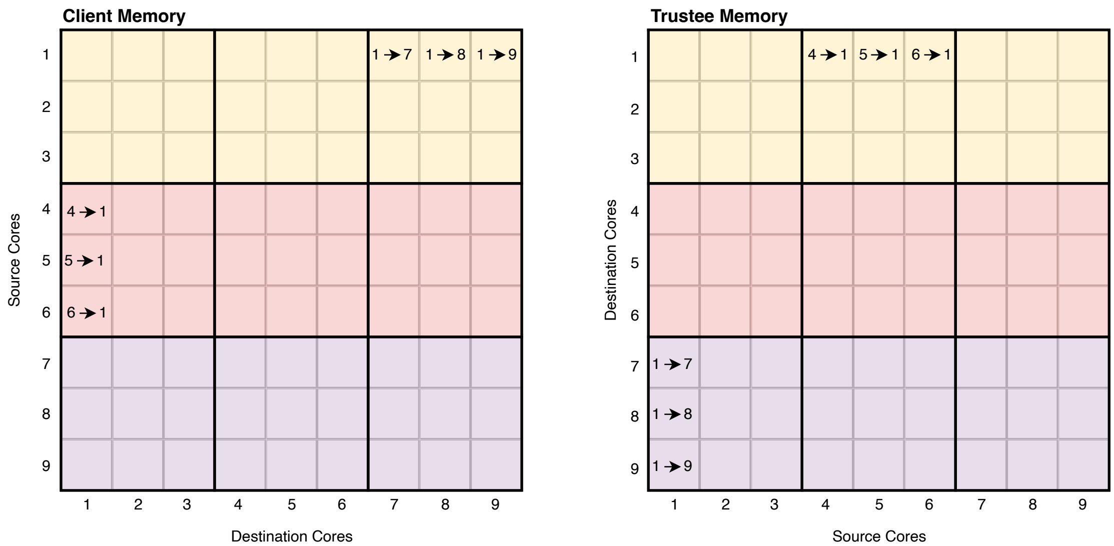


Figure 21: Memory layout for Request Slots in client and trustee memory for a rack of three machines where each machine has three system threads.

we can extend this *square* to have $\mathtt{num\_threads} \times \mathtt{num\_machines}$ side length. The whole of the bigger *square* does not need to be allocated on each machine as that would waste memory. Instead, parts of it are allocated on each machine. Figure 21 shows an example layout for a system with three machines that have three system threads each for a total of nine threads. In the figure, each cell represents a request slot going from a source thread to a destination thread as labeled and horizontal rows represent memory that is physically contiguous while different colors divide the *square* based on which machine the memory is allocated on. Here, unlike in a single machine setting, there are two *squares*. Clients on one machine need to prepare the requests in local memory before writing the request slot to the trustee's memory on another machine. Theoretically, RDMA allows the clients to write the requests in the trustee's memory as they come and flip the ready bit once the slot is full similar to single machine channel, however this would result in many small RDMA write operations, reducing the effectiveness of batching and sinking the performance.

From the perspective of the whole rack as a single system, each row of request slots in the client memory *square* maps to a column in the trustee memory *square*. For example a request slot going from thread 1 (thread 1, machine 1) to thread 7 (thread 1 machine 3) is prepared in memory represented by the cell in row one and column seven in client memory *square* and then written to the cell in row seven and column one in trustee memory *square* and so on. Figure 21 also shows a few additional examples. There is a similar system in place for response slots as well.

4.2.9 Scaling impact of increasing number of Queue Pairs

As discussed in 4.1.1, rdma communications happen via a queue pair (QP). Theoretically each machine pair only needs to have a single QP resulting in $\mathit{num\_machines}^2$ QPs. However, this means sharing a QP among many threads. The QPs use a spin lock internally to make sure QPs can be shared safely, but that results in threads having to wait for each other before they can send requests to a trustee. A better but naive design might be to have a QP for every pair of system threads in the whole rack resulting in QPs in the order of $(\mathit{num\_machines} \times \mathit{num\_threads\_per\_machine})^2$. This also results in lost performance due to NIC cache limitation. The NIC caches any recently used QPs, along with page mappings for any recently used pinned pages in its SRAM. Increasing the number of QPs results in NIC running out of SRAM, which in turn results in cache misses. We use a middle of the road

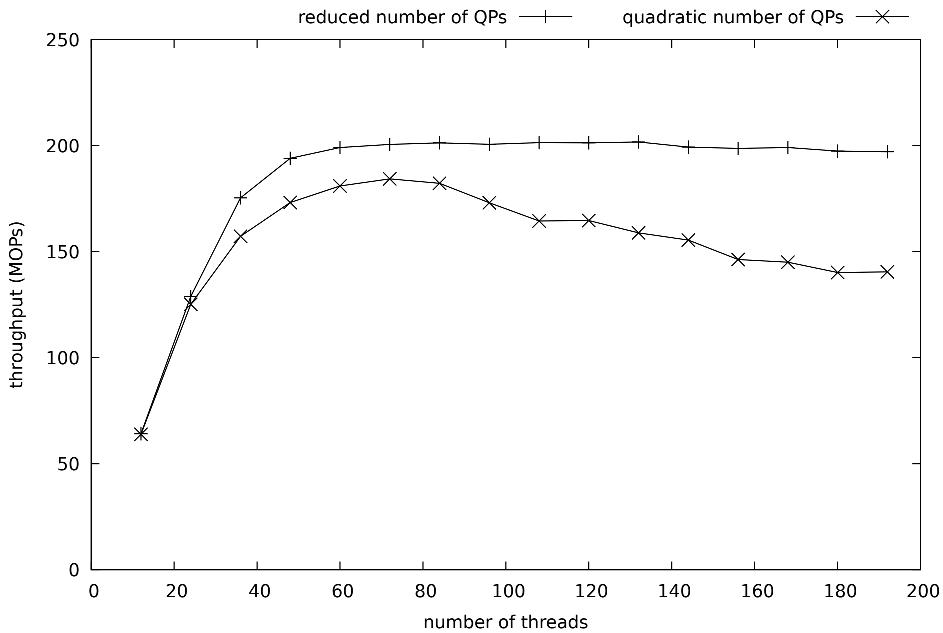


Figure 22: Throughput of RDMA writes in MOPs for quadratic and reduced number of QPs. Each write is 256 Bytes

system where instead of each thread making a separate QP for every other thread, it makes a QP for each machine in the rack. This reduces the total number of QPs in the system to $num\_machines^2 \times num\_threads\_per\_machine$. Figure 22 shows an experiment that demonstrates the performance gained by reducing the number of QPs in the system. This experiment was performed with 6 machines, all connected to switch with 100Gb/s ethernet links. Each thread writes 256 bytes in a randomly chosen remote thread's memory twenty million times. As the number of threads increases, thereby increasing the number of QPs, the system with reduced number of QPs maintains peak throughput whereas the system with quadratic number of QPs starts to fall off, resulting in a 25% decrease at the extreme end.

4.3 Evaluation

In this section we use microbenchmarks to evaluate *Gossamer* on throughput and latency. We also use two application benchmarks to evaluate how *Gossamer* performs in the real world. These experiments were performed on the r6615 cluster from Cloudlab Clemson using 12 machines. Each machine here uses AMD EPYC 9354P CPU and has 32 threads over four NUMA nodes at 3.3 GHz. We do not use hyperthreading for this experiment. Each machine has 100 Gbps Mellanox ConnectX-6 interfaces and connected to a Dell Z9432 switch. These machines are equipped with 188 GB RAM each and are running Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-160-generic x86_64).

4.3.1 Microbenchmark: Throughput

We use an echo application to evaluate the throughput scaling of *Gossamer* as the number of machines in the rack increases. For this experiment, each trustee/server sends the same data it received as a response. Each thread acts as both client and server. Clients choose a server at random form a uniform distribution and send a request containing 32 bytes of data. Each client sends ten million asynchronous echo requests. We compare against eRPC (32) and an MPI library used and developed by graph500 reference benchmark (50).

Figure 23 shows the results of our experiment. Here we use 16 threads per machine as the eRPC benchmark only supports up to two NUMA nodes. Each successive data point has one more machine than the previous one, resulting in an increment of 16 threads. eRPC starts with 2.9 MOPs per thread and maintains that as more threads are added. This is in line with their reported performance in a github issue (51) for two machines connected with a 100 Gbps

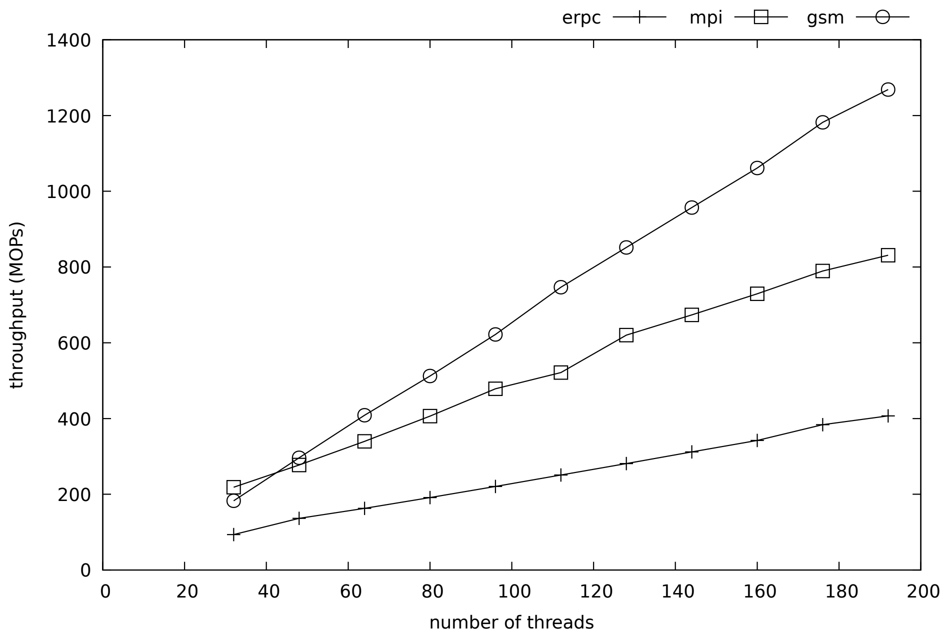


Figure 23: Throughput vs number of threads for eRPC vs mpi vs gossamer.

connection. *Gossamer* starts with double the throughput and scales 3x better than eRPC. Similarly, though the MPI application starts out performing better than *Gossamer*, it falls behind quickly as the number of machines increases.

Figure 24 shows the performance of all three systems across a range of request sizes. The MPI system performs better than both *Gossamer* and eRPC due to smaller request headers, however as the request size increases, this advantage becomes negligible and the other systems overtake it. All three systems experience a drop in the number of requests served as the request size increases, however *Gossamer* experiences a particularly steep drop as it cross th

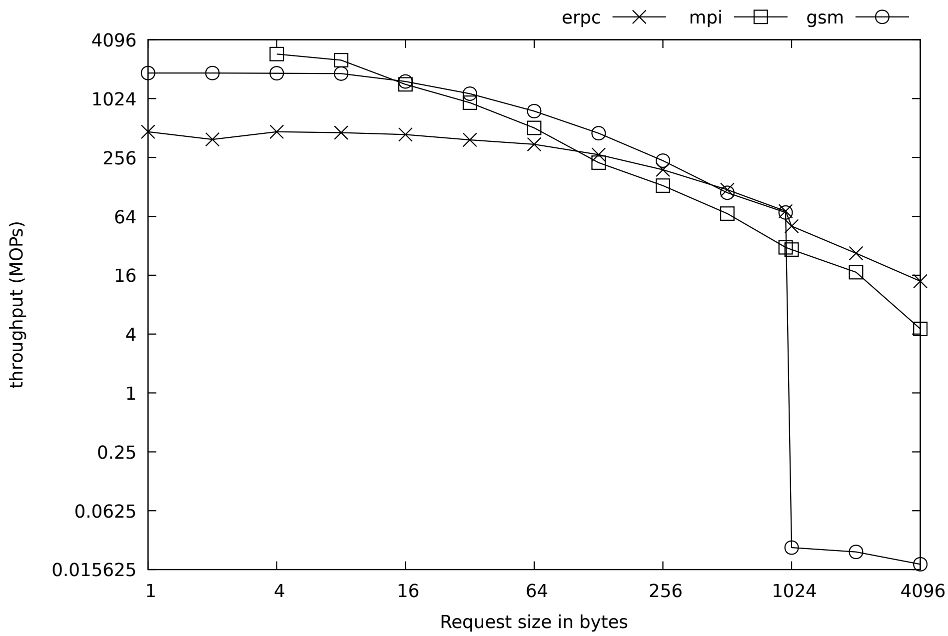


Figure 24: Throughput vs request size for eRPC vs mpi vs gossamer.

1KB threshold. This is explained by the TCP fallback for large requests as discussed in section 4.2.6.

4.3.2 Microbenchmark: Latency

Next we evaluate 99.9th percentile latency under load for eRPC and *Gossamer*. We exclude the mpi application from this experiment as the mpi library we use is highly optimized for throughput at the cost of latency, resulting in it not being competitive. This experiment also uses the echo application with a request size of 32 bytes. As both *Gossamer* and eRPC use batching for asynchronous requests, load here is represented by the number of outstanding requests at any given time.

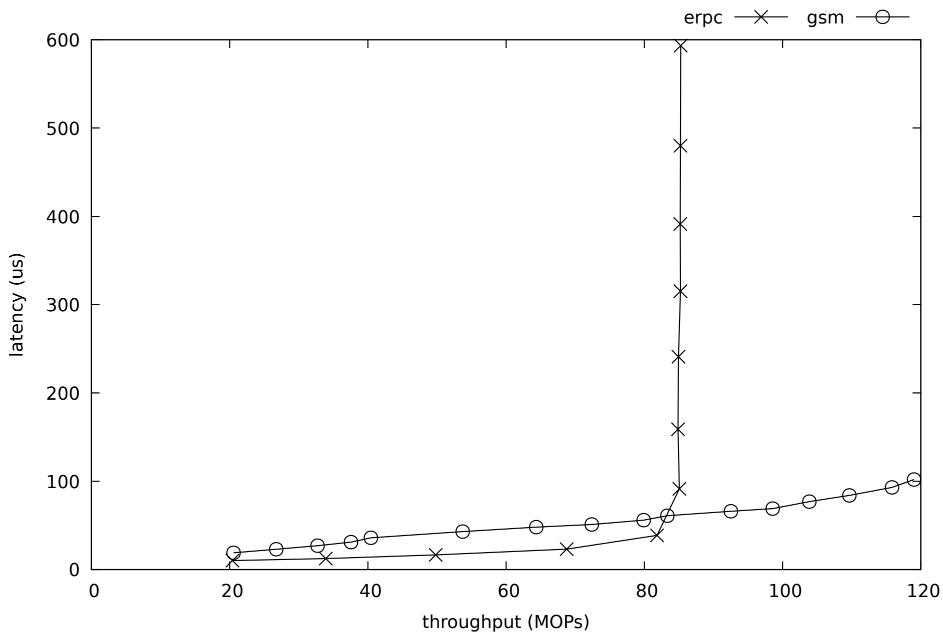


Figure 25: Throughput vs latency under load for Gossamer and eRPC with two machines and 16 threads per machine.

Figure 25 shows the results of this experiment. Each data point here represents a doubling of batch size until 32 and then an increment of 16. Both system have increased throughput and latency as the number of outstanding requests increases, however eRPC does better at smaller loads in terms of both latency and throughput. This is because of the optimizations discussed in 4.2.7, that result in better performance at higher load at the cost of small performance penalty at lower loads, along with the overhead of switching fibers being significant at lower load. eRPC then quickly reaches its peak throughput and any increase in load only increases the latency whereas *Gossamer* is able to achieve much higher throughput while remaining below 100 µs tail latency.

4.3.3 Application: Single Source Shortest Path (SSSP)

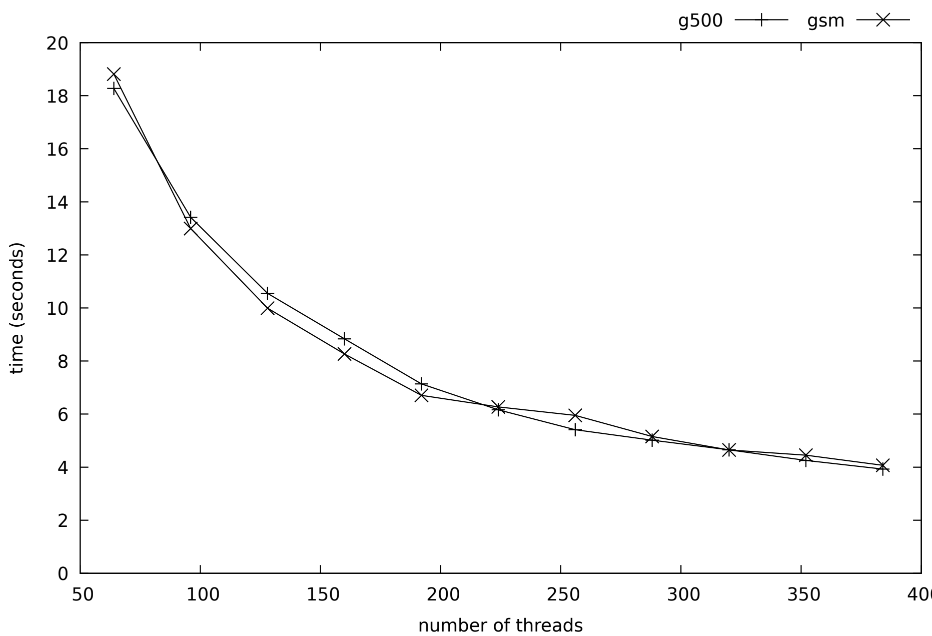


Figure 26: Time taken for SSSP on a graph with 2^{27} nodes and 32 average degree.

To understand the real world performance of *Gossamer*, we implement a distributed version of Dijkstra's algorithm for shortest path to each node in a weighted undirected graph given a source node. We compare this application to the graph500 reference benchmark (50) for supercomputers. Even though the graph500 benchmark uses the delta-stepping algorithm, we observed that both systems traverse roughly equal number of edges, making any algorithmic differences negligible. The graphs for these experiments were generated by graph500's imple-

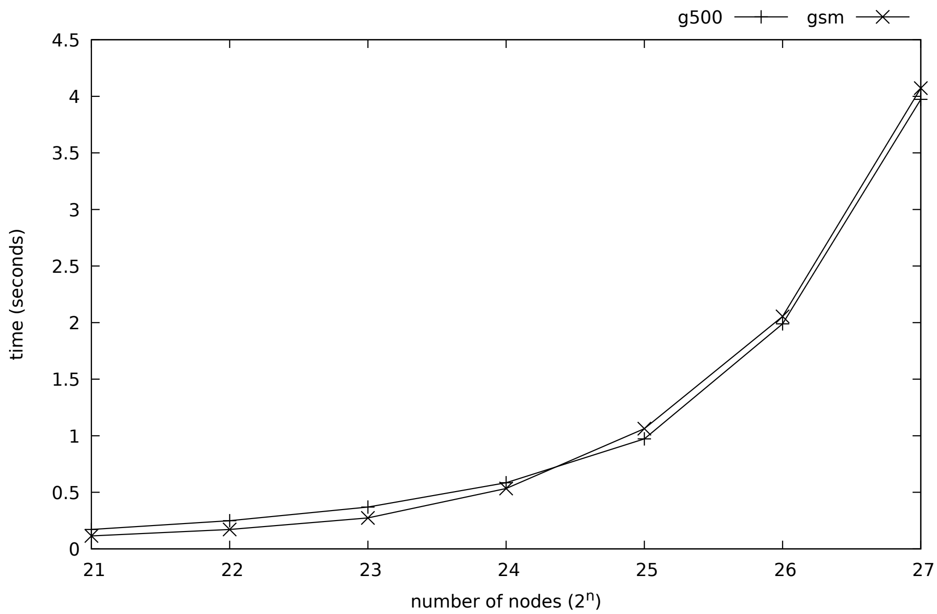


Figure 27: Time taken for SSSP on graphs with 2^n nodes and 32 average degree.

mentation of kronecker generator with an edge factor of 32. For each experiment we ran sssp on eight randomly chosen source nodes and report the average time taken per run.

Figure 26 shows the results of SSSP on a graph with 2^{27} nodes and average degree of 32, as the size of the cluster increases. Each successive data point increments the thread count by 32. Despite graph500 being a highly specialized application designed to benchmark supercomputers, *Gossamer* is able to keep up with it. Similarly, Figure 27 shows *Gossamer* achieving comparable performance to graph500 across a range of graph sizes with 384 threads across 12 machines.

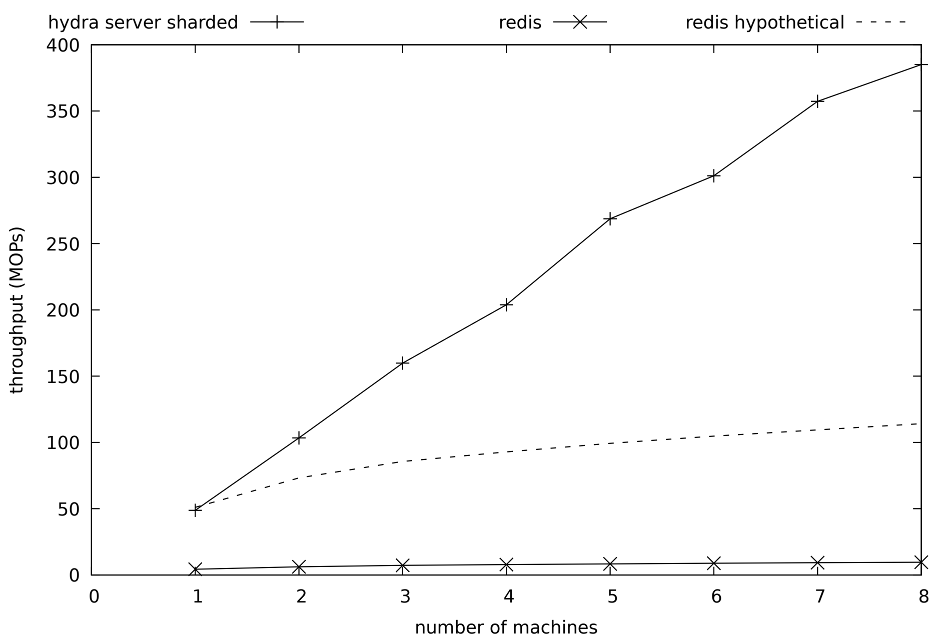


Figure 28: Throughput of key-value stores on up to 8 server clusters and ycsbc workload.

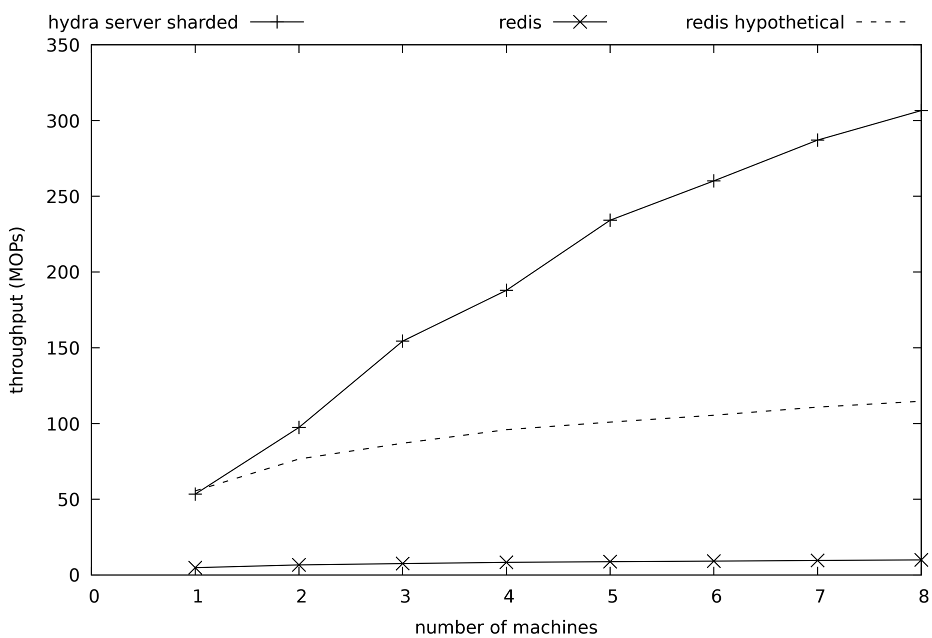


Figure 29: Throughput of key-value stores on up to 8 server clusters and ycsbd workload.

CHAPTER 5

CONCLUSION

APPENDICES

A ACM Reuse Permission

ACM Author Gateway

Author Resources

Home > Author Resources > Author Rights & Responsibilities

ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

Choose

ACM gives authors the opportunity to choose between two levels of rights management for their work. Note that both options obligate ACM to defend the work against improper use by third parties:

- **Exclusive Licensing Agreement:** Authors choosing this option will retain copyright of their work while providing ACM with exclusive publishing rights.
- **Non-exclusive Permission Release:** Authors who wish to retain all rights to their work must choose ACM's author-pays option, which allows for perpetual open access to their work through ACM's digital library. Choosing this option enables authors to display a Creative Commons License on their works.

Post

Otherwise known as "Self-Archiving" or "Posting Rights", all ACM published authors of magazine articles, journal articles, and conference papers retain the right to post the pre-submitted (also known as "pre-prints"), submitted, accepted, and peer-reviewed versions of their work in any and all of the following sites:

- Author's Homepage
- Author's Institutional Repository
- Any Repository legally mandated by the agency or funder funding the research on which the work is based
- Any Non-Commercial Repository or Aggregation that does not duplicate ACM tables of contents. Non-Commercial Repositories are defined as Repositories owned by non-profit organizations that do not charge a fee to access deposited articles and that do not sell advertising or otherwise profit from serving scholarly articles.

For the avoidance of doubt, an example of a site ACM authors may post all versions of their work to, with the exception of the final published "Version of Record", is ArXiv. ACM does request authors, who post to ArXiv or other permitted sites, to also post the published version's Digital Object Identifier (DOI) alongside the pre-published version on these sites, so that easy access may be facilitated to the published "Version of Record" upon publication in the ACM Digital Library.

Examples of sites ACM authors may not post their work to are ResearchGate, Academia.edu, Mendeley, or Sci-Hub, as these sites are all either commercial or in some instances utilize predatory practices that violate copyright, which negatively impacts both ACM and ACM authors.

After an ACM journal submission has been accepted and has entered the production process, ACM makes the Author’s Accepted Manuscript (AAM) available for preview under the ACM “Just Accepted” program until the “Version of Record” is available and assigned to its proper issue. The AAM carries the article’s permanent DOI and can be cited immediately.

Distribute

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library.

- On the Author's own Home Page or
- In the Author's Institutional Repository.

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

B IEEE Reuse Permission

CCC RightsLink

Sign in/Register

Fast Abort-Freedom for Deterministic Transactions

IEEE Requesting permission to reuse content from an IEEE publication

Conference Proceedings:
2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)
Author: Chen Chen
Publisher: IEEE
Date: 27 May 2024

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK CLOSE WINDOW

CITED LITERATURE

1. Ahmad, N., Baenen, B., Chen, C., and Eriksson, J.: Delegation with Trust<T>: A scalable, type- and memory-safe alternative to locks, 2024.

2. Birrell, A. D. and Nelson, B. J.: Implementing remote procedure calls. ACM Trans. Comput. Syst., 2(1):39–59, February 1984.

3. Srinivasan, R.: Rpc: Remote procedure call protocol specification version 2, 1995.

4. Bershad, B., Anderson, T., Lazowska, E., and Levy, H.: Lightweight remote procedure call. In Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP '89, pages 102–113, New York, NY, USA, 1989. ACM.

5. Soumagne, J., Kimpe, D., Zounmevo, J., Chaarawi, M., Koziol, Q., Afsahi, A., and Ross, R.: Mercury: Enabling remote procedure call for high-performance computing. In 2013 IEEE International Conference on Cluster Computing (CLUSTER), pages 1–8, Sep. 2013.

6. Adamson, A. and Williams, N.: Remote procedure call (RPC) security version 3. Technical report, November 2016.

7. Talpey, T. and Callaghan, B.: Remote direct memory access transport for remote procedure call. Technical report, January 2010.

8. Cohen, M., Ponte, T., Rossetto, S., and Rodriguez, N.: Using coroutines for rpc in sensor networks. In 2007 IEEE International Parallel and Distributed Processing Symposium, pages 1–8, March 2007.

9. Brabson, R. F., Majikes, J. J., and Wolf, J. C.: Method and system for improved computer network efficiency in use of remote procedure call applications, March 22 2011. US Patent 7,913,262.

10. Shyam, N., Harmer, C., and Beck, K.: Managing remote procedure calls when a server is unavailable, September 22 2015. US Patent 9,141,449.

11. Merrick, P., Allen, S. O., et al.: Xml remote procedure call (xml-rpc), August 25 2015. US Patent 9,116,762.

12. Soumagne, J., Kimpe, D., Zounmevo, J. A., Chaarawi, M., Koziol, Q., Afsahi, A., and Ross, R.: Mercury: Enabling remote procedure call for high-performance computing. 2013 IEEE International Conference on Cluster Computing (CLUSTER), pages 1–8, 2013.

13. Fatourou, P. and Kallimanis, N. D.: Revisiting the combining synchronization technique. In ACM SIGPLAN Notices, volume 47, pages 257–266. ACM, 2012.

14. Hendler, D., Incze, I., Shavit, N., and Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures, pages 355–364. ACM, 2010.

15. Dice, D., Marathe, V. J., and Shavit, N.: Flat-combining numa locks. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pages 65–74. ACM, 2011.

16. Fatourou, P. and Kallimanis, N. D.: A highly-efficient wait-free universal construction. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pages 325–334. ACM, 2011.

17. Oyama, Y., Taura, K., and Yonezawa, A.: Executing parallel programs with synchronization bottlenecks efficiently. In Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, volume 16. Citeseer, 1999.

18. Yew, P.-C., Tzeng, N.-F., et al.: Distributing hot-spot addressing in large-scale multiprocessors. IEEE Transactions on Computers, 100:388–395, 1987.

19. Shalev, O. and Shavit, N.: Predictive log-synchronization. In ACM SIGOPS Operating Systems Review, volume 40, pages 305–315. ACM, 2006.

20. Gupta, V., Dwivedi, K. K., Kothari, Y., Pan, Y., Zhou, D., and Kashyap, S.: Ship your critical section, not your data: Enabling transparent delegation with TCLOCKS. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 1–16, Boston, MA, July 2023. USENIX Association.

21. Roghanchi, S., Eriksson, J., and Basu, N.: Ffwd: Delegation is (much) faster than you think. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 342–358, New York, NY, USA, 2017. ACM.

22. Brightwell, R., Riesen, R., and Underwood, K. D.: Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. The International Journal of High Performance Computing Applications, 19(2):103–117, 2005.

23. Doolan, D. C., Tabirca, S., and Yang, L. T.: Mmpi a message passing interface for the mobile environment. In Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia, MoMM '08, pages 317–321, New York, NY, USA, 2008. ACM.

24. Brightwell, R.: A new mpi implementation for cray shmem. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, eds. D. Kranzlmüller, P. Kacsuk, and J. Dongarra, pages 122–130, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

25. Brightwell, R. and Underwood, K. D.: An analysis of nic resource usage for offloading mpi. In 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., pages 183–, April 2004.

26. Patil, A., Shah, A., Gaikwad, S., Mishra, A. A., Kohli, S. S., and Dhage, S.: Fault tolerance in cluster computing system. In 2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, pages 408–412, Oct 2011.

27. Forum, M. P.: Mpi: A message-passing interface standard. Technical report, USA, 1994.

28. Petrini, F., , Hoisie, A., Coll, S., and Frachtenberg, E.: The quadrics network (qsnet): high-performance clustering technology. In HOT 9 Interconnects. Symposium on High Performance Interconnects, pages 125–130, Aug 2001.

29. Pfister, G. F.: An introduction to the infiniband architecture. High Performance Mass Storage and Parallel I/O, 42:617–632, 2001.

30. Subramoni, H., Lai, P., Luo, M., and Panda, D. K.: Rdma over ethernet — a preliminary study. In 2009 IEEE International Conference on Cluster Computing and Workshops, pages 1–9, Aug 2009.

31. Peterson, C., Sutton, J., and Wiley, P.: iwarp: a 100-mops, liw microprocessor for multicomputers. IEEE Micro, 11(3):26–29, June 1991.

32. Kalia, A., Kaminsky, M., and Andersen, D.: Datacenter RPCs can be general and fast. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 1–16, Boston, MA, February 2019. USENIX Association.

33. Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., and et al.: Snap: A microkernel approach to host networking. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

34. Balaji, P., Shah, H., and Panda, D.: Sockets vs rdma interface over 10gigabit networks: An in-depth analysis of the memory traffic bottleneck. 01 2004.

35. Calciu, I., Dice, D., Harris, T., Herlihy, M., Kogan, A., Marathe, V., and Moir, M.: Message passing or shared memory: Evaluating the delegation abstraction for multicores. In International Conference on Principles of Distributed Systems, pages 83–97. Springer, 2013.

36. Petrović, D., Ropars, T., and Schiper, A.: On the performance of delegation over cache-coherent shared memory. In Proceedings of the 2015 International Conference on Distributed Computing and Networking, page 17. ACM, 2015.

37. David, T., Guerraoui, R., and Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 33–48. ACM, 2013.

38. Lozi, J.-P., David, F., Thomas, G., Lawall, J. L., Muller, G., et al.: Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In 2012 USENIX Annual Technical Conference (USENIX ATC 12), pages 65–76, 2012.

39. Multicore locks: The case is not closed yet. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 649–662, Denver, CO, June 2016. USENIX Association.

40. Zipf, G. K.: Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. Addison-Wesley, 1949.

41. Stefan, J. and Boltzmann, L.: On the relation between the emission and absorption of radiant heat. Annalen der Physik und Chemie, page 391–428, 1879.

42. Ojovan, M. I. and Lee, W. E.: Topologically disordered systems at the glass transition. Journal of Physics: Condensed Matter, 18(50):11507, nov 2006.

43. DODDS, P., ROTHMAN, D., and WEITZ, J.: Re-examination of the "3/4-law" of metabolism. Journal of Theoretical Biology, 209(1):9–27, 2001.

44. Jerry Neumann: Power Laws in Venture.

45. xacrimon: dashmap. Version 5.5.3.

46. Chen, C.: Optimizing Key-Value Store Systems Across Multiple Dimensions. Phd thesis, University of Illinois Chicago, 2025.

47. Mitchell, C., Geng, Y., and Li, J.: Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), pages 103–114, San Jose, CA, 2013. USENIX.

48. Dragojević, A., Narayanan, D., Castro, M., and Hodson, O.: Farm: Fast remote memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 401–414, Seattle, WA, 2014. USENIX Association.

49. Barak, D.: Rdmamojo.

50. Graph500 reference benchmark.

51. erpc throughput with 100 gbps connection.

VITA

NAME

Noaman Ahmad

EDUCATION

BS, Computer Science, Lahore University of Management Sciences, Lahore, Pakistan, 2018
MS, Computer Science, University of Illinois Chicago, Chicago, Illinois, 2025

TEACHING

Hands-on Rust: A Practical Introduction (CS 194, Summer 2025)
Machine Organization (CS 261, Spring 2021)
Operating Systems Design and Implementation (CS 461, Fall 2024 & Fall 2025)
Systems Programming (CS 361, Spring 2025)

PUBLICATIONS

Noaman Ahmad, Ben Baenen, Chen Chen and Jakob Eriksson "Delegation with Trust<T>: A Scalable, Type-and Memory-Safe Alternative to Locks" In arXiv preprint 2408.11173 (2024).