Gossamer: A novel programming paradigm for rack-wide applications

by

Noaman Ahmad
BS CS, Lahore University of Management Sciences, 2018

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

A brief dedication to someone you care about. For example, “Dedicated to my cats, Neo and Trinity, who are purrfect in every way.”.

An example of “Dedication” can be found on page 15 of the thesis manual[1].

[1] `http://grad.uic.edu/sites/default/files/pdfs/ThesisManual_rev_06Oct2016.pdf`

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

This section should give a rough overview of each chapter in the thesis, highlighting your contributions. Most importantly, for each one of your papers you are quoting, this section should briefly describe what each author's role / contribution was.

An example of "Contribution of Authors" section is on page 3 of the University's guide to iThenticate[1].

[1] http://grad.uic.edu/sites/default/files/pdfs/Introduction_to_Screening_Your_Thesis_or_Dissertation_using_iThenticate-final_a.pdf

TABLE OF CONTENTS

CHAPTER **PAGE**

1 **INTRODUCTION** 1
1.1 *Trust*<*T*> 3
1.2 Gossamer 4
1.3 Dissertation Organization 5

2 ***TRUST*<*T*>** 6
2.1 Introduction 6
2.2 Background and Motivation 6
2.3 *Trust*<*T*>: The Basics 9
2.3.1 Trust: a reference to an object 10
2.3.2 Trustee - a thread in charge of entrusted properties 11
2.3.3 Fiber - a delegation-aware, light-weight user thread 12
2.3.4 Delegated context 13
2.4 Core API 13
2.4.1 `apply()`: synchronous delegation 13
2.4.2 `apply_then()`: non-blocking delegation 14
2.4.3 `launch()`: apply in a trustee-side fiber 15
2.4.3.1 Atomicity and `launch()` 17
2.4.3.2 Variable-size and other heap-allocated values 17
2.5 Evaluation 18
2.5.1 Fetch and Add: Throughput 20
2.5.1.1 Uniform Access Pattern 21
2.5.1.2 Skewed Access Pattern: Zipfian distribution 22
2.5.2 Fetch and Add: Latency 24
2.5.3 Concurrent key-value store 25
2.6 Legacy Application: Memcached 30
2.6.1 Evaluation 32

3 **GOSSAMER** 36
3.1 Background on RDMA 37
3.1.1 RDMA API 37
3.1.2 Transport types 38
3.2 Gossamer Design 39
3.2.1 Gossamer Daemon 40
3.2.2 Gossamer Process 41
3.2.3 Trustee, Trust and Property 42
3.2.4 Delegation Workflow 43

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

3.2.5 NoRef 44
3.2.6 Reference counting for Trusts 45
3.2.7 Request size and TCP fallback 47
3.2.8 Request/Response Slot Format 48
3.2.9 Trsutee and Client memory layout 51
3.2.10 RDMA Optimizations 51
3.3 Plots 52

4 CONCLUSION 63

CITED LITERATURE 64

VITA 68

LIST OF TABLES

TABLE		PAGE
I	Operations supported by each transport type.	39
II	A small example of delegation code	43

LIST OF FIGURES

FIGURE **PAGE**

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. 9

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. 11

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. 14

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. 16

5 Fetch-and-add throughput vs. object count. *Gossamer* is substantially better than locks in congested settings, and matches lock performance in uncongested settings. TCLocks were not found to be competitive. 19

6 Mean latency vs. offered load. In low-load settings, delegation incurs higher latency than locking. However, latency remains much more stable with increasing load. The near-vertical lines show unbounded latency as capacity is reached. 23

7 Key-value store throughput, with 5% writes and varying table size. 25

8 Key-value store throughput, with varying write percentage. 26

9 Memcached throughput with varying table size. Uniform access distribution. S: stock memcached. 32

10 Memcached throughput with varying table size. Zipfian access distribution. 33

11 High-level design of Gossamer 39

12 A Gossamer process on one machine 41

13 Trustee is a worker and clients can delegate work using a trust that holds information about property and trustee 42

14 Request and Response slot layout 48

16 Throughput of RDMA writes in MOPs for quadratic and reduced number of QPs. Each write is 256 Bytes. and the experiment was conducted on 6 r6615 machines on cloudlab. 54

17 throughput vs number of threads for eRPC vs mpi vs gossamer. . . 55

18 throughput vs latency under load for gsm and eRPC with two machines and 14 threads per machine. 56

LIST OF FIGURES (Continued)

FIGURE **PAGE**

19 throughput vs latency under load for gsm and eRPC with two machines and 28 threads per machine. 57
20 Time taken for SSSP on a graph with 2^{26} nodes and 32 average degree. 58
21 Time taken for SSSP on a graph with 2^{27} nodes and 32 average degree. 59
22 Time taken for SSSP on graphs with 2^n nodes and 32 average degree. Expermient performed on 12 r6615 machines on cloudlab. 60
23 Throughput of key-value stores on up to 8 server clusters and ycsbc workload. 61
24 Throughput of key-value stores on up to 8 server clusters and ycsbd workload. 62

LIST OF FIGURES (Continued)

FIGURE **PAGE**

SUMMARY

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

CHAPTER 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 (1; 2; 3; 4; 5; 6; 7; 8; 9; 10), 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

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

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.

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.

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 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 &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 perfrom 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 comlicates the synchronization mechanisms. Disributed 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 (11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21). 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 acheived 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 2 introduces *Trust<T>*, a delegation based programming abstraction that provides safe access to shared data. Chapter 3 presentes *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 4 discusses future work and provides a conclusion.

CHAPTER 2

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

We present *Trust*<*T*>, 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.

Locking is often said to offer 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, as well as `memcached`, in situations with high lock contention. Moreover, *Trust*<*T*> is competitive with locks even in the absence of lock contention.

2.1 Introduction

2.2 Background and Motivation

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 stallign 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 (22; 5; 1; 4; 6; 7; 8), 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 (23) offers a fully transparent combining-based replacement for locks, by capturing and restoring register contents, and automatically prefetching 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 §2.5 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 (10), 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.

2.3 *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>*.

2.3.1 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].

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

[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.

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

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

```
let cm = Arc::new(Mutex::new(17));
let cm2 = cm.clone();
let thread = spawn(move || {
 *(cm2.lock()?) += 1;
});
*(cm.lock()?) += 1;
thread.join()?;
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.

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.

2.3.2 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.

2.3.3 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.

2.3.4 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 §2.4, we describe multiple ways around this constraint.

2.4 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.

2.4.1 `apply()`: synchronous delegation

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

`apply()` is the primary function for blocking, synchronous delegaion 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.

2.4.2 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.

2.4.3 `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 (24), 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 (10).

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.

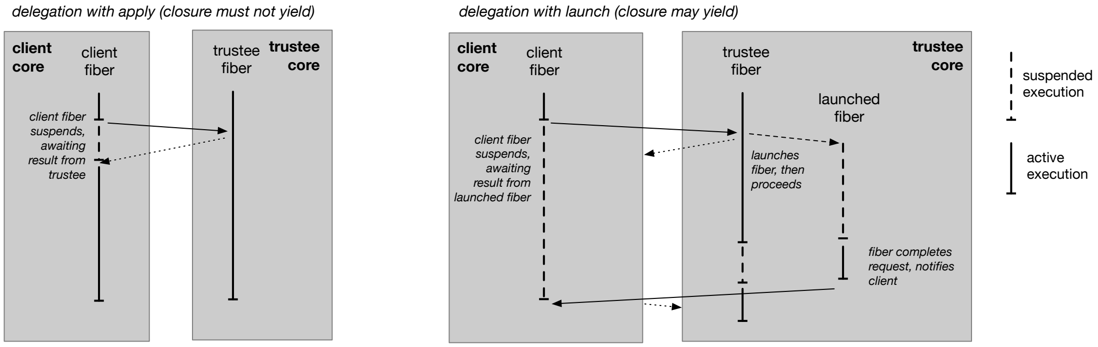

delegation with apply (closure must not yield)
client core
client fiber
trustee fiber
trustee core
client fiber suspends, awaiting result from trustee
delegation with launch (closure may yield)
client core
client fiber
trustee fiber
trustee core
launched fiber
client fiber suspends, awaiting result from launched fiber
launches fiber, then proceeds
fiber completes request, notifies client
suspended execution
active execution

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.

2.4.3.1 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]

2.4.3.2 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 §**??**), 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

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

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
```

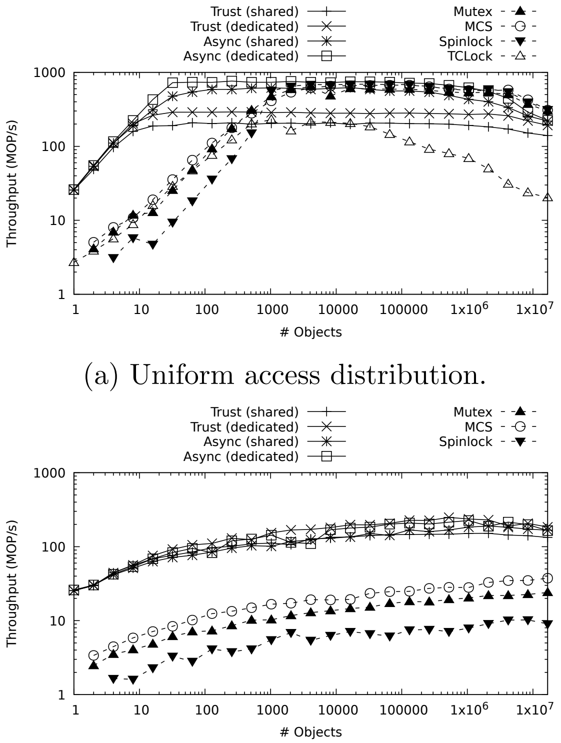
Here, the `w:` argument is any type `V:Serialize+Deserialize`, using the popular traits from the `serde` crate. That is, any type that can be serialized and deserialized, may pass over the delegation channel in serialized form. If more than one argument is needed, these may be passed as a tuple. Thus, 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.

2.5 Evaluation

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



(a) Uniform access distribution.

(b) Zipfian access distribution, $\alpha = 1$.

Figure 5: Fetch-and-add throughput vs. object count. *Gossamer* is substantially better than locks in congested settings, and matches lock performance in uncongested settings. TCLocks were not found to be competitive.

2.5.1 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 (1; 2; 3; 4; 5; 6; 7; 8; 9), 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. 5a, we also include TCLocks, a recent combining approach offering a transparent replacement for standard locks, via the Litl lock wrapper (**?**) 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*).

2.5.1.1 Uniform Access Pattern

Figure 5a 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 (23) 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 programs (as Rust now uses built-in locks rather than `libpthreads` wrappers). We thus elide TCLocks from the remainder of the evaluation.

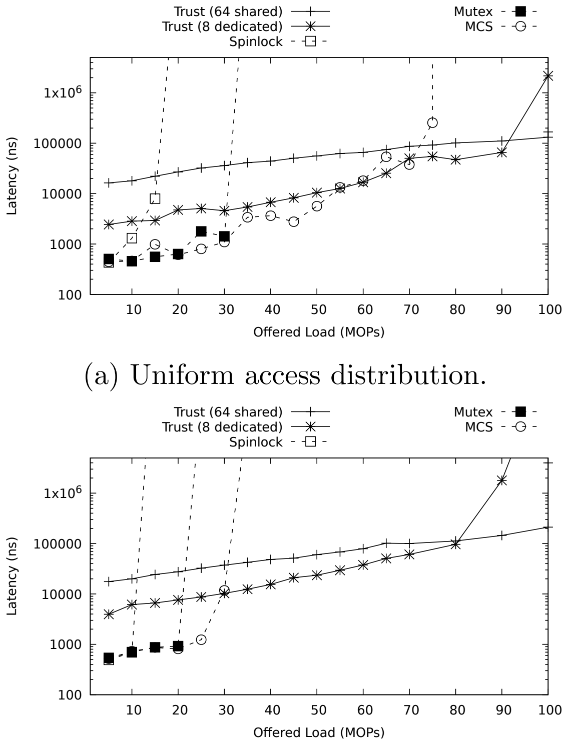
2.5.1.2 Skewed Access Pattern: Zipfian distribution

Zipf's law (?) 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 (?; ?; ?; ?; ?), sometimes with a greater value for α. The higher the α, the more pronounced is the effect of popular keys, resulting in congestion.

Figure 5b 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, *Gossamer* 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

[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).



(a) Uniform access distribution.

(b) Zipfian access distribution, $\alpha = 1$.

Figure 6: Mean latency vs. offered load. In low-load settings, delegation incurs higher latency than locking. However, latency remains much more stable with increasing load. The near-vertical lines show unbounded latency as capacity is reached.

is inevitable. By comparison, a single *Gossamer* 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.

2.5.2 Fetch and Add: Latency

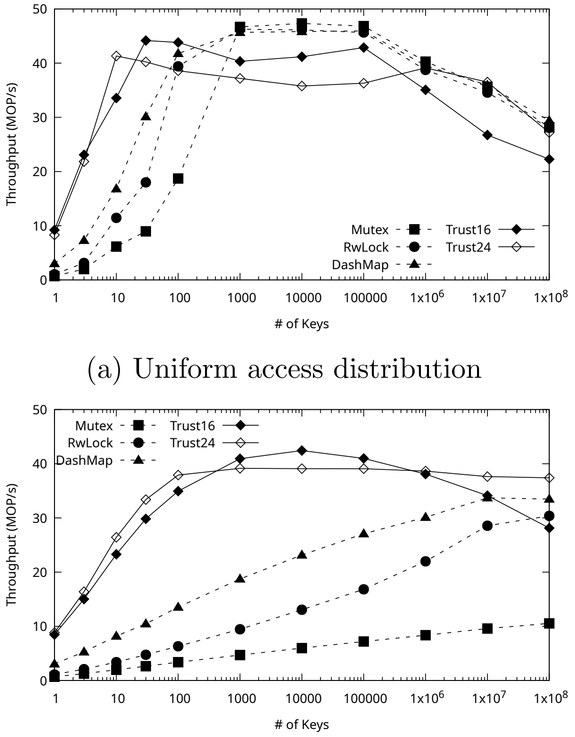
Next we measure mean latency for a scenario with 64 objects (uniform access distribution), and 1,000,000 objects (Zipfian access distribution), while varying the offered load. We show delegation results with 8 dedicated trustee cores, and with 64 shared trustee cores[1]. We also plot the results for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

At low load, low contention results in low latency for locking, a ideal situation for locks. However, as load increases, the locks eventually reach capacity, resulting in a rapid rise in latency. With *Gossamer*, 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, *Gossamer* 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 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

[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.



(a) Uniform access distribution

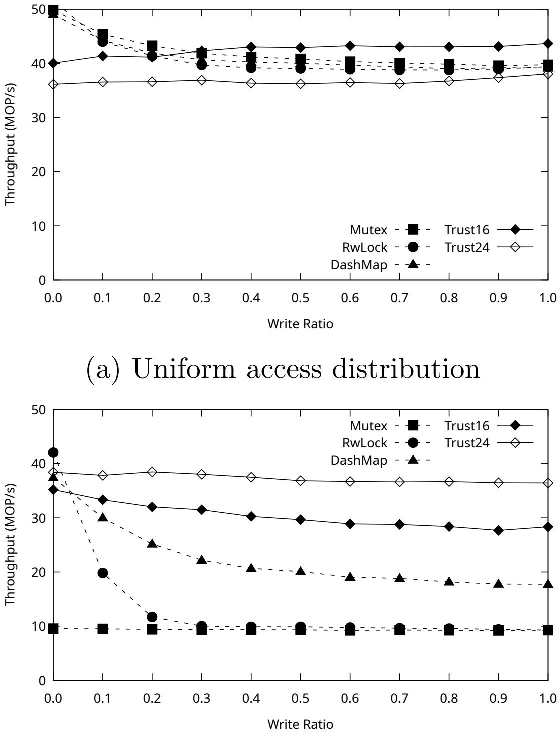
(b) Zipfian access distribution

Figure 7: Key-value store throughput, with 5% writes and varying table size.

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.

2.5.3 Concurrent key-value store

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



(a) Uniform access distribution

(b) Zipfian access distribution

Figure 8: Key-value store throughput, with varying write percentage.

are connected by 100 Gbps Ethernet. We compare our *Gossamer* based solution to Dashmap (**?**), 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 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 *Gossamer* 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. 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 7a–7b 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 (**?**) 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, *Gossamer* 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, *Gossamer* 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 *Gossamer* in that they allow concurrent reader access, while *Gossamer* exclusively allows trustees to access the underlying data structure. To better understand this dynamic, Figures 8a–8b 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 7a–7b. 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 *Gossamer* 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, *Gossamer* 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 *Gossamer* over locking in this application is clear.

2.6 Legacy Application: Memcached

We also port memcached version 1.6.20 to *Gossamer* 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 *Gossamer*, 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 2.5.3 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 *Gossamer* 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, *Gossamer* includes an equivalent of Rust's `Arc<T>` which allows multiple ownership of read-only values.

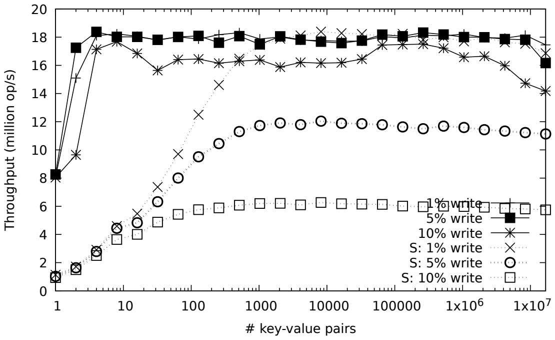


Figure 9: Memcached throughput with varying table size. Uniform access distribution. S: stock memcached.

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 *Gossamer* 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.

2.6.1 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

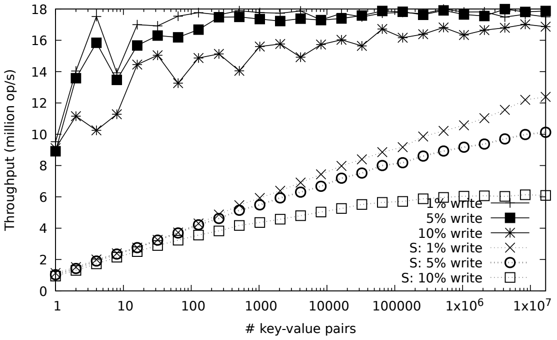


Figure 10: Memcached throughput with varying table size. Zipfian access distribution.

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.

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

Figures Figure 9–Figure 10 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 *Gossamer* 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 *Gossamer*, 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 *Gossamer* 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.

CHAPTER 3

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 describe the design and implementation of *Gossamer*, a new distributed programming model that uses delegation to distribute computing work. We use RDMA for communication between machines as it provides both low latency and high throughput. Moreover, RDMA offers mechanism to write in a remote machine's memory without the involvement of CPU on that machine. This is especially useful when implementing a programming model that can seamlessly work on both a single machine and multiple machines.

Gossamer allows developers to write applications that span multiple machines as if they would be running on a single machine. Programmers can use the API provided by *Gossamer* to delegate a rust closure that can even be executed on a remote machine. *Gossamer* uses light weight user-space threads called *fibers* for multi-threading on both machines. In our evaluation, we compare the performance of applications running delegation over RDMA that of applications running on a single machine, while using the same number of CPUs.

3.1 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 (25; 26; 27; 28), 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.

3.1.1 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 sockets 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* (29; 30).

3.1.2 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 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

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

TABLE I: Operations supported by each transport type.

only means that the local buffers can be reused safely, with no indiation of the remote memory's status. A successful work completion also indicates that any unsignaled work posted earlier has finished successfully (31). *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.

3.2 Gossamer Design

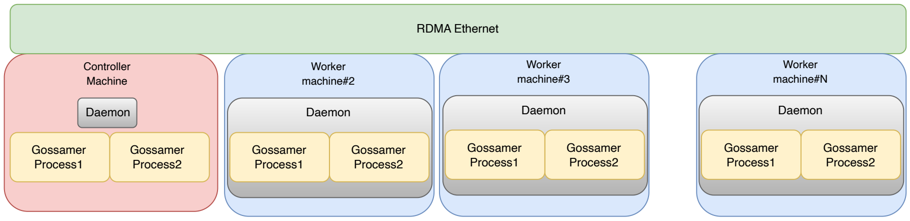


Figure 11: High-level design of Gossamer

Gossamer allows developers to write rack-wide applications just like single machine applications. Programmers will use the delegation framework to write applications that would be distributed by gossamer with very few changes to the code. Figure 11 shows a high level view of what a rack with multiple *Gossamer* applications would look like. Developers would use the controller machine 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 will be the machine that users log into for terminal access. Each machine in the rack would have a daemon running that would always be 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.

3.2.1 Gossamer Daemon

As shown in Figure 12, each machine has *Gossamer* daemon running. The daemon is responsible for managing any *Gossamer* processes running on the rack at any time. To launch an application, users will connect with the controller machine and tell the daemon to start the process specifying how many kernel threads the process should use. The daemon will contact the remote machines and tell the daemons to launch the process. The daemon will store metadata like gossamer-id, machine-ip etc. The daemon is also responsible for detecting crashes on any local instance of a *Gossamer* process and terminating across the whole rack. This means that

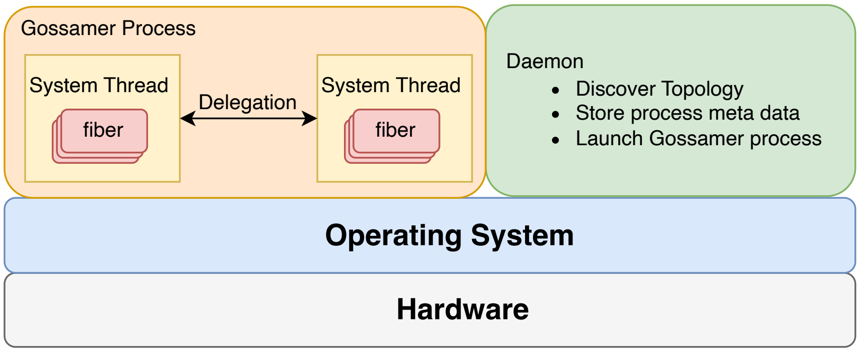


Figure 12: A Gossamer process on one machine

Gossamer processes are fate sharing in the same way single machine processes crash if a single thread crashes.

3.2.2 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 3.2.8), 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 kernel thread for concurrent operation as shown in Figure 12. Fibers are lightweight, userspace threads that can spawn on any kernel 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 th operating system, a single instance of *Gossamer* process behaves like any other normal process. The main difference would be that instead of using system calls like *getpid()*, a *Gossamer* process contacts the daemon to get any metadata needed.

3.2.3 Trustee, Trust and Property

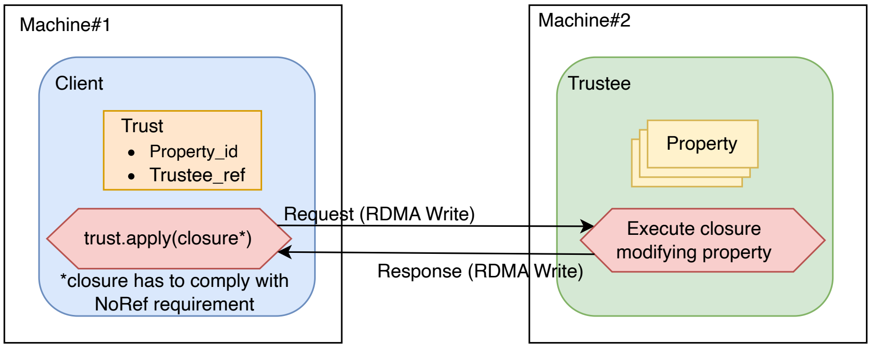


Figure 13: Trustee is a worker and clients can delegate work using a trust that holds information about property and trustee

Gossamer builds on the Trust/Trustee concept from (32), 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 13 shows

```
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 II: A small example of delegation code

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.

3.2.4 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 yeilding the runtime. Each thread has a fiber that periodicaly 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. On the trustee side, one of the fibers is dedicated to polling for any incoming requests. Since any incoming requests from the same thread will be contigous in memory, it can complete all of them and then send all of the responses in a single batch. The fiber that polls for reponses 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 yeilding 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 II shows a small example program that uses both types of delegation.

3.2.5 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 13. This however limits severly 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* prvides 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.

3.2.6 Reference counting for Trusts

Trusts in *Gossamer* 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 acheive that trusts need to be reference counted, but a naive integer count will not suffice due to a combination of the following two issues.

- *Gossamer* 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, *Gossamer* 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, *Gossamer* 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.

3.2.7 Request size and TCP fallback

As describe in 3.2.4, the requsts 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 contigous in memory. This is achieved by the trustee having a pre-allocated sapce 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 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 1kB 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 aound 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 severly as TCP is much slower than RDMA. We use TCP here instead of RDMA because, as mentioned in 3.1.1, RDMA needs the memory used to be pre-registered with the NIC. This would also place an upper limit on how much data can be sent using RDMA at once. 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.

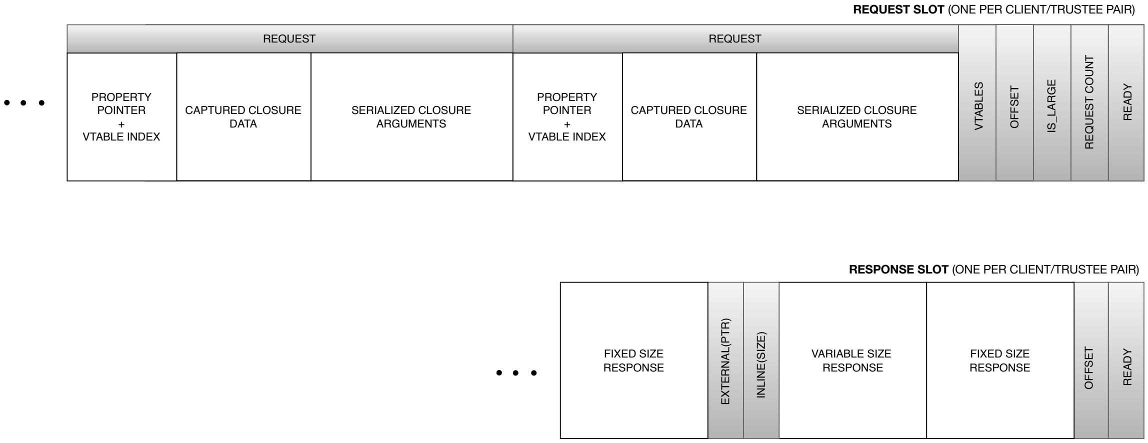

REQUEST SLOT (ONE PER CLIENT/TRUSTEE PAIR)
REQUEST
REQUEST
PROPERTY POINTER + VTABLE INDEX
CAPTURED CLOSURE DATA
SERIALIZED CLOSURE ARGUMENTS
PROPERTY POINTER + VTABLE INDEX
CAPTURED CLOSURE DATA
SERIALIZED CLOSURE ARGUMENTS
VTABLES
OFFSET
IS_LARGE
REQUEST COUNT
READY
RESPONSE SLOT (ONE PER CLIENT/TRUSTEE PAIR)
FIXED SIZE RESPONSE
EXTERNAL(PTR)
INLINE(SIZE)
VARIABLE SIZE RESPONSE
FIXED SIZE RESPONSE
OFFSET
READY

Figure 14: Request and Response slot layout

3.2.8 Request/Response Slot Format

As discussed in the previos 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. 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 propoerty 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 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 whe submitting a request. While the known size of requests allows us to pack as many requests in a request slot as would fit, it can lead to situations where not all of the responses will fit in a single response slot. Based on this observation, the responses are divided in two types.

- Statically known sized responses. Reffered to as fixed size response here.
- Dynamically sized responses.

Both of these can be further divided into responses that are small and can fit inside the slot or large and cannot fit inside the slot. For fixed size and small responses, the response can be sent inline without any metadata. For dynamically sized small responses, the size of the response is accompanied with them. For large responses, both fixed an dynamically sized, the response is sent using TCP and the pointer to the remote buffer is sent in the response slot. In addition, if the response size is zero and statically known, then nothing is sent in the slot. Figure 14 shows the format for both the request and response slots.

In addition to the reqest/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 ther 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 reponses, 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 reasy 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 reequest in the slot or muliptle small ones. As discussed in **??**, 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.

3.2.9 Trsutee and Client memory layout

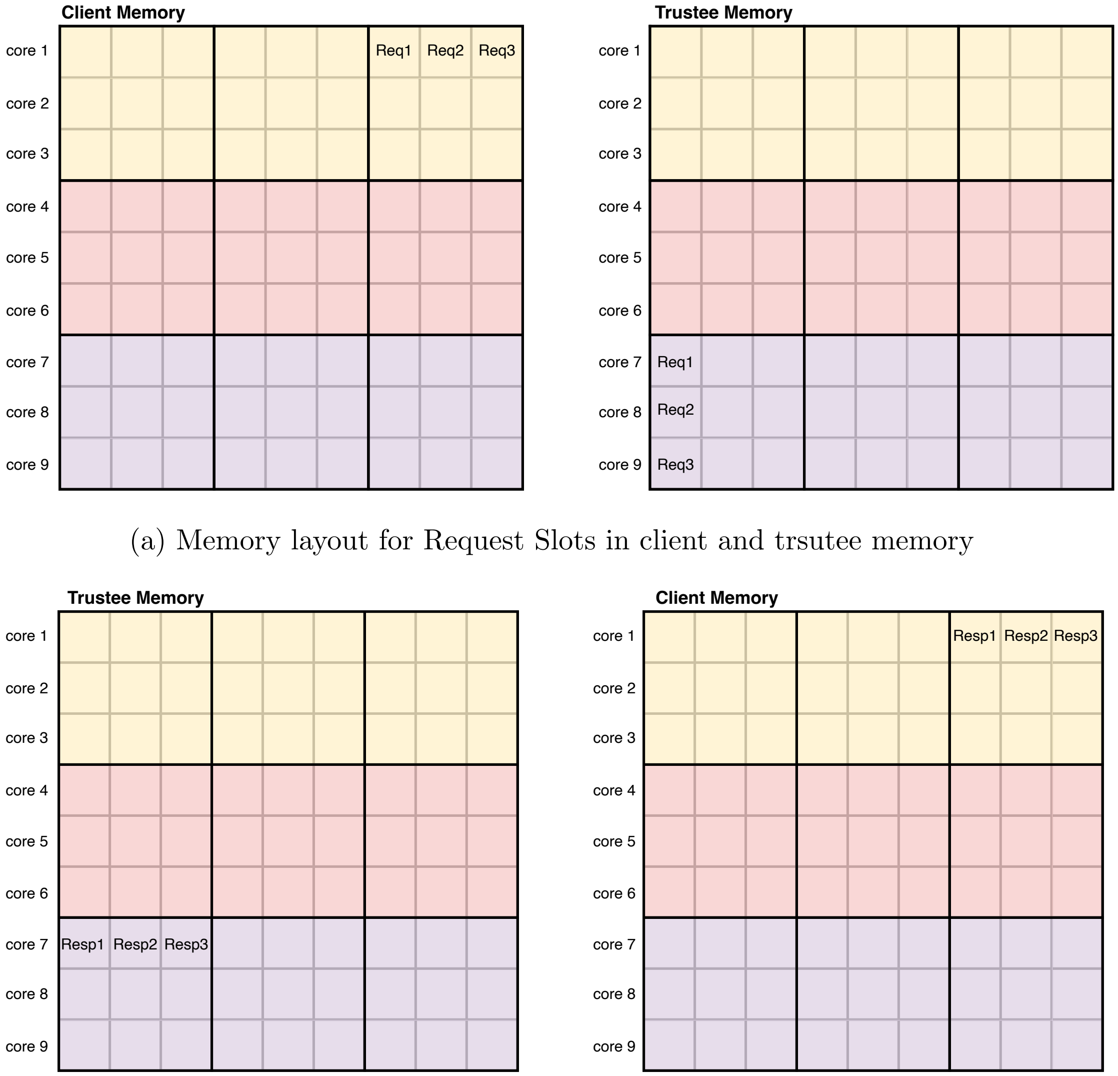
3.2.10 RDMA Optimizations

As discussed in 3.1.1, rdma communications happen via a queue pair (QP). Theoretically each machine pair only needs to have a single QP resulting in $num\_machines^2$ QPs. However, this means sharing a QP among many threads. The QPs use a spin lock internally to mack sure QPs can 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

$(num\_machines * num\_threads\_per\_machine)^2$. This also resluts 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 resluts in NIC running out of SRAM, which in turn results in cache misses. We use a middle of the road 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 $num\_machines^2 * num\_threads\_per\_machine$. Figure 16 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 where as the system with quadratic number of QPs starts to fall off, resulting in a 25% decrease at the extreme end.

3.3 Plots

Various plot comparing gossamer with eRPC and graph500 reference code.



(a) Memory layout for Request Slots in client and trsutee memory

(b) Memory layout for Response Slots in client and trsutee memory

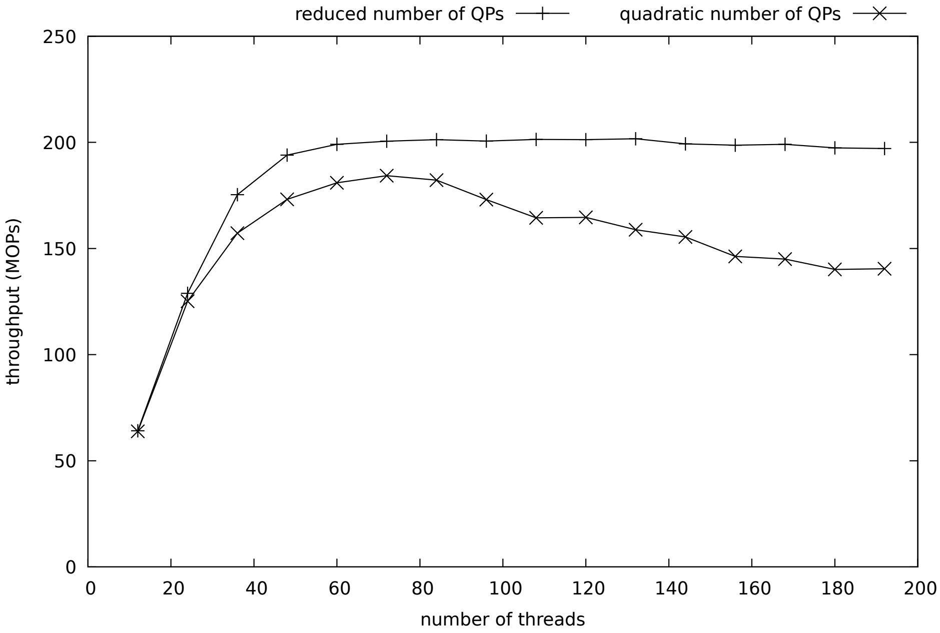


Figure 16: Throughput of RDMA writes in MOPs for quadratic and reduced number of QPs. Each write is 256 Bytes. and the experiment was conducted on 6 r6615 machines on cloudlab.

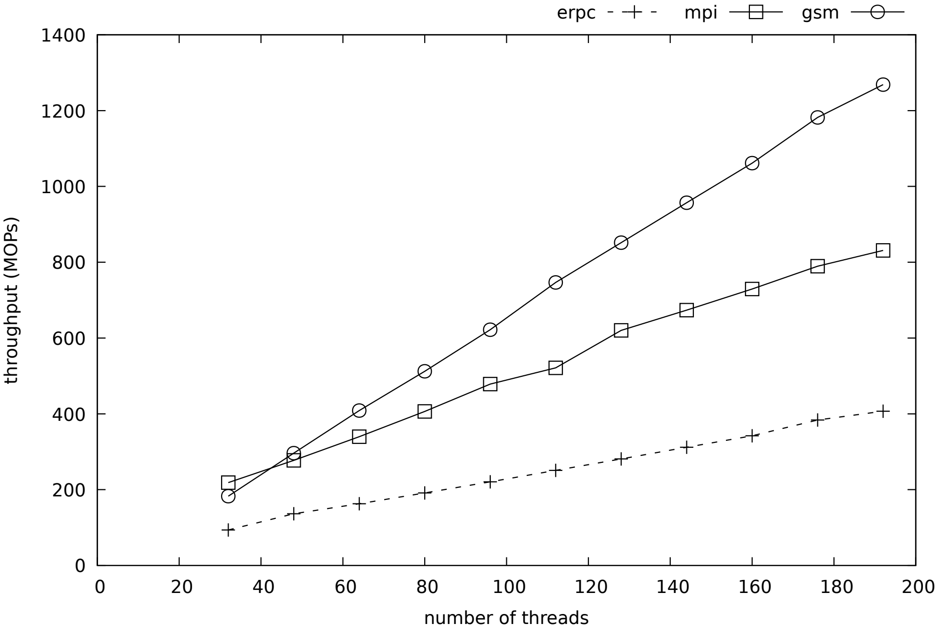


Figure 17: throughput vs number of threads for eRPC vs mpi vs gossamer.

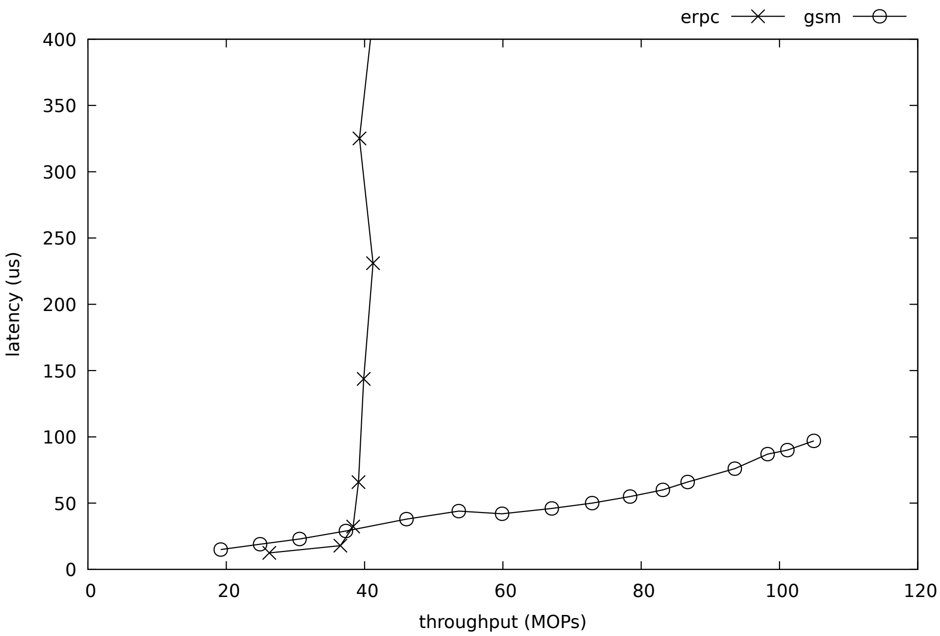


Figure 18: throughput vs latency under load for gsm and eRPC with two machines and 14 threads per machine.

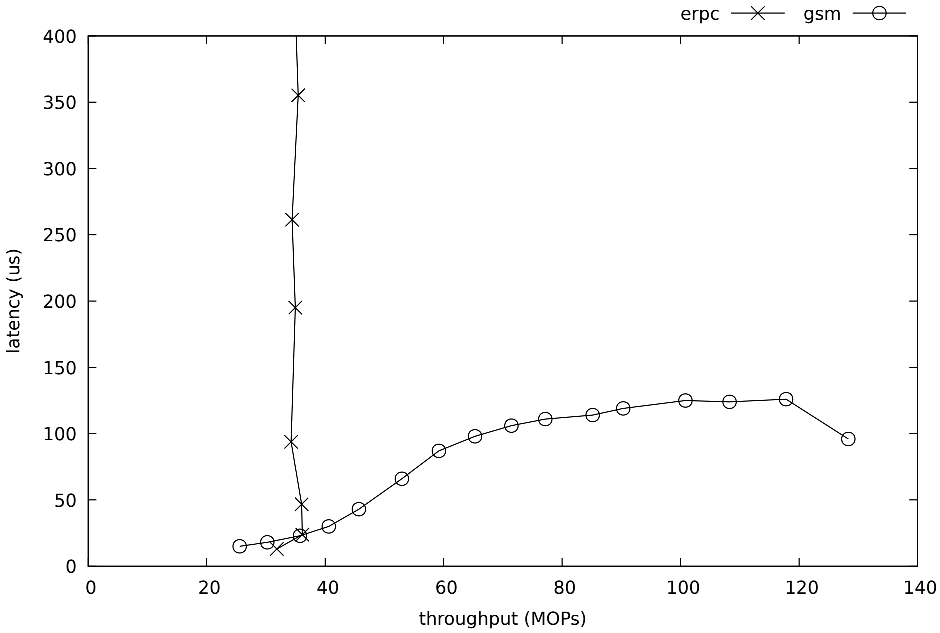


Figure 19: throughput vs latency under load for gsm and eRPC with two machines and 28 threads per machine.

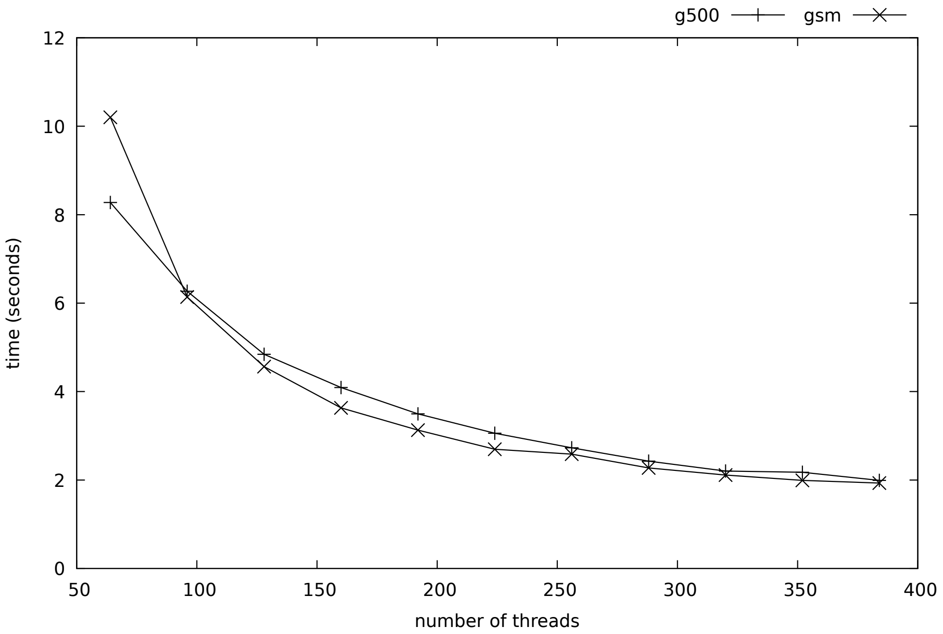


Figure 20: Time taken for SSSP on a graph with 2^{26} nodes and 32 average degree.

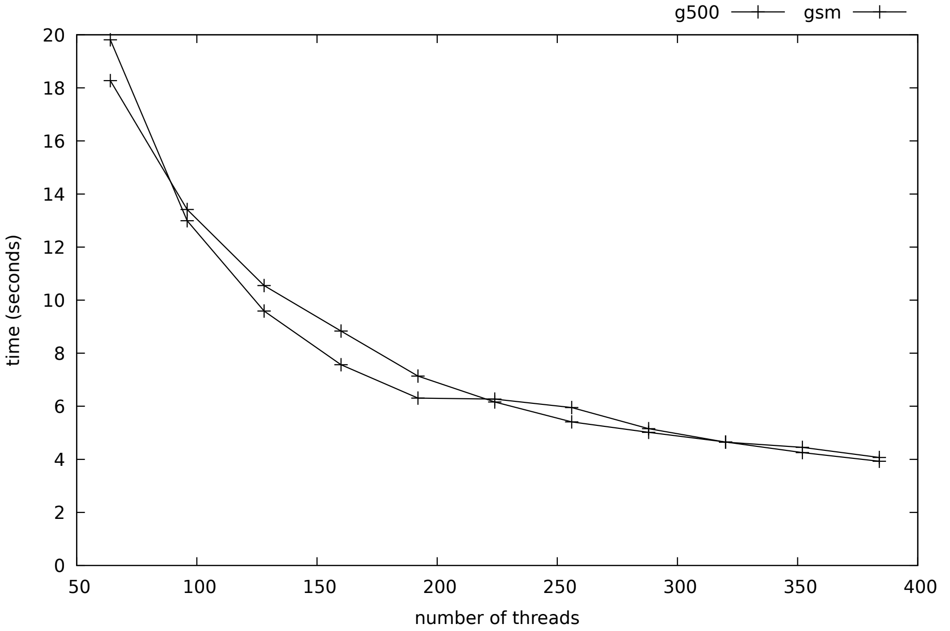


Figure 21: Time taken for SSSP on a graph with 2^{27} nodes and 32 average degree.

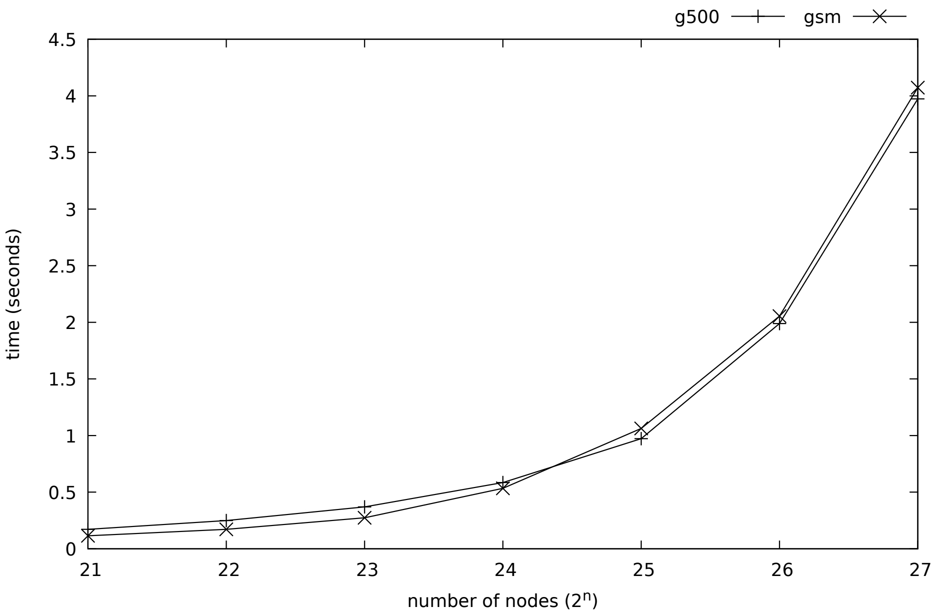


Figure 22: Time taken for SSSP on graphs with 2^n nodes and 32 average degree. Expermient performed on 12 r6615 machines on cloudlab.

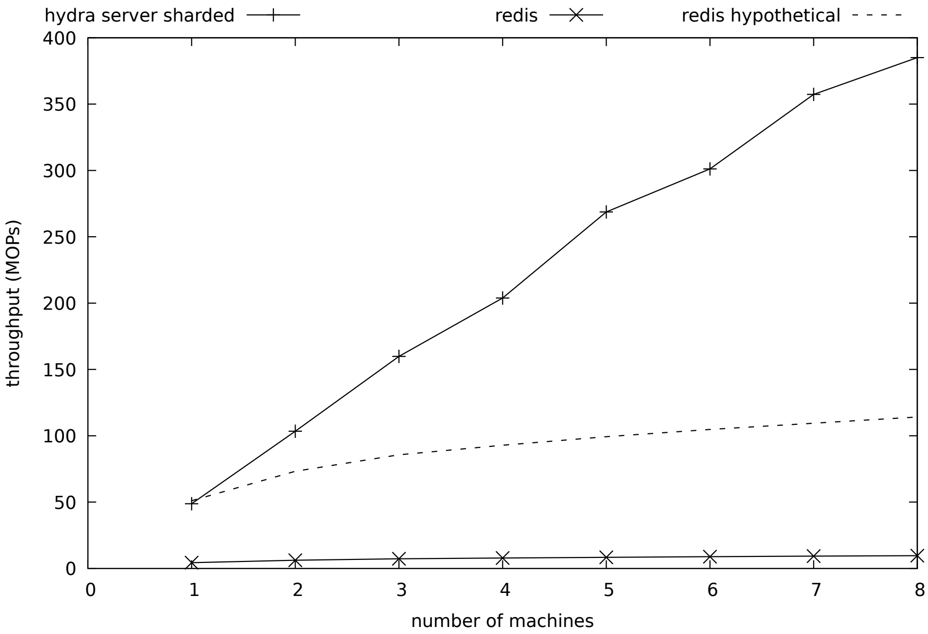


Figure 23: Throughput of key-value stores on up to 8 server clusters and ycsbc workload.

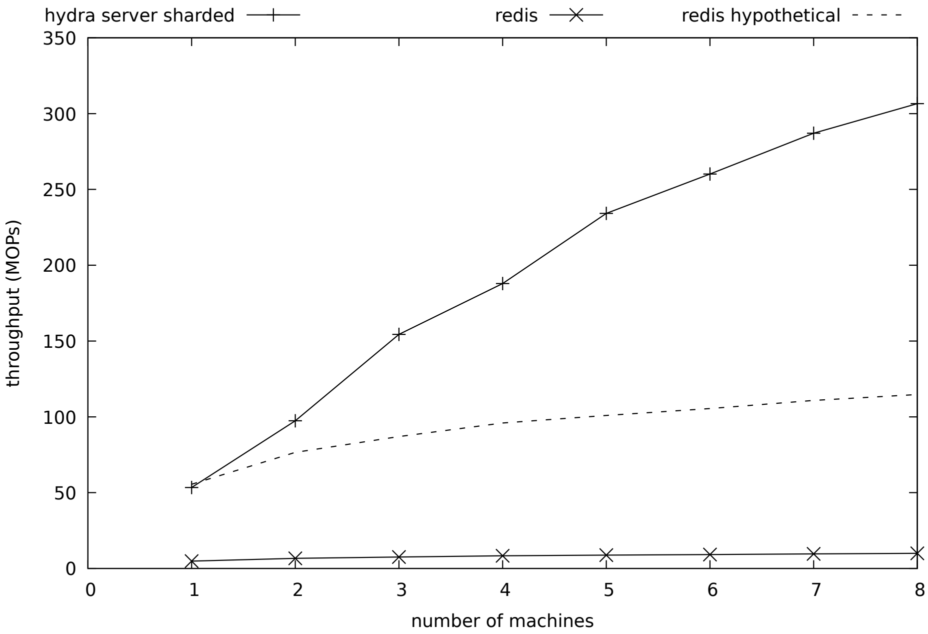


Figure 24: Throughput of key-value stores on up to 8 server clusters and ycsbd workload.

CHAPTER 4

CONCLUSION

CITED LITERATURE

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

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

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

9. 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.

10. 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.

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

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

13. 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.

14. 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.

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

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

17. 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.

18. 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.

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

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

21. 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.

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

23. 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.

24. 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 USENIX Annual Technical Conference, pages 65–76, 2012.

25. 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.

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

27. 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.

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

29. 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.

30. 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.

31. Barak, D.: Rdmamojo.

32. Baenen, B. and Eriksson, J.: Trust¡t¿: A typesafe programming abstraction for delegation in rust. Technical report, University of Illinois at Chicago, Department of Computer Science, 2021.

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 (CS194, Summer 2025)
PUBLICATIONS	