

SWARM: Replicating Shared Disaggregated-Memory Data in No Time

Antoine Murat*

antoine.murat@epfl.ch

École Polytechnique Fédérale de
Lausanne (EPFL)
Switzerland

Igor Zablotchi

igor@mystenlabs.com

Mysten Labs
Switzerland

Clément Burgelin*

clement.burgelin@epfl.ch

École Polytechnique Fédérale de
Lausanne (EPFL)
Switzerland

Marcos K. Aguilera

marcos-k.aguilera@broadcom.com

VMware Research Group
United States

Athanasis Xygkis

athanasis.xygkis@oracle.com

Oracle Labs
Switzerland

Rachid Guerraoui

rachid.guerraoui@epfl.ch

École Polytechnique Fédérale de
Lausanne (EPFL)
Switzerland

Abstract

Memory disaggregation is an emerging data center architecture that improves resource utilization and scalability. Replication is key to ensure the fault tolerance of applications, but replicating shared data in disaggregated memory is hard. We propose SWARM (Swift WAit-free Replication in disaggregated Memory), the first replication scheme for in-disaggregated-memory shared objects to provide (1) single-roundtrip READS and WRITES in the common case, (2) strong consistency (linearizability), and (3) strong liveness (wait-freedom). SWARM makes two independent contributions. The first is Safe-Guess, a novel wait-free replication protocol with single-roundtrip operations. The second is In-n-Out, a novel technique to provide conditional atomic update and atomic retrieval of large buffers in disaggregated memory in one roundtrip. Using SWARM, we build SWARM-KV, a low-latency, strongly consistent and highly available disaggregated key-value store. We evaluate SWARM-KV and find that it has marginal latency overhead compared to an unreplicated key-value store, and that it offers much lower latency and better availability than FUSEE, a state-of-the-art replicated disaggregated key-value store.

1 Introduction

Memory disaggregation allows servers to access external memory provided by a set of memory nodes connected to a low-latency high-throughput fabric, using technologies such as RDMA [30] and, more recently, CXL [10]. Memory disaggregation improves utilization of memory and hence decreases its relative cost [21, 39, 52]. However, failures of the memory nodes can severely disrupt users and decrease overall system reliability. To tolerate failures, traditional systems replicate data so that it remains accessible even if some of the replicas fail. Unfortunately, current replication schemes are ill-suited for disaggregated memory, because of two issues. First, they require several network roundtrips to the memory nodes, which at least doubles the latency of disaggregated memory (§2.3). Second, they often require running code next to the data for concurrency control (e.g.,

to track timestamps, filter messages with old timestamps, issue a vote, etc.), which may be impossible or undesirable: the disaggregated memory may have no compute capability (e.g., with CXL), or it may impose additional overhead to run code (e.g., RDMA requires costly two-sided operations [31, 32]). These issues are particularly problematic for systems requiring low latency, such as data stores in trading systems, control systems, and microsecond-scale microservices.

We propose SWARM (Swift WAit-free Replication in disaggregated Memory), a new replication protocol for shared data in disaggregated memory. SWARM provides several important features for our setting: it can read or write shared objects in one network roundtrip in the common case, when there are no failures or contention, and clocks are nearly synchronized; it can replicate both small and big objects; it provides (a) strong consistency in the form of linearizability [26], and (b) strong liveness in the form of wait-freedom [25].

Designing SWARM required overcoming multiple challenges. First, the protocol must handle concurrent READS and WRITES on the same object, while ensuring strong consistency, with little communication in the common case. Second, the protocol cannot run computation in the memory nodes. Third, the protocol must atomically handle READS and WRITES of objects that are larger than the largest atomic updates supported by the disaggregated memory.

We address these challenges through a two-step modular design. First, we design Safe-Guess, a protocol that provides most of the properties we desire (single-roundtrip operations, linearizability, wait-freedom), but requires memory nodes to support a *max register* object that tracks the maximum value written to it out of large values (larger than the word size). Internally, Safe-Guess’ WRITES guess ordering timestamps speculatively to save a roundtrip; commonly, guesses are proven correct and WRITES finish in one roundtrip; otherwise, Safe-Guess uses a novel wait-free locking mechanism called *timestamp locks* to resolve conflicts with potential readers, so that WRITES can safely re-execute with better timestamps if needed.

Second, we introduce a novel technique, called In-n-Out, that allows us to implement, on memory nodes with no

*These authors contributed equally to this work.

compute, a max register for large values that takes a single roundtrip in the common case. Doing so requires providing conditional updates on large values, but existing techniques take multiple roundtrips as they incur locking or pointer-chasing overheads to perform updates in-place or out-of-place, respectively. Our In-n-Out scheme achieves one roundtrip by simultaneously doing in-place and out-of-place updates: it uses in-place updates without locks to execute quickly when there is no contention, and it falls back to out-of-place updates when an in-place READ finds corrupted data. Furthermore, In-n-Out optimizes out-of-place WRITES to run in one roundtrip using hardware ordering guarantees.

We combine Safe-Guess and In-n-Out to obtain SWARM. To demonstrate the utility of SWARM, we build SWARM-KV, a strongly consistent and highly available disaggregated key-value store with ultra-low latency where clients directly access data on memory nodes. SWARM-KV can serve INSERTS, UPDATES, GETS and DELETES in a single roundtrip and continue operating despite the failure of clients and memory nodes. We evaluate SWARM-KV against two main baselines: (1) a raw disaggregated key-value store that is unreplicated and has no concurrency control but establishes how fast a key-value store can be; and (2) FUSEE [47], the state-of-the-art for disaggregated key-value stores, which is replicated and supports concurrency. Through YCSB benchmarks, we observe that SWARM-KV has sub-RTT latency overhead compared to the raw baseline (0.5 μ s for GETS and 1.5 μ s for UPDATES), and has significantly lower latency than FUSEE (up to 2 \times faster GETS and 3.4 \times faster UPDATES), as well as better availability (no downtime). The cost of using SWARM-KV is higher disaggregated-memory consumption (2 \times FUSEE's).

In summary, our contributions are the following:

- SWARM: the first replication protocol for disaggregated memory to offer strong consistency (linearizability), strong liveness (wait-freedom), and READS and WRITES that complete in one roundtrip in the common case. SWARM need not run code at the memory nodes.
- Safe-Guess: a new protocol that directly provides the above features if memory nodes can provide atomic conditional updates to large objects in one roundtrip.
- In-n-Out: a novel technique for the atomic and conditional update of large disaggregated memory objects in one roundtrip with no compute at memory nodes.
- SWARM-KV: a low-latency, strongly consistent and highly available disaggregated key-value store.
- A thorough evaluation of SWARM-KV using YCSB, which finds much improved latency and availability against the state of the art.

SWARM-KV is open-source, available at <https://github.com/LPD-EPFL/swarm-kv>. The appendix has detailed correctness proofs and roundtrip complexity analyses.

2 Background

2.1 Setting

We consider a data center system where servers host applications that can access disaggregated memory. This disaggregated memory is provided by a number of *memory nodes* connected to a low-latency high-bandwidth interconnect. Our goal is to replicate the data in disaggregated memory across a subset of memory nodes, through a protocol that application threads use to read and write data. Our protocol, SWARM, is agnostic to the disaggregation technology, as long as it satisfies three properties. First, the memory supports READ and WRITE operations, though these need not be atomic (e.g., concurrent WRITES may clobber each other, while a READ concurrent with a WRITE may return partly written data). Second, the memory supports a 64-bit atomic CAS (compare-and-swap). Third, application threads can pipeline two update operations to be executed in order at the same memory node (i.e., if the second update is visible, so is the first), and they execute in one roundtrip. This setting can be realized with RDMA [6, 30], but we believe it could also be realized with other technologies like CXL [10] in the future. Although SWARM makes no synchrony assumption for replication, it requires eventual synchrony to recycle memory.

2.2 Failure Assumptions

The system is subject to crash failures that affect application threads and memory nodes. Any number of application threads can fail, but we assume a majority of memory nodes remains alive (e.g., a deployment with 3 or 5 memory nodes can tolerate 1 or 2 failed nodes, respectively). We do not consider Byzantine failures. The network and disaggregated memory interconnect may also fail, but they eventually recover. The system remains safe under partitions, but application threads need to be able to reach a majority of memory nodes to make progress. While the set of application threads can change over time, SWARM operations are wait-free only if the number of concurrent application threads is bounded.

2.3 Read-Write Replication and the ABD Protocol

We are interested in strongly consistent replication providing linearizability [26], which ensures that operations take effect atomically at a single point in time. Linearizable replication schemes can be divided into two types. State machine replication [34, 35] implements state machines with arbitrary operations, while read-write replication [4] implements a *register* object with a READ and a WRITE operation, such that READS return the value of the latest WRITE [5]. State machine replication tends to be more complex and less efficient than read-write replication because it requires solving consensus [36] which incurs more roundtrips (e.g., clients first contact a leader, then the leader runs a consensus protocol), and is not wait-free (it can be delayed for arbitrarily long in periods without synchrony). Therefore, we focus on read-write

replication, which is simpler and suffices for many use cases, such as key-value stores and storage systems [14, 16, 27, 50].

Algorithm 1. ABD expressed using a max register

```

1 M = (ts: (i: 0, tid: ⊥), v: ⊥) // Max Register
2
3 def WRITE(v): // this block is not atomic
4     fresh_ts = (M.READ().ts.i + 1, tid)
5     M.WRITE((fresh_ts, v))
6
7 def READ():
8     return M.READ().v

```

The canonical read-write replication protocol is ABD [4, 42], named after its creators Attiya, Bar-Noy, and Dolev. Algorithm 1 shows ABD, expressed using a *max register*. A max register is an object that supports READ and WRITE operations such that a READ returns a value greater than or equal to the values of all operations that completed before it (see Appendix A for more details).¹

In ABD, each value written to the max register is augmented with a logical timestamp used to order it. This timestamp comprises a thread id to break ties of otherwise identical timestamps from different writers. To WRITE, a thread first picks a *fresh* timestamp—one that is higher than the timestamps of all completed WRITES—by reading the max register and adding 1 to the timestamp it finds. Then, it writes its value alongside the fresh timestamp to the max register. To READ, a thread reads the max register and returns the value it finds, ignoring the timestamp.

The max register itself is implemented from multiple max registers that may crash, assuming a majority remains alive. Briefly, to WRITE to the max register, we write to a majority of the underlying registers. To READ, we read a majority of the underlying registers, pick the largest value, ensure it is written to a majority of the underlying registers (which may require an additional roundtrip in unlucky cases), and return it. Appendix A provides the pseudocode and a full proof of correctness of this reliable max register implementation.

2.4 Challenges

There are many challenges in achieving our goal of replicating shared data in disaggregated memory with strong consistency, strong liveness, and low latency.

One-roundtrip. While ABD offers strong consistency and liveness, its WRITE operation incurs two roundtrips to disaggregated memory, which doubles the latency compared to a non-replicated system. An ideal replication scheme would always take one roundtrip, but this is provably unachievable [14], so we settle for one roundtrip most of the time. The ABD algorithm suggests that there is little opportunity to save a roundtrip: because writing the value to disaggregated memory is mandatory, we can eliminate only the first roundtrip, used to obtain a fresh timestamp. We can eliminate this step by guessing a fresh timestamp, but doing so is tricky:

¹Our definition of a max register is weaker than [2], but suffices for us.

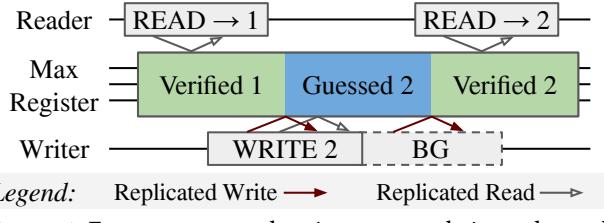


Figure 1. Fast READS complete in one roundtrip to the replicas in disaggregated memory by finding a value tagged as verified. Fast WRITES complete in one roundtrip by writing their value with a guessed timestamp and confirming the freshness of the latter via a parallel READ. Successful WRITES are tagged as verified in the background.

if the guessed timestamp is not fresh, the WRITE must be retried, which effectively writes the value with two different timestamps; this can cause READS to oscillate between two values, violating linearizability. We address this challenge with Safe-Guess, a novel replication protocol (Section 3).

The limits of disaggregated memory. ABD is designed to replicate data in the memory of processes that can run arbitrarily logic. Disaggregated memory, however, is limited in its compute capabilities, which makes it hard to implement ABD’s max register. The latter indeed requires a conditional update (i.e., only write if the provided value is larger than the current value) and atomic READS/WRITES on large values (larger than the word size). While one can extend the atomicity of disaggregated memory by leveraging 64-bit CASES, these incur high latency because of locking or pointer chasing [49]. We address this challenge with In-n-Out, a novel technique to conditionally and atomically manipulate large disaggregated memory buffers in one roundtrip (Section 4).

3 Safe-Guess

Safe-Guess is SWARM’s core replication protocol. Section 3.1 gives an informal overview. Section 3.2 gives its algorithms. Section 3.3 explains how to implement Safe-Guess’ new building block, the *timestamp lock*. A full proof of correctness is given in Appendices B (timestamp lock) and C (Safe-Guess).

3.1 Overview

Safe-Guess adopts a fast-slow path design so that READS and WRITES complete in one roundtrip in the common case, when there are no failures or contention, and clocks are nearly synchronized (Figure 1). In other cases, Safe-Guess runs a slow path with more roundtrips, but preserves wait-freedom.

Similarly to the ABD protocol (§2.3), Safe-Guess orders WRITES using timestamps. A timestamp t associated with a WRITE W is *fresh* if t is greater than the timestamps of all WRITES that completed before W started; otherwise, t is *stale*. ABD requires obtaining a fresh timestamp for each WRITE, which incurs an additional network roundtrip. Safe-Guess avoids this overhead by guessing a timestamp that is fresh most of the time, but not always. When the timestamp is stale, Safe-Guess retries with a new fresh timestamp.

While using stale timestamps might seem harmless, merely requiring a retry, a major difficulty lies in diagnosing such timestamps as stale. Sometimes, a thread cannot tell whether the timestamp it used was fresh. To preserve safety in spite of stale timestamps, Safe-Guess’ slow path retries WRITES with a different timestamp only after ensuring that no thread has/will ever read the value using the guessed timestamp.

Each Safe-Guess-replicated object is built on top of two types of reliable wait-free objects: a *max register* as in ABD, and a set of *timestamp locks*. The max register stores the latest written value alongside its timestamp and a flag that indicates whether this timestamp was verified or guessed. The timestamp lock (one per writer) helps threads determine if a value with a guessed timestamp can be read or rewritten.

3.2 Read and Write Algorithms

Fast Safe-Guess WRITES require writers to efficiently guess fresh timestamps in the common case. Guessing is encapsulated in the `guessTs` function (line 5), which returns a pair, the second element of which is the writer’s thread id. Safe-Guess is oblivious to `guessTs`’ implementation but mandates it to be strictly monotonic at a given thread. Assuming reasonable clock synchrony among the application threads, a good timestamp can be guessed using a core’s local clock.

Algorithm 2. Safe-Guess’ WRITE Pseudocode

```

1 M = ((0, ⊥), VERIFIED, ⊥) // Max Register
2 TSL[tid] = {} // Timestamp Lock

4 def WRITE(v):
5   w = (guessTs(), GUESSED, v)
6   in parallel {m = M.READ(), M.WRITE(w)}
7   if m <= w: // Fast path (fresh timestamp)
8     in bg: M.WRITE(w with VERIFIED) // Spdup reads
9   else: // Slow path (potentially stale timestamp)
10    if TSL[tid].TRYLOCK(w.ts, WRITE):
11      M.WRITE(((m.ts.i+1, tid), VERIFIED, v))

```

Algorithm 2 shows the logic of Safe-Guess’ replicated WRITES. The writer synchronizes with the other threads via two reliable shared objects: a max register `M` (line 1) identical to ABD’s (§2.3), and a timestamp lock `TSL[tid]` (line 2) whose details we explain later. The writer starts by preparing the 3-tuple it will write to `M` (line 5). This 3-tuple is composed of (1) the timestamp returned by `guessTs` thanks to which the writer will try to overwrite the max register (tuples are ordered lexicographically) and linearize its WRITE, (2) a `GUESSED` flag that indicates that the writer is not sure about the freshness of the timestamp, and (3) the value `v` it wants to write. Then, in parallel, the writer reads `M` while writing its 3-tuple to it (line 6). Although reading `M` while writing to it (without ordering guarantees) might seem futile, it actually allows the writer to further establish whether the timestamp it guessed was fresh or not. More precisely, if the 3-tuple it reads is less than or equal to the one it wrote (line 7), then it knows that the timestamp it guessed was fresh and that its WRITE was

successfully linearized. In such cases, `WRITE` returns immediately after having scheduled a background task to set the flag in the 3-tuple to `VERIFIED` (which is greater than `GUESSED` with respect to the ordering function used by the max register) to let readers know that its associated timestamp is fresh and definitive (line 8). This scenario lets `WRITE` complete in a single access latency to `M` in the common case.

Otherwise, i.e., if the writer reads a tuple greater than the one it was trying to write (line 9), it cannot assess the freshness of the timestamp it used: it may have been stale, but it may also have been fresh but overwritten before the writer could read it. In the first case, the `WRITE` should be re-executed with a fresh timestamp. In the second, the written value could have been read by another thread and re-executing the `WRITE` with another timestamp would result in `v` being written twice, breaking consistency.

The writer can safely rewrite its value with another timestamp using a *timestamp lock*, a novel wait-free mechanism. Briefly, this lock lets the writer stop concurrent readers from observing the guessed timestamp. Symmetrically, readers can use it to prevent a rewrite if they deem the guess fresh. Similarly to a readers-writer lock, a timestamp lock can either be held by multiple readers, or by the writer, but never both.² Differently, (1) timestamp locks are never unlocked (but can be relocked at higher timestamps); (2) a reader and a writer trying to lock concurrently may both fail; and (3) locking may also fail if a higher timestamp was used. We formally define and construct timestamp locks in Section 3.3.

Safe-Guess uses timestamp locks as follows. A writer whose guessed timestamp may have been stale tries to lock concurrent readers out (line 10). If it fails to, this means that a reader also tried to lock the guessed timestamp, which implies that this reader deemed the timestamp fresh, so `WRITE` can safely return. If the writer manages to lock readers out, none could have returned the value at the guessed timestamp, and any reader attempting to do so in the future would not be able to lock the timestamp, causing it to retry its `READ`. The writer can thus safely re-execute its `WRITE` with a timestamp higher than `m.ts`—so that it is fresh—and mark it as `VERIFIED` (line 11). `WRITE` returns once the `WRITE` to `M` is over. As `WRITE` has a unidirectional control flow and uses only wait-free objects, it is wait-free.

Algorithm 3 shows the logic of Safe-Guess’ READS. Readers use the same max register `M` and timestamp locks `TSL[*]` as writers. Informally, Safe-Guess’ READS consist in iteratively reading tuples from `M` (lines 16–17) until one is deemed valid and its value is returned. If a tuple read from `M` is marked as `VERIFIED`, its value is immediately returned (line 18). This lets READS usually complete in a single access latency to `M`.

Readers, however, cannot wait to find a `VERIFIED` tuple as this would violate both fault tolerance and wait-freedom.

²Our formal specification allows multiple writers to hold the same lock, but we ensure writer exclusion by having one lock per writer.

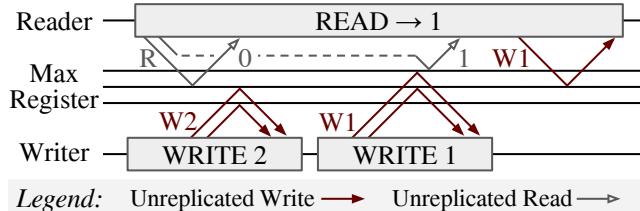


Figure 2. A max register READ reports the wrong maximum out of WRITES concurrent to it. Mismatching majorities lead to a READ of 1, despite 1 being written after 2. However, no subsequent READ can return 1, as WRITE 2 will be over.

They must instead identify, out of all the GUESSED tuples they read from M , one with a value that is safe to return. The condition for a tuple read from M to be safe is two-fold. First, its timestamp must have been fresh (i.e., the writer must have used a timestamp higher than all previously completed WRITES). Second, its writer should not re-execute its WRITE with another timestamp.

Algorithm 3. Safe-Guess' READ Pseudocode

```

12 // Shares M and TSL with writers
13
14 def READ():
15     seen: dict<ThreadId, MValue> = {}
16     while True:
17         m = M.READ()
18         if m is VERIFIED: return m.v // Fast path
19         if m in seen.values(): // Fresh timestamp
20             if TSL[m.ts.tid].TRYLOCK(m.ts, READ):
21                 in bg: M.WRITE(m with VERIFIED) // Spdup rds
22                 return m.v
23             elif m.ts.tid in seen.keys(): // Wait-free path
24                 return seen[m.ts.tid].v
25             seen[m.ts.tid] = m

```

A reader is sure of the freshness of a tuple's timestamp after reading it twice from the max register. Reading it once is not enough, because a READ to a max register can report the wrong maximum out of WRITES concurrent to it (Figure 2). As the second READ is sequential to the first, it disallows such behavior and confirms the freshness of the timestamp. Concretely, the reader tracks the tuples it read in a dictionary (line 15) that it updates at the end of each iteration (line 25), and only considers a tuple as having had a fresh timestamp if it has seen it in a previous iteration (line 19).

After finding a tuple with a fresh timestamp, the reader ensures that its writer did not and will never retry its WRITE with another timestamp by trying to lock the guessed timestamp in read mode (line 20). If this succeeds, the reader deems the tuple valid. The reader thus marks the timestamp as VERIFIED in the background to speed up future READS (line 21), and returns its associated value (line 22).

In case a reader is never able to lock any timestamp, it can still return after seeing two different tuples from the same writer (lines 23–24). Indeed, since this writer started a second WRITE, the first must have completed (either during

Algorithm 4. Timestamp Lock Pseudocode

```

1 CASes = {(\perp, \perp), ...} // 2f+1 CAS Objects
2
3 def TRYLOCK(ts, mode: READ or WRITE):
4     read: dict<CAS, CasValue> = {(\perp, \perp), ...}
5     async for c in CASes:
6         while read[c].0 < ts:
7             expected = read[c]
8             read[c] = c.CAS(expected, (ts, mode))
9             if read[c] == expected: break
10        wait for a majority to complete
11        if any c st read[c].0 > ts: return False
12        if any c st read[c] == (ts, \neg mode): return False
13    return True

```

the READ or just before it), so its value is safe to return.

Thanks to the assumption of a bounded number of writers, this algorithm is forced to terminate in a bounded number of steps. The key observation is that, if a reader fails to lock a timestamp in read mode, then its writer must have seen an even higher timestamp, which ensures that the reader will discover a new tuple in the next iteration. Given that a reader can only loop as long as it does not see two different tuples from the same writer and that there is a bounded number of writers, the reader will return a value in at most $2 \times \#writers + 1$ iterations. Moreover, as all accessed objects are wait-free, so is READ.

3.3 Building Timestamp Locks

Formally, a timestamp lock has a single TRYLOCK(ts , READ or WRITE)→bool method. This method has two properties: (1) TRYLOCK(ts, m) returns True if there are no preceding or concurrent calls to TRYLOCK($ts, \neg m$) (with \neg READ=WRITE) or to TRYLOCK(ts', \star) with $ts' > ts$, and (2) it is impossible for TRYLOCK(ts , READ) and TRYLOCK(ts , WRITE) to both return True. Timestamp locks are similar to conflict detectors [3] and splitters [45], but are more space efficient since a single timestamp lock handles many timestamps.

A wait-free reliable timestamp lock can be built using a set of fallible CAS (compare-and-swap) objects, the majority of which is assumed not to fail (we will deploy a fallible CAS object on each memory node). These CAS objects provide a single CAS(x, y) method, which atomically sets the object's value to y if it was previously equal to x , and returns the previously stored value.

Intuitively, both lock modes race to take control of a majority of the CAS objects, and fail if they hear about the other in the process. The key observation here is that it is impossible for both lock modes to be written at a majority, which ensures that at most one side can return True.

Algorithm 4 details the pseudocode of TRYLOCK. The locker first CASes the tuple ($ts, mode$) to each individual CAS object until a majority of them is associated to a timestamp greater than or equal to ts (lines 5–10). Line 6 ensures that no CAS object ever changes its opinion on the lock mode of a given timestamp, which ensures the safety of the timestamp lock.

If any of the CAS objects contains a timestamp higher than ts , TRYLOCK returns False as TRYLOCK(ts', \star) was called with $ts' > ts$ (line 11). TRYLOCK also returns False if any of the CAS objects contains ts with the opposite lock mode (line 12). Otherwise, TRYLOCK returns True (line 13).

4 In-n-Out

Section 2.3 explains how to build ABD’s and Safe-Guess’ max registers using a set of failure-prone max registers. Implementing these primitive max registers over message passing with compute-capable replicas is simple, but doing so efficiently over disaggregated memory is challenging. The difficulty lies in its limited compute capability and lack of atomic READS and WRITES of large values, that is, values larger than the word size (8B in RDMA [48] and CXLv3 [29]): such operations can co-mingle their execution if executed concurrently. For example, a READ may return only half the data of a concurrent WRITE. This section explains how to overcome these issues using a new technique we call *In-n-Out*.

4.1 Standard Approaches

There are two standard ways of implementing atomic READS and WRITES of large values.

In-place updates with locks. The first technique consists in employing readers–writer locks to ensure that no writer modifies the data while it is being read. This technique has three drawbacks: (1) it does not tolerate thread failures, (2) it requires additional roundtrips to take the lock, and (3) it is not wait-free.

Out-of-place updates with atomic pointer swing. The second technique consists in writing new data to a separate buffer, and then atomically updating an 8B pointer to this buffer. This technique is wait-free but has two performance drawbacks: (1) WRITES must completely write the buffer before updating the pointer (which, depending on ordering guarantees, might require an additional roundtrip), and (2) READS require an extra roundtrip to chase the pointer.

4.2 Overview

In-n-Out combines both in-place and out-of-place updates. It uses in-place updates without any locks so that READS can complete in one roundtrip (no need to chase pointers). However, READS may sometimes find garbled data. In those cases, In-n-Out relies on out-of-place updates so that these READS can return correct data. Importantly, in-place data is validated via a hash that includes a pointer to out-of-place data, so that the former is deemed valid only if it matches the latter. To update the out-of-place buffer and its pointer in a single roundtrip, In-n-Out leverages the ordering guarantees of disaggregated memory (§2).

4.3 Read and Write Algorithms

Figure 3 shows In-n-Out’s memory layout for a given max register. This layout is divided in three segments located on the same memory node: (1) private memory buffers used to

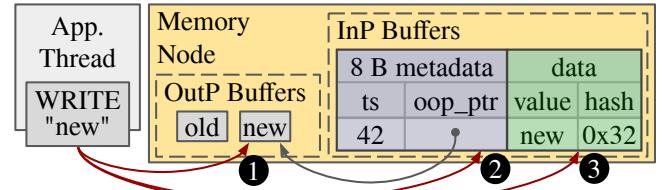


Figure 3. Outline of an In-n-Out WRITE. In one roundtrip, the thread (1) writes to an out-of-place buffer, (2) updates the metadata to point to it, and (3) updates the in-place data.

Algorithm 6. In-n-Out’s READ Pseudocode

```

9 def READ():
10    (metadata, data) = READ(&metadata_data)
11    if hash(metadata, data.value) == data.hash:
12        return data.value
13    return READ(metadata.oop_ptr) // fall back

```

safely store data out of place, (2) 8B of metadata that stores the data’s timestamp and a pointer to its out-of-place buffer, and (3) a buffer for in-place data augmented with a hash. For efficiency, the in-place data buffer is located next to the 8B metadata so that both can be read in a single operation.

Algorithm 5. In-n-Out’s Max WRITE Pseudocode

```

1 def WRITE(v):
2     ts = timestamp(v)
3     oop_ptr = out_of_place_buffers.allocate()
4     pipelined_disaggregated_memory_operations([
5         WRITE(oop_ptr, v),
6         MAX(&metadata, (ts, oop_ptr)))
7     in bg: WRITE(&data, (v, hash((ts, oop_ptr), v))
8     wait for MAX to complete

```

Algorithm 5 shows the pseudocode for In-n-Out WRITES. As it specifically implements ABD’s and Safe-Guess’ large *max* registers, it assumes that the written values include a small timestamp that orders them (Safe-Guess’ GUESSED or VERIFIED flag is encoded in its least significant bit). A WRITE starts by extracting said timestamp using a timestamp function (line 2). Then, the writer allocates a new out-of-place buffer (line 3). Writers pre-allocate large memory chunks so that this buffer allocation does not require accessing disaggregated memory. The writer then issues a series of two memory operations (lines 4–6) that get executed in order thanks to hardware support for FIFO pipelining (§2.1). The first operation fills the just-allocated out-of-place buffer with the value to be written (line 5). The second operation atomically sets the 8B metadata to a pair made of the timestamp and the out-of-place pointer if this is higher than the tuple it previously stored (line 6). The writer then schedules a background update of the in-place buffer with the written value and its hash (line 7). The WRITE is deemed over when the MAX operation completes (line 8).

Algorithm 6 shows the pseudocode for READS. The thread first reads the 8B metadata and the in-place data (line 10). If

the hash of the in-place data is correct (i.e., the in-place data matches the out-of-place data), its value is returned (lines 11–12). Otherwise, the value is read from the out-of-place buffer using the pointer in the metadata (line 13).

4.4 The MAX Operation

In-n-Out’s WRITE procedure (Algorithm 5) uses a MAX read-modify-write atomic operation. While this operation is conceptually similar to a CAS (compare-and-swap), it is typically not supported in hardware. We thus emulate this operation using a CAS (Algorithm 7).

Algorithm 7. CAS-Based MAX Replacement

```

1 def MAX(dst, v):
2     prev = ⊥
3     while prev < v:
4         expected = prev
5         prev = CAS(dst, expected, v)
6         if prev == expected: break
```

This emulation has two drawbacks. First, while a native MAX operation would always complete in one roundtrip, this implementation can retry a (bounded) number of times. Second, this replacement needs a roundtrip to discover the previous value of the 8 B buffer. In practice, as WRITES often follow READS, the value of the 8 B buffer can be cached for MAX to complete in one roundtrip. Still, WRITES to frequently modified keys can suffer from contention.

We reduce contention by replacing the 8 B buffer with an array of 8 B buffers, each updated by a subset of the writers. To find the maximum, readers scan the array and return the largest entry. While the scan is not atomic, this technique provides the required semantics of a max register: READS return a value greater than or equal to all operations that completed before them. Moreover, by using one 8 B buffer per writer, MAXes can be made to complete in one roundtrip. We evaluate the benefits of this technique (§7.9).

4.5 Recycling Memory

Each In-n-Out WRITE allocates a new out-of-place buffer (Algorithm 5). These buffers eventually need to be recycled with care, lest readers can obtain invalid data due to concurrent (unexpected) WRITES. One way to recycle is to use wait-free hazard pointers [1]: a reader advertises its wish to read data before accessing its out-of-place buffer, so that if a writer were to recycle its memory, it would first satisfy the reader’s wish by handing it a valid buffer, and only recycle unrequested buffers. This technique preserves wait-freedom, but requires an extra roundtrip to advertise READS. We use a different approach, which does not require the extra roundtrip: to recycle, a thread first asks all readers to stop accessing the buffers to be freed. In theory, this approach is not wait-free because a writer that runs out of memory must wait for slow readers to respond to its recycling request. In practice, this is a sensible trade-off as real-life systems are partially synchronous and use membership services [22, 28], so they can

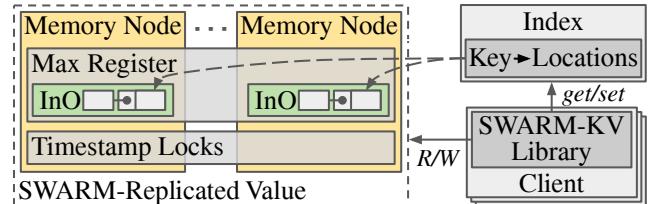


Figure 4. Architecture of SWARM-KV with a single key. Clients use SWARM READS and WRITES to directly access the value replicated in disaggregated memory. The replicas that form Safe-Guess’ max register are spread across different memory nodes and implemented via In-n-Out max registers. The location of the replicas is stored in a reliable index.

recycle memory in background tasks with bounded delays, so that the read-write protocol never runs out of memory and thereby remains wait-free.

5 SWARM-KV

We use SWARM to build SWARM-KV, the first fault-tolerant disaggregated key-value store to provide single-roundtrip GETS, UPDATES, INSERTS and DELETES in the common case. Section 5.1 establishes the setting, Section 5.2 gives the outline of SWARM-KV, Section 5.3 explains the protocol for each operation, and Section 5.4 discusses memory recycling.

5.1 Setting

SWARM-KV clients run on traditional data-center servers that directly INSERT, UPDATE, GET and DELETE key-value pairs replicated in disaggregated memory using the SWARM-KV library. There are no intermediary servers between the clients and the data in disaggregated memory, which minimizes the latency. Any number of SWARM-KV clients can crash, along with any set of memory nodes, provided each key has a majority of its replicas accessible. SWARM-KV also requires a fast index as well as a membership service, which can run on traditional servers and are fault-tolerant. While SWARM-KV is safe under partitions, clients might temporarily lose liveness if they are unable to contact a majority of disaggregated replicas, the index, or the membership service.

5.2 Overview

Figure 4 shows SWARM-KV’s architecture. SWARM-KV’s clients directly access key-value pairs replicated over a set of memory nodes using SWARM. Each replica is a max register implemented via In-n-Out. An index tracks the locations of keys’ replicas (i.e., their memory nodes and addresses on them). SWARM-KV is oblivious to the choice of index, as long as it is reliable and allows clients to set and get the replicas associated to a key in a single roundtrip in the common case. When a client accesses a key-value pair for the first time, it finds the location of its replicas via the index and caches it for subsequent operations. This location data is small (≈ 24 B per key), and is the only significant source of memory consumption for clients, so clients can cache tens of millions of keys.

5.3 Protocols

We now give the protocols for `INSERT` (§5.3.1), `DELETE` (§5.3.2), `UPDATE` (§5.3.3), and `GET` (§5.3.4) SWARM-KV operations.

5.3.1 INSERTS. The client executes two things in parallel: it replicates the value over a set of memory nodes, and it inserts the replicas’ location in the index. More precisely, the client first picks a set of memory nodes and allocates on each an In-n-Out max-register replica (a buffer to fit In-n-Out’s 8 B metadata and in-place data, §4). The client then uses a SWARM `WRITE` to replicate the inserted value over the chosen replicas. Importantly, upon the replicas’ allocation, their In-n-Out 8 B metadata is cleared to indicate that they are empty. Clients pre-allocate cleared buffers so that this step completes in one roundtrip.

In parallel to this `WRITE`, the client asks the index to map the inserted key to the chosen replicas. If a mapping to replicas marked for deletion already exists (§5.3.2), it is overwritten. Otherwise, if a mapping to writable replicas already exists, the client recycles the In-n-Out buffers it just allocated, and the `INSERT` request turns into an `UPDATE` (§5.3.3) that uses the existing replicas. The location of the replicas is then cached so that future `GET` and `UPDATE` operations on this key can bypass the index.

The `INSERT` is over once both tasks complete, which parallelization allows for in one roundtrip in the common case. It is fine for the index insertion to finish before the SWARM `WRITE`, as concurrent readers would find empty In-n-Out replicas, indicating that the key does not exist. Similarly, client failure may leave an index entry pointing to empty replicas, but this does not affect correctness.

5.3.2 DELETES. The client consults its cache to find the replicas of a key; if not found, it consults the index. Then, it runs a special SWARM `WRITE` that uses the max timestamp (i.e., all bits set) to ensure that it cannot be overwritten and that future SWARM `READS` and `WRITES` will fail until the deleted key is re-inserted. This way, the clients that may have cached replicas for the deleted key will see it deleted. Once the special SWARM `WRITE` is over, the `DELETE` completes and the client simply schedules a background task to unmap the just-written replicas from the key.

5.3.3 UPDATES. The client finds the key’s replicas using its cache or the index. Then, it issues a SWARM `WRITE` to replicate its value and return upon its successful completion. If the key is not indexed, the `UPDATE` fails. If the replicas were cached but the `WRITE` fails due to a previous `DELETE`, the client flushes its cache, deletes the old mapping in the index (in case the deleter failed), and retries the `UPDATE`.

5.3.4 GETS. The client locates the key’s replicas and then issues a SWARM `READ`. If the key is not (fully) inserted, the `GET` fails. Similarly to `UPDATES`, if the `READ` fails due to a `DELETE`, the client flushes its cache and retries the `GET`.

5.4 Recycling Memory

In the background, clients check if the in-place buffers they allocated are still indexed, and if their out-of-place buffers are still accessible. To help with this phase, clients who delete or update a key inform its allocators that their buffers should be recycled. Before a client recycles a buffer, it ensures other clients no longer access it by asking them to remove the key from their index cache and to stop accessing to-be-recycled out-of-place buffers. Failed clients may not reply to these requests. We use uKharon [22], a fast membership manager, to monitor the health of clients and instruct memory nodes to disconnect from suspected clients so they do not access freed memory. Some memory nodes may be inaccessible and not do so, but, for each key, a majority will be able to recycle memory, which is enough to ensure the liveness of SWARM-KV.

6 Implementation

Our implementation of SWARM-KV targets RDMA-based disaggregated memory using uKharon’s framework [22], and consists of 3,151 lines of C++17 (CLOC [12]). It uses xxHash3 [9] for In-n-Out’s hashes and FUSEE’s index [47] that we modify to provide strong consistency. Our clients guess timestamps via a loosely synchronized TSC-based clock [23] that they re-synchronize every time they guess a stale timestamp. We did not implement memory recycling (§4.5 and §5.4).

To optimize bandwidth, memory, and latency, our implementation slightly diverges from the pseudocode previously given. First, Safe-Guess optimistically tries to replicate its `WRITES` to a mere majority of the memory nodes determined by hashing the keys to spread the load. If one of the memory nodes does not respond immediately, Safe-Guess then tries to replicate to all replicas. Similarly, Safe-Guess `READS` optimistically read from a mere majority of the memory nodes. Second, In-n-Out’s in-place data is only stored at one of the replicas—also determined by hashing of the key—and only when marking a Safe-Guess tuple as verified (§3.2).

7 Evaluation

We evaluate SWARM through SWARM-KV with the goal of answering the following questions:

- What is SWARM-KV’s latency under YCSB (§7.1)?
- How does throughput impact latency (§7.2)?
- How does SWARM-KV scale with clients (§7.3)?
- How do different value sizes affect SWARM-KV (§7.4)?
- What is the impact of the replication factor (§7.5)?
- What is SWARM-KV’s resource consumption (§7.6)?
- What is the impact of failures on availability (§7.7)?
- What is the impact of extreme contention (§7.8)?
- How does our MAX substitute handle contention (§7.9)?

Testbed. Our testbed is a cluster with 4 servers and 4 memory nodes configured per Table 1. The dual-socket machines

have an RDMA NIC attached to the first socket. Our experiments execute on cores of the first socket using local NUMA memory. We measure time using TSC [23] via `clock_gettime` with the `CLOCK_MONOTONIC` parameter. All machines are connected to a single switch, but we expect the latency benefits of SWARM-KV to increase in settings with more hops.

Baselines. We compare SWARM-KV against three baselines. The first is an unreplicated disaggregated key-value store that provides no consistency under concurrent operations, which we call *RAW*. Such a system is not useful in practice, but it is useful to establish a lower bound on latency.

The second baseline is a disaggregated key-value store replicated via ABD, we call *DM-ABD*, that supports concurrency via out-of-place updates. It represents a good engineering solution using known techniques. DM-ABD’s GETs and UPDATES commonly finish in two roundtrips: while GETs require an extra indirection, UPDATES hide latency by writing out-of-place data in parallel to finding a fresh timestamp.

The third baseline is FUSEE, a state-of-the-art disaggregated key-value store that uses a synchronous replication scheme. FUSEE uses out-of-place updates that take at least four roundtrips. FUSEE caches UPDATES’ location so that GETs run in one roundtrip if they re-access a key before it is updated, but take two roundtrips otherwise. Due to FUSEE’s stronger synchrony assumptions and higher latency, we limit the comparison with it to end-to-end latency (§7.1).

Table 2 shows the number of roundtrips for GETs and UPDATES of all systems, for both the common case and the 99th percentile, as observed under a standard workload (§7.1). In all systems, operations are so fast that same-key contention is rare. Due to their use of a synchronous index, all systems have a theoretical unbounded latency in the worst case [17].

Workloads. We run YCSB [11] workloads A (50% GETs and 50% UPDATES) and B (95% GETs and 5% UPDATES) with Zipfian (.99) key distribution. Unless stated otherwise, we

Table 1. Configuration details of machines.

4 Client Servers			
CPU	2× 8c/16t Intel Xeon Gold 6244 @ 3.60 GHz		
NIC	Mellanox CX-6 MT28908		
4 Memory Nodes			
Memory	4× SK Hynix 32GB DDR4-2400 RDIMM @ 2133 MT/s		
NIC	Mellanox CX-4 MT27700		
Switch	MSB7700 EDR 100 Gbps		
Software	Linux 5.15.0-102-generic / Mlnx OFED 5.3-1.0.0.1		

Table 2. Number of roundtrips for GETs and UPDATES.

	Common		99th percentile	
	GET	UPDATE	GET	UPDATE
RAW	1	1	1	1
SWARM-KV	1	1	1	1
DM-ABD	2	2	2	2
FUSEE	1–2	4	2	5

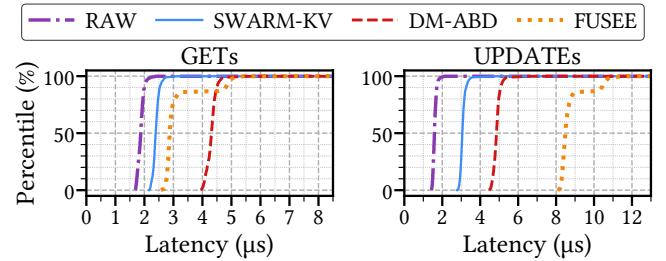


Figure 5. Latency CDFs of RAW, SWARM-KV, DM-ABD and FUSEE with YCSB workload B (95% GETs and 5% UPDATES), Zipfian key distribution, and 4 clients.

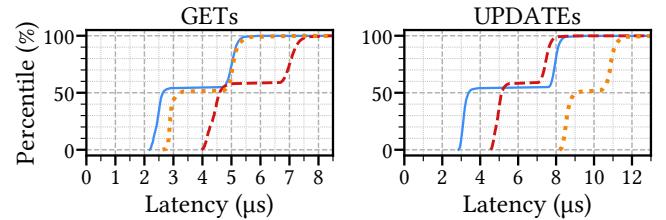


Figure 6. Experiment of Figure 5, but with 1M keys which do not all fit in index caches, and excluding RAW.

use 3 replicas, 100K keys of 24 B, 64 B values, 4 clients that issue one operation at a time, and index caches large enough to cache all key locations. There is a warm-up phase of 1M operations and we measure the following 1M operations.

7.1 Latency

We evaluate the latency of all four systems with 4 clients, each issuing 1 operation at a time. Figure 5 shows the results for YCSB workload B with Zipfian key distribution.

For GETs, RAW has a median latency of 1.9 μs. SWARM-KV has a median latency of 2.4 μs—a mere 27% increase over RAW—, and a very low spread (P1=2.2 μs and P99=2.8 μs). FUSEE has a bimodal distribution: 87% of operations run in one roundtrip in 2.9 μs (+53% vs RAW), while keys recently modified by other clients require accessing the index to obtain the new location, which takes a second roundtrip, reaching 4.8 μs at the 90th percentile (+157% vs RAW). DM-ABD comes last with a median latency of 4.3 μs (+130% vs RAW) due to its use of pure out-of-place updates.

For UPDATES, RAW has a median latency of 1.6 μs. SWARM-KV runs in one roundtrip with a median latency of 3.1 μs, a 92% increase over RAW. DM-ABD always runs in two roundtrips, with a median of 4.9 μs (+206% vs RAW). FUSEE comes last with a bimodal distribution: rarely modified keys take 4 roundtrips in 8.5 μs (+431% vs RAW), while hot keys take 5 in 10.4 μs at the 90th percentile (+554% vs RAW).

Under YCSB workload A (not shown), all systems perform similarly, except FUSEE whose switch to slower operations begins much earlier at the 41st percentile. Overall, SWARM-KV adds less than 1 RTT of overhead over RAW, and has substantially lower overhead than other replicated solutions.

We run a similar experiment for 1M keys with clients

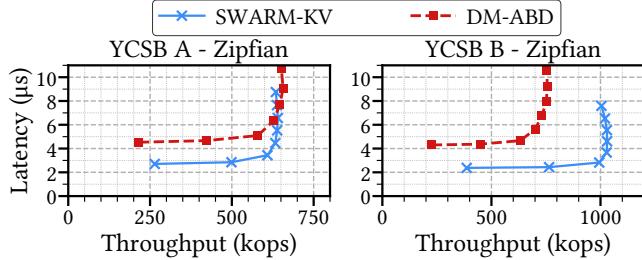


Figure 7. Per-core throughput-latency graph of SWARM-KV and DM-ABD with YCSB workloads A and B, Zipfian key distribution, and 4 clients. Each marker is a measurement for a given number of concurrent operations from 1 to 8.

using small index caches, and observe how the latency distribution changes across SWARM-KV, DM-ABD and FUSEE (RAW, which merely established a lower bound on latency, is omitted). We limit the cache size of each system to 5 MiB of system-specific metadata. More precisely, in DM-ABD and FUSEE, cache entries have 24 B, while in SWARM-KV, they have 32 B since they also include In-n-Out’s metadata field (see Figure 3). As a result, DM-ABD and FUSEE can cache more keys (21.8% of them), compared to SWARM-KV (16.4% of them). This accounting does not include metadata for the cache replacement policy, which is the same for all systems (≈ 32 B per entry for an approximation of LFU).³ exhibit a bimodal distribution. On cache hits, the latency distribution follows that of Figure 5. Cache misses add 1 extra roundtrip to read the index on all systems, except for UPDATES on SWARM-KV where 2 additional roundtrips are required: one to read the index, and one to read the latest metadata buffer. With such small caches, we measure that DM-ABD and FUSEE have a miss rate of 42.5%, but, similarly to Figure 5, 11% of FUSEE’s cache hits occur on stale pointers due to recent modifications by other clients. Thus, FUSEE needs to access the index 48.8% of the time. While SWARM-KV caches 25% fewer keys, the combination of a Zipfian distribution with an LFU replacement policy ensures that the remaining 75% are likely to be the hottest, so SWARM-KV’s miss rate only increases to 45.6%. In this configuration, SWARM-KV’s average latency remains better than DM-ABD’s and FUSEE’s for both types of operations. Upon a cache miss, SWARM-KV’s and DM-ABD’s UPDATE algorithms are equivalent, but the former performs slightly worse as it fetches multiple metadata buffers (§4.4).

7.2 Per-Core Throughput and Latency under Load

We study SWARM-KV’s per-core throughput and latency under load. Figure 7 shows the results with each single-threaded

³Had we accounted for this extra overhead, the difference in cache coverage between DM-ABD/FUSEE and SWARM-KV would be smaller, thus benefiting SWARM-KV. We extend the warm-up to 8M operations to stabilize the cache policy.

Figure 6 shows the results. With this configuration, all systems

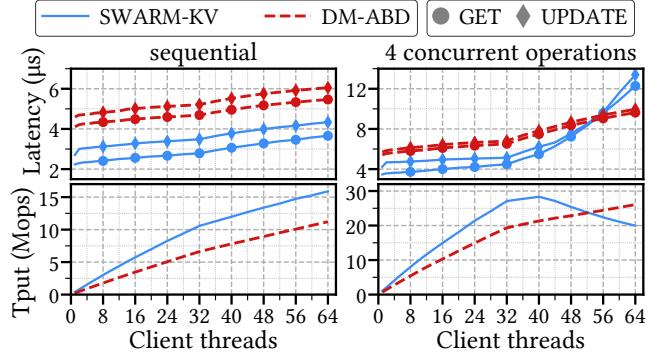


Figure 8. Throughput and average latency of SWARM-KV and DM-ABD with YCSB workload B, Zipfian key distribution, when varying the number of clients and concurrency.

client running from 1 to 8 concurrent operations and DM-ABD as a reference, for YCSB workloads A and B with Zipfian key distribution.

Both workloads show large gains in throughput and low impact on latency with 3 concurrent operations, before reaching a throughput-latency wall as the CPU is bottlenecked by the submission of RDMA requests. With YCSB A and 1 operation at a time, SWARM-KV’s average latency is 2.7 μs with a throughput of 264 kops. Running a second operation has little impact on latency (+5%), which reaches 2.8 μs, while nearly doubling the throughput at 499 kops. Running a third operation increases the latency by 21%, reaching 3.4 μs, but the throughput only increases by 22% at 609 kops. Additional operations increase the latency by 1 μs each; a maximum throughput of 640 kops is reached with 6 concurrent operations. DM-ABD reaches a similar maximum throughput.

YCSB B has fewer UPDATES, so SWARM-KV’s average latency starts at a lower 2.4 μs with a throughput of 389 kops. Similarly to YCSB A, having more than 3 concurrent operations yields negligible throughput enhancement while increasing latency by nearly 1 μs with each additional operation. A maximum throughput of 1030 kops is achieved with 5 concurrent operations. SWARM-KV outperforms DM-ABD.

This limited throughput gain is due to SWARM-KV’s lack of batching and the fixed cost of issuing a series of RDMA operations to a memory node. As the latter takes 200+ ns and each GET/UPDATE reaches 2 memory nodes, a core can send 1 additional operation while waiting for the first to complete, but issuing a second delays the processing of the first.

7.3 Scalability

We evaluate the scalability of SWARM-KV by running the workloads with a number of single-threaded clients ranging from 1 to 64 while measuring the latency of GETs and UPDATES, and the total throughput. Figure 8 shows the results for YCSB workload B with Zipfian key distribution when clients try to optimize latency (i.e., with a single operation at a time) and when clients try to optimize throughput (i.e., with 4 concurrent operations). We plot DM-ABD for reference.

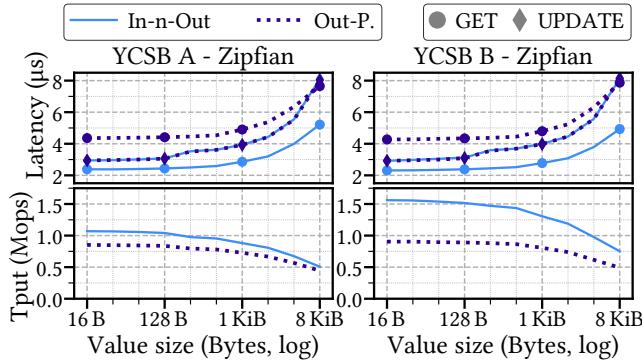


Figure 9. Throughput and average latency of SWARM-KV with YCSB workloads A and B, and varying value sizes. Comparison with a SWARM-KV variant without in-place updates.

With 1 operation at a time, SWARM-KV’s throughput scales nearly linearly with the number of clients, reaching a maximum of 15.9 Mops with 64 clients. After 32 clients, the throughput grows slower due to hyper-threading. The latency of GETs moderately increases from 2.2 μs with a single client to 3.69 μs with 64, with virtually all GETs completing in a single roundtrip. Similarly, the latency of UPDATES, increases from 2.7 μs to 4.3 μs with 64 clients. Both GETs and UPDATES have much lower latency than DM-ABD’s.

With 4 concurrent operations, the throughput of SWARM-KV scales nearly linearly until 40 clients where it reaches a peak of 28.3 Mops with 5.5 μs GETs, and 6.22 μs UPDATES. After this point, SWARM-KV bottlenecks our 100 Gbps fabric, which leads to a fast increase in latency—underperforming DM-ABD at 56 clients—and a drop in throughput.

SWARM-KV’s good scalability comes from its full bypass of the CPUs of memory nodes, the contention-handling mechanism of its MAX substitute (§4.4), and its efficient guessing of fresh timestamps using loosely synchronized clocks.

7.4 Impact of Value Size

We study the effect of different value sizes ranging from 16 B to 8 KiB on SWARM-KV’s latency and throughput. Suspecting that large values might perform poorly due to In-n-Out’s combination of in-place and out-of-place updates, we run the same experiments with a version of SWARM-KV that does not use in-place updates. Figure 9 shows the results for YCSB workloads A and B under Zipfian key distribution.

Regardless of the workload, the latency of SWARM-KV operations grows linearly with the increase in value size and values up to 8 KiB still achieve single-digit latency.

One could expect In-n-Out’s combination of in-place and out-of-place updates to be detrimental to the latency of large keys: it is not. Our evaluation shows that, regardless of the YCSB workload and the value size: (1) GETs always benefit from in-place data, with large 8 KiB values still being 33% faster (by 2.5 μs), (2) UPDATES comprising in-place updates are just as fast as purely out-of-place ones thanks to the lazy writing of in-place data (§6), and (3) In-n-Out’s combination

Table 3. Resource consumption, for 1M keys, values of 1 KiB, YCSB workload B, and 4 clients each executing 200 kops.

	Per single-threaded client			Disagg. Mem.
	CPU	Cache	IO BW	
RAW	46.6%	22.9 MiB	6.55 Gbps	0.95 GiB
DM-ABD	99.0%	22.9 MiB	6.99 Gbps	3.00 GiB
SWARM-KV	61.3%	30.5 MiB	7.41 Gbps	4.06 GiB
FUSEE	74.2%	22.9 MiB	8.15 Gbps	2.04 GiB

leads to higher total throughput, especially for read-intensive workloads (+50% at 8 KiB for YCSB workload B).

7.5 Impact of Replication Factor

We study the impact of the replication factor on SWARM-KV’s latency and throughput by using 3, 5, or 7 replicas per key. Due to our setup having only 4 memory nodes, we sometimes store 2 replicas on the same node. Figure 10 shows the results for YCSB workload B with Zipfian key distribution and DM-ABD for reference. Both systems replicate to a mere majority of the replicas (e.g., with 3 replicas, both write to only 2 replicas to tolerate 1 failure) (§6).

Both SWARM-KV and DM-ABD see their latency increase linearly with the number of replicas. SWARM-KV starts with a median latency of 2.3 μs for GETs and 3 μs for UPDATES with 3 replicas. Each 2 additional replicas increase the latency of GETs by 0.2 μs and UPDATES by 0.5 μs, due to the cost of issuing an additional series of RDMA operations. Thus, SWARM-KV’s throughput drops by 9% moving from 3 to 5 replicas, and by an additional 7% moving from 5 to 7. DM-ABD shows similar latency increase, but starts at 4.3 μs for GETs and 4.7 μs for UPDATES. For both systems, the spread in latency between the 1st and the 99th percentiles is stable, increasing by a mere 10% with each additional 2 replicas.

7.6 Resource Consumption

Table 3 shows the computed resource consumption of the four systems when tolerating 1 failure, storing 1M keys with values of size 1 KiB, with 4 clients executing 200 kops each under YCSB workload B, and running garbage collection (§4.5 and §5.4) once per second.

The cost of SWARM-KV’s low latency is a higher use of disaggregated memory. RAW, due to its lack of replication, has

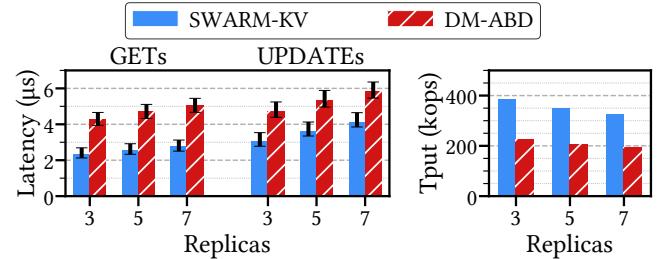


Figure 10. Median latency and per-client throughput of SWARM-KV and DM-ABD with YCSB workload B, Zipfian key distribution and different numbers of replicas. Whiskers depict the 1st and the 99th percentiles.

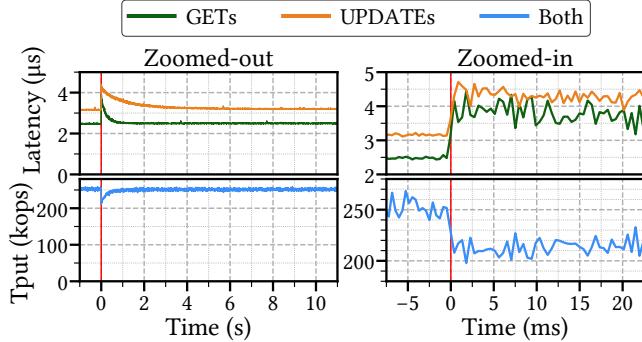


Figure 11. Latency and throughput of a SWARM-KV client with YCSB workload A before and after the failure (at $t=0$) of a memory node. Zoomed-out on the left to see performance recovery. Zoomed-in on the right to see the transition.

minimal resource use. DM-ABD uses 3 replicas per key plus additional metadata, so its consumption of disaggregated memory is $3.1\times$ higher than RAW’s. However, as DM-ABD UPDATES replicate values to only 2 replicas and GETs retrieve the full values from only 1, DM-ABD’s bandwidth use is only 7% higher than RAW’s. SWARM-KV uses 35% more disaggregated memory and 6% more bandwidth than DM-ABD due to In-n-Out and its contention-reduction mechanism (§4.4). Moreover, SWARM-KV’s clients consume 33% more memory than DM-ABD’s due to their need to cache In-n-Out’s 8 B metadata. Thanks to its synchronous replication protocol, FUSEE requires only 2 replicas to tolerate 1 failure, which makes it consume 32% and 50% less disaggregated memory than DM-ABD and SWARM-KV, respectively. However, FUSEE’s optimistic GETs fail 13% of the time, wasting bandwidth, so FUSEE consumes 16% and 10% more IO than DM-ABD and SWARM-KV, respectively.

Thanks to its low latency, SWARM-KV does little polling, so its CPU use is only 30% higher than RAW’s, while being 38% lower than DM-ABD’s, and 16% lower than FUSEE’s.

7.7 Memory Node Failure

We study the impact of the failure of a memory node on SWARM-KV’s performance and availability by crashing a memory node while measuring SWARM-KV’s throughput and latency. Figure 11 shows the results.

The failure of a memory node has no impact on the availability of SWARM-KV, but it temporarily affects its latency. This impact results from SWARM-KV’s bandwidth-reduction mechanism which saves bandwidth in the absence of failure (§6). Upon a failure, ongoing SWARM-KV operations merely contact additional memory nodes as the majority they initially contacted takes longer than expected to respond. Importantly, this change of replicas requires no reconfiguration of the system, so it has negligible impact. SWARM-KV’s latency increases after a failure due to the loss of in-place data, and the lack of unanimity across the remaining replicas. Both are eventually rebuilt by subsequent operations, which allows operations to recover their latency.

In contrast, synchronous systems like FUSEE reportedly take at least tens of milliseconds to recover from failures [28, 47]. This is because they need to accurately detect failures before initiating recovery procedures involving multiple phases such as scanning logs, transferring state, and changing roles.

7.8 Extreme Contention

We evaluate the impact of extreme contention on SWARM-KV’s performance by studying the latency of GETs and UPDATES when either 16 or 32 clients hit a single key. Figure 12 shows the results under YCSB workload A with 16 clients.

With 16 clients, GETs remain live but the latency of their 99th percentile degrades to 30 μ s. We see that only 14% of GETs complete in 1 roundtrip by finding a correct in-place value. 8% of GETs complete in 2 roundtrips by reading the value out of place. The next 78% of GETs, however, complete in more than 2 roundtrips as they must either iterate a few times before identifying a valid value, or reading Safe-Guess’ max register requires a writing step (§2.3). The latency of DM-ABD’s GETs also significantly increases because of the latter, but it is drastically exacerbated by the contention on the MAX substitute, which lacks SWARM-KV’s contention-reduction mechanism (§4.4).

Meanwhile, UPDATES complete in at most 4 roundtrips, with their 99th percentile reaching 10 μ s. 73% of UPDATES complete in 1 roundtrip by guessing a fresh timestamp and identifying it as so immediately. 7% of UPDATES complete in 2 roundtrips by guessing a fresh timestamp and using their timestamp lock to detect it as fresh. 14% of UPDATES complete in 3 roundtrips by also having to either (a) replicate the value of another writer to a majority or (b) retry their WRITE with a fresh timestamp. 6% of UPDATES complete in 4 roundtrips by having to do both (a) and (b). Similarly to GETs, the latency of DM-ABD UPDATES significantly degrades due to high contention on the unreliable max registers.

With 32 clients (not shown), SWARM-KV’s GETs see their median latency degrade by 4 μ s while UPDATES are mostly unaffected. DM-ABD, however, is severely impacted as the median latency of its GETs and UPDATES both reach 47 μ s.

7.9 Scalability of our MAX Substitute

We now evaluate how varying the number of In-n-Out 8 B metadata buffers (§4.4) affects the latency of SWARM-KV by running YCSB workloads A and B with 64 clients. Figure 13

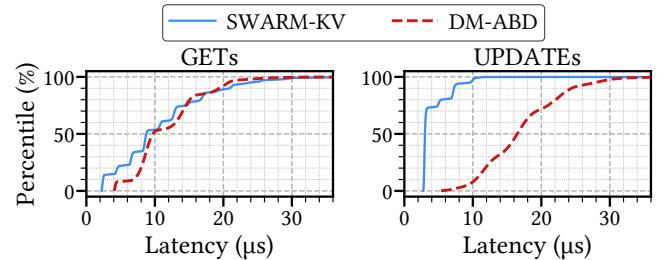


Figure 12. Latency CDFs for SWARM-KV and DM-ABD with YCSB workload A, a single key-value pair, and 16 clients.

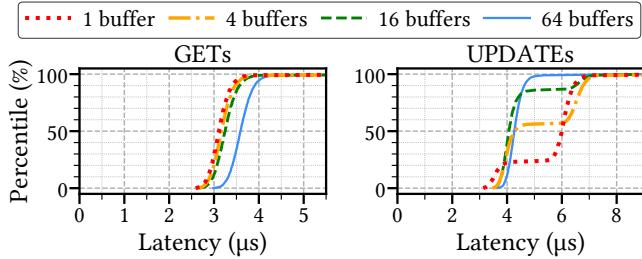


Figure 13. Latency CDFs for SWARM-KV with YCSB workload B, 64 clients, and a varying number of metadata buffers.

shows the results for workload B.

The use of multiple metadata buffers per key significantly reduces the latency of **UPDATES** but slightly increases the latency of **GETs**. With a single metadata buffer per key that is shared among all writers, only 23% of **UPDATES** complete in one roundtrip by having accessed the key since its last **UPDATE**. With 4 buffers per key, each shared by 16 clients, merely 57% of **UPDATES** complete in one roundtrip. This raises to 86% with 16 buffers per key. Lastly, with one metadata buffer per key and per writer, 99% of **UPDATES** finish in 1 roundtrip, the last percent resulting from concurrency or from the guess of stale timestamps.

In the meantime, increasing the number of buffers increases the latency of **GETs** and single-roundtrip **UPDATES** because they must read larger arrays of buffers. With 1 buffer, **GETs** and single-roundtrip **UPDATES** have a median latency of 3.1 μ s and 3.6 μ s, respectively. These latencies grow to 3.2 μ s and 3.9 μ s with 4 buffers, 3.3 μ s and 4 μ s with 16 buffers, and 3.6 μ s and 4.3 μ s with 64 buffers.

Under YCSB workload A (not shown) **GETs** have the same latency, but only 2% of **UPDATES** complete in one roundtrip with 1 buffer, 11% with 4, 39% with 16, and 99% with 64. We thus recommend using 1 buffer per client as it has low impact on the latency of 1-roundtrip operations (+15% for 64 clients), but makes 1-roundtrip **UPDATES** much more frequent. However, this means that the more clients there are, the larger the arrays of metadata buffers become, which hinders performance and limits scalability.

8 Related Work

Our work touches on various topics: replication, concurrency control, key-value stores and disaggregated memory.

Replication. Linearizable READ-WRITE replication has been the subject of active research. The seminal ABD [4] protocol is the first of a long line of protocols [14, 16, 27, 50]. Dutta *et al.* [14] prove that one cannot always have one-roundtrip READS or WRITES, so several researchers [19, 20, 33] investigated semi-fast implementations, where WRITES take one roundtrip, but READS take longer.

Similarly to SWARM, Gus [50] achieves single-roundtrip latency for both operations in the common case. However, Gus fundamentally differs from SWARM in the extra assumptions it makes and the techniques it uses. Specifically: (1)

Gus assumes that replicas are regular machines capable of running arbitrary RPCs; in contrast, we consider memory replicas that are operated on by one-sided RDMA, which is a challenge we address. (2) Gus requires each client to be co-hosted with one of the replicas for low latency; in contrast, our clients are hosts remotely connected to memory nodes, which makes the problem harder. (3) Gus requires more than $2f+1$ replicas to tolerate f failures when $f > 2$; in contrast, we require only $2f+1$ replicas. Gus, however, has a theoretical advantage over our proposal: even in the worst case, READS and WRITES complete in at most 2 RTTs (while SWARM/Safe-Guess may take a slow path in rare cases).

Various consensus-based protocols have been proposed to achieve strongly consistent memory replication. EPaxos [46], Giza [8], and TEMPO [15] are recent examples of such protocols. While these protocols can replicate any data object, SWARM considers only READ/WRITE operations. However, consensus-based replication protocols typically have a slower fast path than SWARM (i.e., requiring more roundtrips), and incur long delays when synchrony assumptions do not hold (e.g., when timeouts are exceeded). Gryff [7] is an interesting consensus-based system which combines consensus and ABD: it provides READ/WRITE operations without utilizing consensus, and resorts to consensus for other operations. Although Gryff matches SWARM’s single-roundtrip READS, it requires two roundtrips for WRITES. Importantly, none of the aforementioned systems considers the disaggregated memory setting where replicas sit on nodes without general compute capability. Carbink [52] implements reliable disaggregated memory, but considering a non-shared access: only one client accesses a given memory location.

Concurrency control. Some of SWARM’s techniques to achieve single-roundtrip READ/WRITE latency are inspired from other systems. Utilizing real-time timestamps for optimistic ordering appears in NCC [41] to efficiently order transactions. Self-validating READS appear in both Pilaf [44] and FARM [13] to avoid inconsistencies stemming from RDMA’s weak READ guarantees. Out-of-place updates appear in Clover [49], which uses them to achieve concurrent lock-free access to disaggregated memory. Sherman [51] leverages the FIFO guarantee of RDMA to minimize memory roundtrips when modifying a lock-protected tree structure.

Disaggregated KVS. Several recent key-value stores have been designed for disaggregated memory, e.g., DINOMO [38], AsymNVM [43], Clover [49], ROLEX [40], and FUSEE [47]. These works differ from SWARM-KV in various ways. DINOMO requires clients to communicate with intermediary key-value servers instead of directly accessing memory nodes, which requires at least two roundtrips to update data. DINOMO, AsymNVM, and ROLEX make different assumptions from us: they require memory nodes to have some compute to offload custom functionality of the key-value store. DINOMO, AsymNVM, and Clover have a different goal

from us: they aim to provide durability, which require persistent memory and different techniques. Moreover, Clover is not designed for scalability since it uses a metadata server in the control plane, which becomes a bottleneck. Meanwhile, ROLEX focuses on improving the index performance through learning. The closest related work to SWARM-KV is FUSEE, which allows clients to access disaggregated memory directly, requires no compute at memory nodes, provides scalability, and targets volatile memory. FUSEE heavily relies on caching to keep latency low, but its GETs often finish in 2 roundtrips, and its UPDATES take at least 4 roundtrips, resulting in lower performance, as shown by our evaluation.

9 Concluding Remarks

SWARM is the first replication scheme for in-disaggregated-memory shared objects to provide (1) single-roundtrip READS and WRITES in the common case, (2) strong consistency (linearizability), and (3) strong liveness (wait-freedom). SWARM makes two independent contributions. The first, Safe-Guess, is the first such protocol assuming memory nodes can provide atomic conditional updates to large objects in a single roundtrip. The second, In-n-Out, is a novel technique to obtain the atomic conditional update of large buffers in one roundtrip without requiring compute at memory nodes. We showed the utility of SWARM by building SWARM-KV, a low-latency, strongly consistent and highly available disaggregated key-value store that significantly outperforms the state of the art and incurs little replication latency overhead.

We foresee several extensions to our work. While SWARM considers in-memory replication, adding durability without harming performance is appealing, but appears non-trivial [18, 37]. Also, formally proving SWARM’s algorithms, but also its implementation (along the lines of IronFleet [24]), is interesting future work.

Acknowledgments

We thank the anonymous reviewers and our shepherd Wyatt Lloyd for their valuable comments, as well as the anonymous artifact evaluators for reviewing our implementation. We also thank Thomas Bourgeat for his feedback.

References

- [1] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual Event) (PLDI ’21). Association for Computing Machinery, New York, NY, USA, 526–541. <https://doi.org/10.1145/3453483.3454060>
- [2] James Aspnes, Hagit Attiya, and Keren Censor. 2009. Max registers, counters, and monotone circuits. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada) (PODC ’09). Association for Computing Machinery, New York, NY, USA, 36–45. <https://doi.org/10.1145/1582716.1582728>
- [3] James Aspnes and Faith Ellen. 2014. Tight Bounds for Adopt-Commit Objects. *Theory of Computing Systems* 55, 3 (Oct. 2014), 451–474. <https://doi.org/10.1007/s00224-013-9448-1>
- [4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (Jan. 1995), 124–142. <https://doi.org/10.1145/200836.200869>
- [5] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley and Sons, Inc., Hoboken, NJ, USA.
- [6] Motti Beck and Michael Kagan. 2011. Performance Evaluation of the RDMA over Ethernet (RoCE) Standard in Enterprise Data Centers Infrastructure. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching* (San Francisco, CA, USA) (DC-CaVES ’11). International Teletraffic Congress, San Francisco, CA, USA, 9–15. <https://dl.acm.org/doi/10.5555/2043535.2043537>
- [7] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) (NSDI ’20). USENIX Association, Berkeley, CA, USA, 591–617. <https://www.usenix.org/conference/nsdi20/presentation/burke>
- [8] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA, USA) (USENIX ATC ’17). USENIX Association, Berkeley, CA, USA, 539–551. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin>
- [9] Yann Collet. 2022. xxHash: Extremely fast non-cryptographic hash algorithm. <https://github.com/Cyan4973/xxHash> Accessed 2024-03-17.
- [10] Compute Express Link Consortium. 2022. Compute Express Link (CXL) Specification, Revision 3.0. <https://www.computeexpresslink.org/> Accessed 2024-04-13.
- [11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, IN, USA) (SoCC ’10). Association for Computing Machinery, New York, NY, USA, 143–154. <https://doi.org/10.1145/1807128.1807152>
- [12] Al Danial. 2022. CLOC: Count Lines of Code. <https://github.com/AlDanial/cloc>
- [13] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (Seattle, WA, USA) (NSDI ’14). USENIX Association, Berkeley, CA, USA, 401–414. <https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic%24%87>
- [14] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. 2004. How fast can a distributed atomic read be?. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing* (St. John’s, Newfoundland, Canada) (PODC ’04). Association for Computing Machinery, New York, NY, USA, 236–245. <https://doi.org/10.1145/1011767.1011802>
- [15] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. In *Proceedings of the 16th European Conference on Computer Systems* (Virtual Event) (EuroSys ’21). Association for Computing Machinery, New York, NY, USA, 178–193. <https://doi.org/10.1145/3447786.3456236>
- [16] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. 2009. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *13th International Conference on Principles of Distributed Systems* (Nîmes, France) (OPODIS ’09). Springer-Verlag, Berlin, Germany, 240–254. https://doi.org/10.1007/978-3-642-10877-8_20
- [17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (apr 1985), 374–382. <https://doi.org/10.1145/3149.214121>

- [18] Aishwarya Ganesan, Ramnaththan Alagappan, Andrea C. Arpac-Dusseau, and Remzi H. Arpac-Dusseau. 2021. Strong and Efficient Consistency with Consistency-aware Durability. *ACM Transactions on Storage* 17, 1, Article 4 (Jan. 2021), 27 pages. <https://doi.org/10.1145/3423138>
- [19] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. 2008. On the Robustness of (Semi) Fast Quorum-Based Implementations of Atomic Shared Memory. In *Proceedings of the 22nd International Symposium on Distributed Computing* (Arcachon, France) (*DISC '08*, Vol. 5218). Springer-Verlag, Berlin, Germany, 289–304. https://doi.org/10.1007/978-3-540-87779-0_20
- [20] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. 2009. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel and Distrib. Comput.* 69, 1 (2009), 62–79. <https://doi.org/10.1016/j.jpdc.2008.05.004>
- [21] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA). USENIX Association, Berkeley, CA, USA, 649–667. <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu>
- [22] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. 2022. uKharon: A Membership Service for Microsecond Applications. In *Proceedings of the 2022 USENIX Annual Technical Conference* (Carlsbad, CA, USA) (*USENIX ATC '22*). USENIX Association, Berkeley, CA, USA, 101–120. <https://www.usenix.org/conference/atc22/presentation/guerraoui>
- [23] Red Hat. 2020. RHEL for Real Time Timestamping. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-timestamping Accessed: April 14, 2023.
- [24] Chris Hawblitzel, Jon Howell, Manos Kapritsis, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Monterey, CA, USA) (*SOSP '15*). Association for Computing Machinery, New York, NY, USA, 1–17. <https://doi.org/10.1145/2815400.2815428>
- [25] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (Jan. 1991), 124–149. <https://doi.org/10.1145/114005.102808>
- [26] Maurice Herlihy and Jeannette Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. <https://doi.org/10.1145/78969.78972>
- [27] Kaile Huang, Yu Huang, and Hengfeng Wei. 2020. Fine-grained Analysis on Fast Implementations of Distributed Multi-writer Atomic Registers. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing* (Virtual Event) (*PODC '20*). Association for Computing Machinery, New York, NY, USA, 200–209. <https://doi.org/10.1145/3382734.3405698>
- [28] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Boston, MA, USA) (*USENIX ATC '10*). USENIX Association, Berkeley, CA, USA, 11 pages. <https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems>
- [29] Sunita Jain, Nagaradheesh Yelavarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. In *Proceedings of the 3rd Workshop on Heterogeneous Composable and Disaggregated Systems* (San Diego, CA, USA) (*HCDs '24*). Association for Computing Machinery, New York, NY, USA, 4 pages. <https://arxiv.org/pdf/2404.03245.pdf>
- [30] Hai Jin, Rajkumar Buyya, and Toni Cortes. 2002. An Introduction to the InfiniBand Architecture. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications* (1st ed.). John Wiley and Sons, Inc., Hoboken, NJ, USA, 616–632. <https://doi.org/10.1109/9780470544839>
- [31] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) (*NSDI '19*). USENIX Association, Berkeley, CA, USA, 1–16. <https://www.usenix.org/conference/nsdi19/presentation/kalia>
- [32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, IL, USA) (*SIGCOMM '14*). Association for Computing Machinery, New York, NY, USA, 295–306. <https://doi.org/10.1145/2619239.2626299>
- [33] Kishori M. Konwar, Saptaparni Kumar, and Lewis Tseng. 2020. Semi-Fast Byzantine-tolerant Shared Register without Reliable Broadcast. In *Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems* (Singapore) (*ICDCS '20*). IEEE Computer Society, Los Alamitos, CA, USA, 743–753. <https://doi.org/10.1109/ICDCS47774.2020.00057>
- [34] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. <https://doi.org/10.1145/359545.359563>
- [35] Leslie Lamport. 1984. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems* 6, 2 (April 1984), 254–280. <https://doi.org/10.1145/2993.2994>
- [36] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169. <https://doi.org/10.1145/279227.279229>
- [37] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholz, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the 18th European Conference on Computer Systems* (Rome, Italy) (*EuroSys '23*). Association for Computing Machinery, New York, NY, USA, 718–733. <https://doi.org/10.1145/3552326.3567498>
- [38] Sekwon Lee, Soujanya Ponnappalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proceedings of the VLDB Endowment* 15, 13 (Sept. 2022), 4023–4037. <https://doi.org/10.14778/3565838.3565854>
- [39] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) (*ASPLOS '23*). Association for Computing Machinery, New York, NY, USA, 574–587. <https://doi.org/10.1145/3575693.3578835>
- [40] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (*FAST '23*). USENIX Association, Berkeley, CA, USA, 99–114. <https://www.usenix.org/conference/fast23/presentation/li-pengfei>
- [41] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, USA) (*OSDI '23*). USENIX Association, Berkeley, CA, USA, 305–323. <https://www.usenix.org/conference/osdi23/presentation/lu>
- [42] Nancy A. Lynch and Alexander A. Shvartsman. 1997. Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing* (Seattle, WA, USA) (*FTCS '97*). IEEE Computer Society,

- Los Alamitos, CA, USA, 272–281. <https://doi.org/10.1109/FTCS.1997.614100>
- [43] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 757–773. <https://doi.org/10.1145/3373376.3378511>
- [44] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, USA) (USENIX ATC '13). USENIX Association, Berkeley, CA, USA, 103–114. <https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell>
- [45] Mark Moir and James H. Anderson. 1995. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming* 25, 1 (1995), 1–39. [https://doi.org/10.1016/0167-6423\(95\)00009-H](https://doi.org/10.1016/0167-6423(95)00009-H)
- [46] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, PA, USA) (SOSP '13). Association for Computing Machinery, New York, NY, USA, 358–372. <https://doi.org/10.1145/2517349.2517350>
- [47] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (FAST '23). USENIX Association, Berkeley, CA, USA, 81–98. <https://www.usenix.org/conference/fast23/presentation/shen>
- [48] Mellanox Technologies. 2015. RDMA Aware Networks Programming User Manual. Rev 1.7. <https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17> Accessed 2024-03-17.
- [49] Shin-Yeh Tsai, Yizhou Shan, and Yiyang Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference* (Virtual Event) (USENIX ATC '20). USENIX Association, Berkeley, CA, USA, Article 3, 16 pages. <https://www.usenix.org/conference/atc20/presentation/tsai>
- [50] Lewis Tseng, Neo Zhou, Cole Dumas, Tigran Bantikyan, and Roberto Palmieri. 2023. Distributed Multi-writer Multi-reader Atomic Register with Optimistically Fast Read and Write. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) (SPAA '23). Association for Computing Machinery, New York, NY, USA, 479–488. <https://doi.org/10.1145/3558481.3591086>
- [51] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 1033–1048. <https://doi.org/10.1145/3514221.3517824>
- [52] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, USA) (OSDI '22). USENIX Association, Berkeley, CA, USA, 55–71. <https://www.usenix.org/conference/osdi22/presentation/zhou-yang>

NON-PEER-REVIEWED APPENDICES

A Reliable Max Register

Algorithm 8. Reliable Max Register Pseudocode

```

1 maxregs: shared array of unreliable max registers,
   ↪ initialized to [ $\perp, \dots, \perp$ ]
2 cache: local array of values, same length as
   ↪ maxregs, initialized to [ $\perp, \dots, \perp$ ]

4 def inner_write(v):
5   for mr in 0..maxregs.length:
6     if cache[mr] < v:
7       async:
8         maxregs[mr].write(v)
9         cache[mr] = v
10    wait until majority of cache >= v

12 def WRITE(v):
13   inner_write(v)

15 def READ():
16   for mr in 0..maxregs.length:
17     async: cache[mr] = maxregs[mr].read()
18   wait for a majority of answers
19   v = max(cache.values)
20   inner_write(v)
21   return v

```

Interface Max Registers have the following interface:

- $\text{READ}() \rightarrow v$
- $\text{WRITE}(v) \rightarrow \text{ok}$

Properties Max Registers satisfy the following properties:

- **Validity:** If $\text{READ } R$ returns v , then either (a) $v = \perp$, or (b) some operation $\text{WRITE}(v)$ was invoked before R returns.
- **Read-read monotonicity:** If a READ returns value y and a preceding READ returns value x , then $x \leq y$.
- **Write-read monotonicity:** If a READ returns value y and a preceding WRITE writes value x , then $x \leq y$.
- **Liveness (wait-freedom):** Every invoked operation eventually returns.

A.1 Correctness

We prove that our implementation of a reliable Max Register object, shown in Algorithm 8, satisfies the properties above.

Theorem A.1. Algorithm 8 satisfies validity.

Proof. We say that a value v is valid at time t if either (a) $v = \perp$, or (b) some operation $\text{WRITE}(v)$ was invoked before time t . Thus the validity property can be restated as: if $\text{READ } R$ returns v at time t , then v is valid at t . Note that if v is valid at t , then v is valid at any $t' \geq t$.

We prove validity by showing that at any time t , maxregs and cache only contain valid values. Then validity follows immediately, since the value returned by READ at p is the maximum value in p 's cache.

Initially maxregs and cache only contain \perp , which is a valid

value. Suppose by contradiction that there is a time t and a value v such that maxregs or cache contains v at time t and v is not valid at t . Assume wlog that t is the earliest such time.

Time t must correspond to a step by some process p which updates either maxregs or cache . Let us examine all such possible steps:

- Line 8. The value v is the argument of the inner_write function, thus either v is the argument of a WRITE at time $t' \leq t$, or v is obtained at Line 19 at time $t'' < t$. In the former case, v must be valid at time t' and thus also at time t , since it is the input of a WRITE invoked before t' . In the latter case, v is valid at time t'' , since $t'' < t$ and t is the earliest time when maxregs or cache contain an invalid value. In both cases, v is valid at t , a contradiction.
- Line 9. By the same argument as above, v must be valid at t , a contradiction.
- Line 17. This line contains two steps: first, at time $t' < t$, some value v is read from $\text{maxregs}[mr]$, and then, at time t , v is written into $\text{cache}[mr]$. By our assumption v must be valid at time t' (thus also at t), so we again reach a contradiction.

All possibilities lead to a contradiction, thus it must be the case that maxregs and cache only contain valid values at all times. This completes the proof. \square

Observation 1. If $\text{inner_write}(v)$ completes at time t at process p , then a majority of p 's cache slots contain values $\geq v$ at time t .

Lemma A.2. At any process p and time t , for any $0 \leq mr \leq \text{maxregs.length}$, if $\text{cache}[mr] = v$ at time t , then some $\text{maxregs}[mr].\text{write}(v)$ or $\text{maxregs}[mr].\text{read()} \rightarrow v$ completed before t .

Proof. Observe that $\text{cache}[mr]$ is only updated at Line 9 and Line 17. If $\text{cache}[mr]$ becomes equal to v at Line 9, then $\text{maxregs}[mr].\text{write}(v)$ was called in the previous line. Otherwise, if $\text{cache}[mr]$ becomes equal to v at Line 17, then $\text{maxregs}[mr].\text{read}()$ returned v on the same line. \square

Lemma A.3. If $\text{inner_write}(v)$ completes at time t , then after t , reading from all maxregs and waiting for a majority of responses produces at least one value $\geq v$.

Proof. Let p be the process that called $\text{inner_write}(v)$. By Observation 1, a majority M_1 of p 's cache slots contain values $\geq v$ at t . Let M_2 be the majority of maxregs for which READ responses are received. M_1 and M_2 must intersect in at least one index i . Since $i \in M_1$, by Lemma A.2, some $\text{maxregs}[i].\text{write}(v')$ or $\text{maxregs}[i].\text{read}() \rightarrow v'$ completed before t , where $v' \geq v$. Therefore, by the read-read or write-read monotonicity of $\text{maxregs}[i]$, the READ of $\text{maxregs}[i]$ after t returns a value $\geq v$, as required by the lemma. \square

Theorem A.4. Algorithm 8 satisfies read-read monotonicity.

Proof. Let R_1 and R_2 be two READS such that R_1 precedes R_2 , R_1 returns x , and R_2 returns y . We will show that $x \geq y$.

By Lemma A.3, since R_1 calls `inner_write(x)` before it returns, at least one of the READS performed by R_2 in lines 16–18 returns a value $\geq x$. So the cache of the invoking process of R_2 contains at least one value $\geq x$ at the time when it invokes Line 19. Thus the return value of R_2 is $\geq x$. \square

Theorem A.5. Algorithm 8 satisfies write-read monotonicity.

Proof. Let W be a WRITE and R be a READ such that W precedes R , W writes x , and R returns y . We will show that $x \geq y$.

By Lemma A.3, since W calls `inner_write(x)` before it returns, at least one of the READS performed by R in lines 16–18 returns a value $\geq x$. So the cache of the invoking process of R contains at least one value $\geq x$ at the time when it invokes Line 19. Thus the return value of R is $\geq x$. \square

Theorem A.6. Algorithm 8 satisfies wait-freedom.

Proof. We start by showing that the `inner_write` function is wait-free. Let p be the process that invokes it. The only blocking step of the function is Line 10. If p 's local cache already contains a majority of values $\geq v$, then the function returns immediately, without waiting for any asynchronous operation invoked in the for loop. Otherwise, the local cache contains a majority of values $< v$, so the if statement at Line 6 is triggered for a majority of indices. Thus, v will be written asynchronously to a majority of locations in p 's cache. At the latest when all such WRITES have completed, the wait statement at Line 10 and the function will return.

The WRITE operation is a mere call to `inner_write`, so it is clearly wait-free as well.

The READ operation has an additional blocking step at Line 18. This step completes once a majority of the max registers have been read from, which cannot be blocked or prevented. Then, the READ operation invokes `inner_write`, which is wait-free and thus must eventually return. So the READ operation is wait-free as well. \square

A.2 Roundtrip Complexity

We assume that a WRITE to, or READ from, one of the underlying unreliable max registers takes 1 roundtrip time (RTT) (this is true in our implementation). Then, the `inner_write` function returns in at most 1 RTT, since all of the WRITES in Line 8 are performed asynchronously in parallel. Furthermore, if the cache already contains v , or a larger value, in a majority of slots, then `inner_write` returns immediately, in 0 RTTs, even if the if statement at Line 6 is triggered for a minority of cache slots. This is because the resulting WRITES to `maxregs` are performed in the background and do not block

`inner_write` from returning. Since the `WRITE` function is just a wrapper around `inner_write`, all of the above is also true about `WRITE`.

The READ function always requires 1 RTT to complete lines 16–18, after which it calls the `inner_write` function which, as discussed above, may incur 0 or 1 RTTs. Overall, READ incurs 1 or 2 RTTs. However, there are two common-case scenarios in which READ requires only 1 RTT:

- Under low contention, we can assume that WRITES are not concurrent with any other operation, and that enough time passes between a `WRITE(v)` and a subsequent READ such that the background WRITES of v to `maxregs` complete before the READ begins. Then, when the READ performs the READS at lines 16–18, all such READS will return v , and thus will populate a majority of the local cache with v . Therefore, the `inner_write` at Line 20 will return in 0 RTTs.
- It is often the case that, among a collection of machines, some are consistently more responsive than others. If this is the case for the memories hosting each max register in `maxregs`, then all operations will receive responses first from the same majority of `maxregs` for writing and reading (lines 5–10 and lines 16–18). If this occurs, and there is low contention (WRITES are not concurrent with any other operation), then READS always return in 1 RTT, even if operations occur one right after the other, without any background operations completing in the meantime.

In ABD and Safe-Guess, when a Max Register is read to obtain a fresh timestamp as part of a WRITE (Algorithms 1 and 2, respectively), read-read monotonicity is unnecessary. It is then enough to use a weaker READ operation that does not call `inner_write(v)` and always completes in 1 RTT. Our proof does not cover this optimization.

B Timestamp Lock

Algorithm 9. Timestamp Lock Pseudocode

```

1 CASes = {(\perp, \perp), ...} // 2f+1 CAS Objects
2
3 def TRYLOCK(ts, mode: READ or WRITE):
4     read: dict<CAS, CasValue> = {(\perp, \perp), ...}
5     async for c in CASes:
6         while read[c].ts < ts:
7             expected = read[c]
8             read[c] = c.CAS(expected, (ts, mode))
9             if read[c] == expected: break
10        wait for a majority to complete
11        if any c s.t. read[c].ts > ts: return False
12        if any c s.t. read[c] == (ts, ~mode): return False
13    return True

```

Interface Timestamp locks have a single operation:

- `TRYLOCK(ts, mode: READ or WRITE) → bool`

Properties Timestamp locks satisfy three properties:

- True safety: if a lock operation $\text{TRYLOCK}(ts, mode)$ returns r at time t , and no operation $\text{TRYLOCK}(ts', mode')$ was invoked before t such that (i) $ts' > ts$, or (ii) $ts' = ts$ and $mode' = \neg mode$, then $r = \text{True}$.
- True exclusion: $\text{TRYLOCK}(ts, m)$ and $\text{TRYLOCK}(ts, \neg m)$ cannot both return True .
- Liveness (wait-freedom): Every operation eventually returns.

B.1 Correctness

Theorem B.1. Algorithm 9 satisfies True safety.

Proof. Let us define a $\text{TRYLOCK}(ts', mode')$ operation to be $(ts, mode, t)$ -conflicting if it is invoked before time t and either (i) $ts' > ts$ or (ii) $ts' = ts$ and $mode' = \neg mode$. Thus, the theorem can be stated as: if a lock operation $\text{TRYLOCK}(ts, mode)$ returns r at time t , and no $(ts, mode, t)$ -conflicting operation exists, then $r = \text{True}$.

Let D be a $\text{TRYLOCK}(ts, m)$ operation that returns at time t and assume no $(ts, mode, t)$ -conflicting operation exists. We will show that D must return True . Since no $(ts, mode, t)$ -conflicting operation exists, by time t none of the shared CAS objects contains a tuple $(ts', mode')$ such that $ts' > ts$, or $ts' = ts$ and $mode' = \neg mode$, since a CAS object is populated with $(ts, mode)$ only at Line 8 and only if $(ts, mode)$ are the arguments to TRYLOCK . Thus also none of the fields of the read object in D can contain such a tuple either by time t , since this read array is only populated with values from the CAS objects (Line 8). It then follows that neither of the if statements at Line 11 or Line 12 is triggered, so D must return True at Line 13. \square

Observation 2. In Algorithm 9, CAS objects never decrease in value. Furthermore, CAS objects never change their contents from $(ts, mode)$ to $(ts, \neg mode)$ for any $ts, mode$.

Proof. CAS objects are only updated at Line 8. Take a CAS object c . At Line 8, $c.\text{CAS}(\text{expected}, (ts, mode))$ is successful only if (1) c already contains expected , which is equal to $\text{read}[c]$, and (2) $ts > \text{read}[c]$. Thus, c takes strictly monotonically increasing values over time. \square

Theorem B.2. Algorithm 9 satisfies True exclusion.

Proof. Assume by contradiction that there exist two operations $D_1 = \text{TRYLOCK}(ts, mode)$ and $D_2 = \text{TRYLOCK}(ts, \neg mode)$ such that D_1 and D_2 both return True .

Since D_1 returns, a majority of iterations of the for loop at Line 5 must complete. Denote by M_1 the set of CAS objects whose iterations complete. Similarly for D_2 ; denote by M_2 the set of CAS objects whose iterations complete at D_2 . M_1 and M_2 must intersect in at least one CAS object m . D_1 must observe m to contain $(ts, mode)$, otherwise D_1 would return True at Line 11 or Line 12. Similarly, D_2 must observe m to contain $(ts, \neg mode)$. Thus, m must change in

value from $(ts, \neg mode)$ to $(ts, mode)$ or vice-versa, possibly passing through one or more intermediate values. By Observation 2, m cannot pass directly from $(ts, mode)$ to $(ts, \neg mode)$ or vice-versa. Furthermore, also by Observation 2, if m changes from (ts, \cdot) to some other value (ts', \cdot) , then $ts' > ts$, after which m will never again contain (ts, \cdot) . We have reached a contradiction in all cases, thus the theorem must hold. \square

Theorem B.3. Algorithm 9 satisfies wait-freedom.

Proof. The only blocking step of the TRYLOCK operation is at Line 10. To prove that this step eventually completes, we show that a majority of the iterations of the for loop eventually complete.

Let c be a correct CAS object, i.e., one that never fails. We show that at each iteration of the while loop, either (a) the CAS operation succeeds, or (b) the contents of $\text{read}[c]$ strictly increase. Assume the CAS operation fails and consider two cases:

- $\text{read}[c] = \perp$. Since the CAS fails, $\text{read}[c]$ becomes the current value of c , which cannot be equal to \perp , otherwise the CAS would succeed. Since any value is greater than \perp , $\text{read}[c]$ strictly increases in value.
- $\text{read}[c] \neq \perp$. In this case, $\text{read}[c]$ must be equal to the contents of c from a previous iteration of the while loop. After the CAS operation, $\text{read}[c]$ becomes equal to the current contents of c . By Observation 2, the contents of c can only strictly increase in value, thus $\text{read}[c]$ also strictly increases in value in this case.

We have shown that at each iteration of the while loop, either (a) the CAS operation succeeds, or (b) the contents of $\text{read}[c]$ strictly increase. In case (a), the while loop will exit immediately afterwards, due to the check at Line 9. In case (b), there can only be a finite number of such iterations, since ts is a fixed value and thus $\text{read}[c]$ can only take a finite number of values before it becomes greater than ts , at which point the while loop, and corresponding for loop iteration, complete.

We have shown that for each correct CAS object, the corresponding for loop iteration eventually completes. Since we assume there are a majority of such correct CAS objects in the system, it follows that a majority of the iterations complete. Thus, each TRYLOCK operation eventually completes. \square

B.2 Roundtrip Complexity

We assume that performing a CAS on an underlying CAS object incurs a single RTT. Thus, each iteration of the while loop at Line 6 takes 1 RTT. Therefore, the round complexity of a TRYLOCK call is equal to the highest number of iterations of the while loop performed before a majority of iterations of the for loop have completed.

Fix a CAS object c . At the start of the TRYLOCK call, $\text{read}[c] = \perp$. In the worst case, $\text{read}[c]$ takes a new value between \perp and ts every time the CAS at Line 8 is attempted. Since

Algorithm 10. Safe-Guess' Pseudocode

```

1 M = ((0, ⊥), VERIFIED, ⊥) // Max Register
2 TSL[tid] = {} // Timestamp Lock

4 def WRITE(v):
5   w = (guessTs(), GUESSED, v)
6   in parallel {m = M.READ(), M.WRITE(x)}
7   if m <= w: // Fast path
8     in bg: M.WRITE(w with VERIFIED) // Spdup reads
9   else: // Slow path (potentially stale timestamp)
10    if TSL[tid].TRYLOCK(w.ts, WRITE):
11      M.WRITE(((m.ts.i+1,tid), VERIFIED, v))

13 def READ():
14   seen: dict<ThreadId, MValue> = {}
15   while True:
16     m = M.READ()
17     if m is VERIFIED: return m.v // Fast path
18     if m in seen.values: // Fresh timestamp
19       if TSL[m.ts.tid].TRYLOCK(m.ts, READ):
20         in bg: M.WRITE(m with VERIFIED) // Spdup rds
21         return m.v
22     elif m.ts.tid in seen.keys: // Wait-free path
23       return seen[m.ts.tid].v
24     seen[m.ts.tid] = m

```

timestamps can only take positive, integer values, this means that at most $ts + 1$ iterations can be performed before c takes a value $\geq ts$. Therefore, the worst case complexity of $TRYLOCK(ts, mode)$ is $ts + 1$.

C Safe-Guess

Safe-Guess implements a wait-free atomic multi-writer multi-reader register, with the standard read/write interface and usual correctness properties: linearizability and wait-freedom.

C.1 Linearizability

We prove the linearizability of Safe-Guess by describing an explicit linearization for each execution of Safe-Guess and then showing that the linearization is valid: (1) the linearization is a legal sequential execution of a register, (2) the linearization is equivalent to the concurrent execution from which it is derived, and (3) the linearization preserves the real-time ordering of the concurrent execution.

C.1.1 Preliminaries We define the notion of *timestamp of a WRITE*.

Definition 1. We associate each WRITE W with a timestamp $TS(W)$, in the following way:

- If W performs the WRITE at Line 11, then $TS(W)$ is the timestamp computed by W in Line 11. We also call this the *verified timestamp of W* .
- Otherwise, then $TS(W)$ is the timestamp guessed by W in Line 5. We also call this the *guessed timestamp of W* .

We also rely on a number of observations, assumptions and lemmas in our proof of linearizability. We start with

the following observation, which follows from the fact that READS only return values read from M , and WRITES to M are always input values of WRITES.

Observation 3. In Algorithm 10, the return value of a READ operation is the input value of a WRITE operation, or the initial value \perp .

Furthermore, we make the (reasonable) assumption that a process guesses for each WRITE a higher timestamp than it used for all previous WRITES. If the verified timestamp is computed, it is even higher, thus higher than all previous timestamps used by this process.

Assumption 1. In Algorithm 10, if W_1 and W_2 are two WRITES by the same process such that W_1 precedes W_2 , then $TS(W_1) < TS(W_2)$.

We also make the standard assumption [26] that each thread has at most one outstanding operation at a time.

Assumption 2. In any valid execution, a thread T invokes a new operation only after T 's current operation completes.

The next observation follows from Assumptions 1 and 2, and from the fact that timestamps include the process identifier.

Observation 4. In Algorithm 10, no two WRITES have the same timestamp, in the sense of Definition 1.

We next prove the following technical lemma about our reliable max register, which will help us reason about repeated READS of M performed in different iterations of the READ function's while loop.

Lemma C.1 (Double-read lemma). *Let M be a Reliable Max Register, and $a < b$ two values. Let R_1 and R_2 be two READS from M such that R_1 precedes R_2 and both R_1 and R_2 return a . Then, no WRITE of b precedes the first invocation of a WRITE of a .*

Proof. Assume not. Let $W(a)$ be the WRITE of a with the earliest invocation. Let $W(b)$ be a WRITE of b such that $W(b)$ precedes $W(a)$. We have the following:

- $t_1 > t_2$, where t_1 is the return time of $W(b)$ and t_2 is the invocation time of R_2 . This is because of the write-read monotonicity of M .
- $t_2 > t_3$, where t_3 is the return time of R_1 . This is by our assumption that R_1 precedes R_2 .
- $t_3 > t_4$, where t_4 is the invocation time of $W(a)$. This is because of the validity property of M .
- $t_4 > t_1$, by our assumption that $W(b)$ precedes $W(a)$.

We have reached a circular inequality, thus the lemma must hold. \square

Finally, we prove the following lemma which helps explain what happens if TRYLOCK at Line 10 returns False.

Lemma C.2. *In Algorithm 10, let W be a WRITE operation and w the tuple it initializes at Line 5. If W performs the TRYLOCK at Line 10 and gets False, then some READ operation reads w from \mathbb{M} in two separate iterations of its while loop.*

Proof. Let p be the process that executes W . By the True safety property of the timestamp lock, if W 's trylock call returns False, then there exists some concurrent or preceding trylock call A with either (1) a higher timestamp than w , or (2) the same timestamp but in read mode. Case (1) is impossible: if A is invoked by a WRITE, this WRITE must be performed by p (since timestamp locks are indexed per process in Algorithm 10) earlier than W with a higher timestamp—impossible by Assumption 1; if A is invoked by a READ, this READ must have seen the higher timestamp associated to a value written by p —impossible. Thus, case (2) must be true: A must have the same timestamp as W but used the read lock mode. This is only possible if a READ R has read the value of W from \mathbb{M} . Furthermore, R must have read this value twice, since R invokes TRYLOCK only if R finds the current value in its seen collection, meaning that the same value was read in a previous iteration. \square

C.1.2 Linearization Rules & Proof Now we can give our linearization construction. Given an execution \mathcal{E} of Safe-Guess, we linearize it as follows:

1. Linearize WRITES in order of increasing timestamps, in the sense of Definition 1.
2. If a READ R returns \perp , linearize R before the first WRITE.
3. Otherwise, linearize a READ R after the WRITE whose value it returns (Observation 3), but before the following WRITE.
4. If two READS R_1 and R_2 return the same value and R_1 precedes R_2 in \mathcal{E} , linearize R_1 before R_2 .

We denote by $\mathcal{L}(\mathcal{E})$ the resulting linearization. We now show that $\mathcal{L}(\mathcal{E})$ is valid. Clearly, $\mathcal{L}(\mathcal{E})$ is a legal sequential execution of a register, since every READ returns the value of the latest WRITE, with the exception of READS occurring before the first WRITE, which return \perp . Furthermore, $\mathcal{L}(\mathcal{E})$ is equivalent to \mathcal{E} , since all operations return the same values.

It only remains to prove that $\mathcal{L}(\mathcal{E})$ preserves the real-time ordering present in \mathcal{E} . We show that if some operation O_1 precedes another operation O_2 in \mathcal{E} , then the same is true in $\mathcal{L}(\mathcal{E})$. We consider all four cases $(O_1, O_2) \in \{\text{write, read}\} \times \{\text{write, read}\}$.

$(O_1, O_2) = (\text{write, write})$ We rely on the following lemma:

Lemma C.3. *If W_1 and W_2 are two WRITES such that W_1 precedes W_2 , then $TS(W_1) < TS(W_2)$.*

Proof. By the write-read monotonicity of \mathbb{M} , the READ of W_2 returns a value m with a timestamp not lower than $TS(W_1)$. We distinguish three cases:

- W_2 performs the fast path. In this case, $m \leq w$, where w is the value computed at Line 5. By Definition 1, w is also

equal to $TS(W_2)$. It follows that $TS(W_2) > TS(W_1)$.

- W_2 performs the WRITE at Line 11. In this case, we have $TS(W_2) = m.ts + 1 > m.ts \geq TS(W_1)$.
- W_2 enters the slow path, but does not perform the WRITE at Line 11. In this case, the TRYLOCK call at Line 10 must have returned False. By Lemma C.2, some READ must have read w twice, sequentially. By Lemma C.1, no WRITE to \mathbb{M} with a higher timestamp than w could have preceded W_2 's WRITE of w at Line 6. So $TS(W_1)$ must be lower than $w.ts = TS(W_2)$.

\square

Thus, according to the first linearization rule, W_2 is linearized after W_1 , as required.

$(O_1, O_2) = (\text{read, read})$ Let R_1 and R_2 be two READS such that R_1 precedes R_2 . We show that R_1 is linearized before R_2 .

In case R_1 and R_2 return the same value, then by the fourth linearization rule, R_1 is linearized before R_2 .

In case R_1 and R_2 return different values, let v_1 and v_2 be their respective return values. Let r_1 be the (latest) M.READ call inside R_1 that returns v_1 and r_2 be the (latest) M.READ call inside R_2 that returns v_2 . Since r_1 precedes r_2 , by the read-read monotonicity of \mathbb{M} , it must be that $v_1 < v_2$. By the second and third linearization rules, R_1 is linearized before the WRITE of v_2 , and R_2 is linearized after the WRITE of v_2 , so R_1 is linearized before R_2 .

$(O_1, O_2) = (\text{write, read})$ Let W and R be a WRITE and a READ, respectively, such that W precedes R . We show that W is linearized before R .

Let v be the return value of R and let r be the earliest M.READ call inside R that returns v . By the write-read monotonicity of \mathbb{M} , the timestamp associated with v cannot be lower than the timestamp of W . In other words, R returns a value installed by W or a later WRITE, so by the third linearization rule R must be linearized after W .

$(O_1, O_2) = (\text{read, write})$ Let R and W be a WRITE and a READ, respectively, such that R precedes W . We show that R is linearized before W .

Let T be the timestamp associated to the value returned by R . Since W is invoked after R returns, by read-read monotonicity, W 's READ at Line 6 returns a value m with a timestamp $T_m \geq T$. We distinguish three cases:

- W enters the fast path. In this case, W 's guessed timestamp must have been greater than T_m , so in turn greater than T .
- W performs the WRITE at Line 11. In this case, W picks a timestamp greater than T_m , so in turn greater than T .
- W enters the fast path but does not perform the WRITE at Line 11. Then, by Lemma C.2, some READ operation must have read w (i.e., W 's value, initialized at Line 5) from \mathbb{M} twice, sequentially. The second such READ must

have been invoked after W was invoked, so by read-read monotonicity, the timestamp this second READ sees (which is the timestamp of w) must be greater than T .

We have shown that in all cases, W has a higher timestamp than the value returned by R . R therefore cannot return W 's value, or a later one, and is linearized before W .

C.2 Wait-Freedom

The WRITE function is clearly wait-free, since it does not have any blocking steps, and the external functions it calls ($M.\text{READ}$, $M.\text{WRITE}$, $\text{TSL}.\text{TRYLOCK}$) are all wait-free.

The case of the READ function is less straightforward. We start with the following observation, which follows from the fact that the WRITE and READ functions call $\text{TRYLOCK}(m.ts, *)$ only after having written m to M or read m from M , respectively.

Observation 5. *In Algorithm 10, if $\text{TSL}[p].\text{TRYLOCK}(m.ts, *)$ is invoked at time t , then either $M.\text{WRITE}(m)$ or $M.\text{READ}() \rightarrow m$ completed before t .*

Using Observation 5, we prove the following lemma, which crucially bounds the number of times a READ can see values from each process before the READ is forced to return.

Lemma C.4. *If a READ operation R reads a value from the same process p in more than two iterations of R 's while loop, then R returns on the third iteration in which R reads from p .*

Proof. Assume not: R reads a value from the same process p in three iterations, without returning on the third such iteration. Clearly, none of the values can be VERIFIED, otherwise R would return. Furthermore, since R returns as soon as it sees two distinct values from the same process, the values must all be equal to each other: call this value m . The second time R reads m , it invokes $\text{TSL}[m.ts.\text{tid}].\text{TRYLOCK}(m.ts, \text{READ})$ at Line 19. This call, denoted by C , must return False, otherwise R would return on the second iteration.

By True safety, some $\text{TSL}[p].\text{TRYLOCK}(m'.ts, \text{mode}')$ call must be concurrent with, or precede, C , with either (i) $m'.ts > m.ts$, or (ii) $m'.ts = m.ts$ and $\text{mode}' = \text{WRITE}$.

- In case (i), by Observation 5, some preceding READ or WRITE of m' must exist. Thus, in the third iteration of R which reads from p , by M 's monotonicity properties, $M.\text{READ}$ cannot return m again. This means that in this third iteration that reads from p , R sees a value different from m , and returns. Contradiction.
- In case (ii), p must have called $\text{TSL}[p].\text{TRYLOCK}(m.ts, \text{WRITE})$ as part of the slow path of some WRITE W , so W 's $M.\text{READ}$ at Line 5 returned a value m' with a higher timestamp than that of m . By read-read monotonicity, in the third iteration of R which reads from p , $M.\text{READ}$ cannot return m again. This again means that in this third iteration that reads from p , R sees a value different from m , and returns. Contradiction.

We have reached a contradiction in all cases, thus the lemma must hold. \square

Wait-freedom follows easily from Lemma C.4: since a READ can see values from each process in at most two iterations without returning, each READ must return after at most $2n+1$ iterations, where n is the number of processes.

C.3 Roundtrip Complexity

Recall that $M.\text{WRITE}$ incurs 0 or 1 RTTs, $M.\text{READ}$ incurs 1 or 2 RTTs, and $\text{TRYLOCK}(ts, p)$ takes between 1 and $ts + 1$ RTTs.

First, let us examine the round complexity of Safe-Guess's WRITE function. Line 6 takes between 1 and 2 RTTs. If the fast path is entered, no further RTTs are incurred on the critical path. Otherwise, the slow path is entered, and TRYLOCK takes between 1 and $w.ts + 1$ RTTs, where $w.ts$ is the output of $\text{guessTs}()$. Finally, if TRYLOCK returns True, then the WRITE at Line 11 takes an additional 0 or 1 RTTs. Overall, the WRITE function incurs 1 or 2 RTTs on the fast path and between 2 and $2 + w.ts + 1 + 1 = w.ts + 4$ RTTs on the slow path.

Now let us examine the round complexity of Safe-Guess's READ function. Each iteration of the while loop incurs 1 or 2 RTTs for the $M.\text{READ}$ at Line 16, as well as between 1 and $1+m.ts$ RTTs in case the TRYLOCK at Line 19 is invoked. So each iteration of the while loop incurs between 1 and $3 + m.ts$ RTTs. As argued above, a READ performs at most $2n + 1$ iterations of the while loop, so the overall complexity of a READ is between 1 and $(2n + 1)(3 + m.ts)$ RTTs.

When do writes and reads return in a single RTT?

A WRITE returns in a single RTT if: (i) the $M.\text{READ}$ at Line 6 returns in a single RTT and (ii) the WRITE takes the fast path.

- As discussed in Appendix A.2, condition (i) happens if the write-back part of the READ is not necessary, because a majority of the underlying max registers already contain the max value. This can occur if there is low contention, or if some majority of the underlying max registers is consistently more responsive than the rest of the max registers.
- Condition (ii) occurs if writer p guesses a fresh timestamp, and no other process writes a higher timestamp before p reads from M . Process p guesses a fresh timestamp if the system has good clock synchrony.

A READ returns in a single RTT if and only if (1) the $M.\text{READ}$ on the first iteration of the while loop returns in 1 RTT, and (2) the value returned by this READ is VERIFIED. Condition (1) occurs in the same scenarios as discussed above for (i). Condition (2) is true if (a) the previous WRITE performed the slow path WRITE at Line 11, or (b) the previous WRITE performed the fast path and enough time has passed, such that its background VERIFIED WRITE has completed, or (c) the previous WRITE performed the fast path and, in the meantime, some READ saw the GUESSED value and wrote it back as a VERIFIED value.

Overall, in the common case of low contention and good clock synchrony, every READ and WRITE operation completes in a single RTT.