FUSEE: A Fully Memory-Disaggregated Key-Value Store (Extended Version)

Jiacheng Shen[1*], Pengfei Zuo[2], Xuchuan Luo[3], Tianyi Yang[1],
Yuxin Su[4], Yangfan Zhou[3], and Michael R. Lyu[1]

[1]*The Chinese University of Hong Kong,* [2]*Huawei Cloud,* [3]*Fudan University,* [4]*Sun Yat-sen University*

Abstract

Distributed in-memory key-value (KV) stores are embracing the disaggregated memory (DM) architecture for higher resource utilization. However, existing KV stores on DM employ a *semi-disaggregated* design that stores KV pairs on DM but manages metadata with monolithic metadata servers, hence still suffering from low resource efficiency on metadata servers. To address this issue, this paper proposes FUSEE, a ***FU****lly memory-di****S****aggr****E****gated* KV Stor***E*** that brings disaggregation to metadata management. FUSEE replicates metadata, *i.e.*, the index and memory management information, on memory nodes, manages them directly on the client side, and handles complex failures under the DM architecture. To scalably replicate the index on clients, FUSEE proposes a client-centric replication protocol that allows clients to concurrently access and modify the replicated index. To efficiently manage disaggregated memory, FUSEE adopts a two-level memory management scheme that splits the memory management duty among clients and memory nodes. Finally, to handle the metadata corruption under client failures, FUSEE leverages an embedded operation log scheme to repair metadata with low log maintenance overhead. We evaluate FUSEE with both micro and YCSB hybrid benchmarks. The experimental results show that FUSEE outperforms the state-of-the-art KV stores on DM by up to 4.5 times with less resource consumption.

1 Introduction

Traditional in-memory key-value (KV) stores on monolithic servers have recently been ported to the disaggregated memory (DM) architecture for better resource efficiency [60, 73]. Compared with monolithic servers, DM decouples the compute and memory resources into independent network-attached compute and memory pools [3, 23, 25, 39, 48, 55, 56, 65]. KV stores on DM can thus enjoy efficient resource pooling and have higher resource efficiency.

However, constructing KV stores on DM is challenging because the memory pool generally lacks the compute power to manage data and metadata. Existing work [60] proposes a *semi-disaggregated* design that stores KV pairs in the disaggregated memory pool but retains metadata management on monolithic servers. In such a design, the KV pair storage enjoys high resource utilization due to exploiting the DM architecture, but the metadata management does not. Many additional resources are exclusively assigned to the metadata servers in order to achieve high overall throughput [13, 54, 69].

To achieve full resource utilization, it is critical to bring disaggregation to the metadata management, *i.e.*, building a *fully memory-disaggregated* KV store. The metadata, *i.e.*, the index and memory management information, should be stored in the memory pool and directly managed by clients rather than metadata servers. However, it is non-trivial to achieve a fully memory-disaggregated KV store due to the following challenges incurred from handling complex failures and the weak compute power in the memory pool.

1) Client-centric index replication. To tolerate memory node failures, clients need to replicate the index on memory nodes in the memory pool and guarantee the consistency of index replicas. In existing replication approaches, *e.g.*, state machine replication [34, 47, 51, 62] and shared register protocols [5, 7, 44], the replication protocols are executed by server-side CPUs. These protocols cannot be executed on DM due to the weak compute power in the memory pool. Meanwhile, if clients simply employ consensus protocols [37, 47, 51] or remote locks [60], the KV store suffers from poor scalability due to the explicit serialization of conflicting requests [4, 11, 64, 70].

2) Remote memory allocation. Existing semi-disaggregated KV stores manage memory spaces with monolithic metadata servers. However, in the fully memory-disaggregated setting, such a server-centric memory management scheme is infeasible. Specifically, memory nodes cannot handle the compute-heavy fine-grained memory allocation for KV pairs due to their poor compute power [25, 60]. Meanwhile, clients cannot efficiently allocate memory spaces because multiple RTTs are required to modify the memory management information stored on memory nodes [39].

3) Metadata corruption under client failures. In semi-disaggregated KV stores, client failures do not affect metadata because the CPUs of monolithic servers exclusively modify metadata. However, clients directly access and modify metadata on memory nodes in the fully memory-disaggregated setting. As a result, client failures can leave partially modified metadata accessible by others, compromising the correctness of the entire KV store.

To address these challenges, we propose FUSEE, a fully memory-disaggregated key-value store that has efficient index replication, memory allocation, and fault-tolerance on DM.

*Work mainly done during the internship at Huawei Cloud.

First, to maintain the strong consistency of the replicated index in a scalable manner, FUSEE proposes the SNAPSHOT replication protocol. The key to achieving scalability is to resolve write conflicts without involving the expensive request serialization [7]. SNAPSHOT adopts three simple yet effective conflict-resolution rules on clients to allow conflicts to be resolved collaboratively among clients instead of sequentially. Second, to achieve efficient remote memory management, FUSEE employs a two-level memory management scheme that splits the server-centric memory management process into compute-light and compute-heavy tasks. The compute-light coarse-grained memory blocks are managed by the memory nodes with weak compute power, and the compute-heavy fine-grained objects are handled by clients. Finally, to deal with the problem of metadata corruption, FUSEE adopts an embedded operation log scheme to resume clients' partially executed operations. The embedded operation log reuses the memory allocation order and embeds log entries in KV pairs to reduce the log-maintenance overhead on DM.

We implement FUSEE from scratch and evaluate its performance using both micro and YCSB benchmarks [15]. Compared with Clover and pDPM-Direct [60], two state-of-the-art KV stores on DM, FUSEE achieves up to 4.5 times higher overall throughput and exhibits lower operation latency with less resource consumption. The code of FUSEE is available at https://github.com/dmemsys/FUSEE.

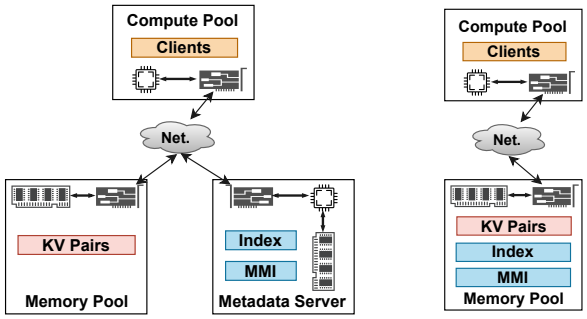
In summary, this paper makes the following contributions:

- A fully memory-disaggregated KV store with disaggregated metadata and data that is resilient to failures on DM.
- A client-centric replication protocol that uses conflict resolution rules to enable clients to resolve conflicts collaboratively. The protocol is formally verified with TLA+ [36] for safety and the absence of deadlocks under crash-stop failures.
- A two-level memory management scheme that leverages both memory nodes and clients to efficiently manage the remote memory space.
- An embedded operation log scheme to repair the corrupted metadata with low log maintenance overhead.
- The implementation and evaluation of FUSEE to demonstrate the efficiency and effectiveness of our design.

2 Background and Motivation

2.1 The Disaggregated Memory Architecture

The disaggregated memory architecture is proposed to address the resource underutilization issue of traditional datacenters composed of monolithic servers [25, 39, 48, 55, 56, 65]. DM separates CPUs and memory of monolithic servers into two independent hardware resource pools containing compute nodes (CNs) and memory nodes (MNs) [56, 60, 64, 73]. CNs have abundant CPU cores and a small amount of memory as local caches [64]. MNs host various memory media, *e.g.*, DRAM and persistent memory, to accommodate different application requirements with weak compute power. CPUs in CNs directly access memory in MNs with fast remote-access interconnect techniques, such as one-sided RDMA (remote direct memory access), Omni-path [16], CXL [43], and Gen-Z [14]. Each MN provides `READ`, `WRITE`, and atomic operations, *i.e.*, compare-and-swap (`CAS`) and fetch-and-add (`FAA`), for CNs to access memory data. Besides, MNs own limited compute power (*e.g.*, 1-2 CPU cores) to manage local memory and establish connections from CNs, providing CNs with the `ALLOC` and `FREE` memory management interfaces. Without loss of generality, in this paper, we consider CNs accessing MNs using one-sided RDMA verbs.



(a) Clover (b) FUSEE

Figure 1: Two architectures of memory-disaggregated KV stores. (a) The semi-disaggregated architecture (Clover [60]). (b) The fully disaggregated architecture proposed in this paper.

2.2 KV Stores on Disaggregated Memory

Clover [60] is a state-of-the-art KV store built on DM. It adopts a semi-disaggregated design that separates data and metadata to lower the ownership cost and prevent the compute power of data nodes from becoming the performance bottleneck. As shown in Figure 1a, Clover deploys clients on CNs and stores KV pairs on MNs. It adopts additional monolithic metadata servers to manage the metadata, including *memory management information (MMI)* and the *hash index*. For `SEARCH` requests, clients look up the addresses of the KV pairs from metadata servers and then fetch the data on MNs using `RDMA_READ` operations. For `INSERT` and `UPDATE` requests, clients allocate memory blocks from metadata servers with RPCs, write KV pairs to MNs with `RDMA_WRITE` operations, and update the hash index on the metadata servers through RPCs. To prevent clients' frequent requests from overwhelming the metadata servers, clients allocate a batch of memory blocks one at a time and cache the hash index locally. As a result, Clover achieves higher throughput under read-intensive workloads with less resource consumption.

However, the semi-disaggregated design of Clover cannot fully exploit the resource efficiency of the DM architecture due to its monolithic-server-based metadata management. On the one hand, monolithic metadata servers consume additional resources, including CPUs, memory, and RNICs. On the other hand, many compute and memory resources have to

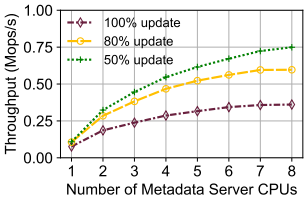


Figure 2: The throughput of Clover with an increasing number of metadata server CPUs.

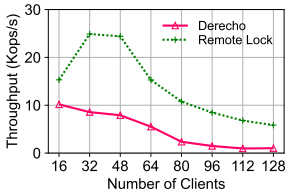


Figure 3: The throughput of Derecho [28] and lock-based approaches.

be reserved and assigned to the metadata server of Clover to achieve good performance due to the CPU-intensive nature of metadata management [13, 54, 69]. To show the resource utilization issue of Clover, we evaluate its throughput with 2 MNs, 64 clients, and a metadata server with different numbers of CPU cores. We control the number of CPU cores by assigning different percentages of CPU time with cgroup [10]. As shown in Figure 2, Clover has a low overall throughput with a small number of CPU cores assigned to its metadata server. At least six additional cores have to be assigned until the metadata server is no longer the performance bottleneck.

To attack the problem, FUSEE adopts a *fully memory-disaggregated* design that enables clients to directly access and modify the hash index and manage memory spaces on MNs, as shown in Figure 1b. Compared with the semi-disaggregated design, resource efficiency can be improved because client-side metadata management eliminates the additional metadata servers. The overall throughput can also be improved because the computation bottleneck of metadata management no longer exists.

3 Challenges

This section introduces the three challenges of constructing a fully memory-disaggregated KV store, *i.e.*, index replication, remote memory allocation, and metadata corruption.

3.1 Client-Centric Index Replication

The index must be replicated to tolerate MN failures. Strong consistency, *i.e.*, linearizability [26], is the most commonly adopted correctness standard for data replication because it reduces the complexity of implementing upper-level applications [1, 7, 12]. Linearizability requires that operations on an object appear to be executed in some total order that respects the operations' real-time order [26]. The key challenge of achieving a linearizable replicated hash index under the fully memory-disaggregated setting comes from the client-centric computation nature of DM.

First, existing replication methods are not applicable in the fully memory-disaggregated setting due to their server-centric nature. State machine replication (SMR) [34, 45, 47, 50, 51, 59, 62] and shared register protocols [7, 44] are two major replication approaches that achieve linearizability. However, both approaches are designed with a server-centric assumption that a data replica is exclusively accessed and modified by the CPU that manages the data. First, the SMR approaches consider the CPU and the data replica as a state machine and achieve strong consistency by forcing the state machines to execute deterministic KV operations in the same global order [50, 51]. Server CPUs are extensively used to reach a consensus on a global operation order and apply state transitions to data replicas. Second, shared register protocols view the CPU and the data replica as a shared register with `READ` and `WRITE` interfaces. Linearizability is achieved with a last-writer-wins conflict resolution scheme [44] that forces a majority of shared registers to always hold data with the newest timestamps. Shared register protocols also heavily rely on server-side CPUs to compare timestamps and apply data updates. The challenge with the server-centric approaches is that in the fully memory-disaggregated scenario, there is no such management CPU because all clients directly access and modify the hash index with one-sided RDMA verbs.

Second, naively adopting consensus protocols or remote locks among clients results in poor throughput due to the expensive request serialization. To show the performance issues of consensus protocols and remote locks, we store and replicate a shared object on two MNs and vary the number of concurrent clients. We use a state-of-the-art consensus protocol Derecho [28] and an RDMA `CAS`-based spin lock to ensure the strong consistency of the replicated object. As shown in Figure 3, both Derecho and lock-based approaches exhibit poor overall throughput and cannot scale with the growing number of concurrent clients.

3.2 Remote Memory Allocation

The key challenge of managing DM is where to execute the memory-management computation. There are two possible DM management approaches [39], *i.e.*, compute-centric ones and memory-centric ones. The compute-centric approaches store the memory management metadata on MNs and allow clients to allocate memory spaces by directly modifying the on-MN metadata. Since the memory management metadata are shared by all clients, clients' accesses have to be synchronized. As a result, compute-centric approaches suffer from the high memory allocation latency incurred by the expensive and complex remote synchronization process on DM [39]. The memory-centric approaches maintain all memory management metadata on MNs with their weak compute power. Such approaches are also infeasible because the poor memory-side compute power can be overwhelmed by the frequent fine-grained KV allocation requests from clients. Although there are several approaches that conduct memory management on DM, they all target page-level memory allocation and rely on special hardware, *i.e.*, programmable switches [39] and SmartNICs [25], which are orthogonal to our problem.

3.3 Metadata Corruption

In fully memory-disaggregated KV stores, crashed clients can leave partially modified metadata accessible by other

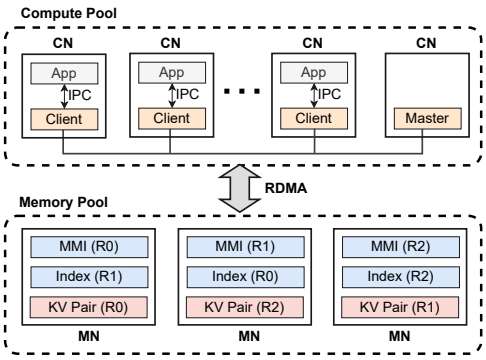


Figure 4: The FUSEE overview *(MMI, Index, and KV pairs have multiple replicas, i.e., R_0, R_1, and R_2. R_0 is the primary replica.)*.

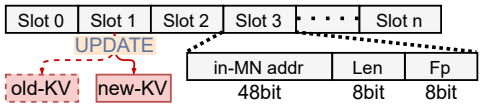


Figure 5: The structure of an index replica.

healthy clients. Since the metadata contains important system state, metadata corruption compromises the correctness of the entire KV store. First, crashed clients may leave the index in a partially modified state. Other healthy clients may not be able to access data or even access wrong data with the corrupted index. Second, crashed clients may allocate memory spaces but not use them, causing severe memory leakage. Hence, in order to ensure the correctness of the KV store, the corrupted metadata has to be repaired under client failures.

4 The FUSEE Design

4.1 Overview

As shown in Figure 4, FUSEE consists of clients, MNs, and a master. Clients provide `SEARCH`, `INSERT`, `DELETE`, and `UPDATE` interfaces for applications to access KV pairs. MNs store the replicated memory management information (MMI), hash index, and KV pairs. The master is a cluster management process responsible only for initializing clients and MNs and recovering data under client and MN failures.

FUSEE replicates both the hash index and KV pairs to tolerate MN failures. We adopt RACE hashing (Section 4.2) to index KV pairs and propose the SNAPSHOT replication protocol to enforce the strong consistency of the replicated hash index (Section 4.3). A two-level memory management scheme is adopted to efficiently allocate and replicate variable-sized KV pairs (Section 4.4). FUSEE uses logs to handle the corrupted metadata under client failures and adopts an embedded operation log scheme to reduce the log maintenance overhead (Section 4.5). Other optimizations are introduced in Section 4.6 to further improve the system performance.

4.2 RACE Hashing

RACE hashing is a one-sided RDMA-friendly hash index. As shown in Figure 5, it contains multiple 8-byte slots, with each storing a pointer referring to the address of a KV pair, an 8-bit fingerprint (Fp), *i.e.*, a part of the key's hash value, and the length of the KV pair (Len) [73]. For `SEARCH` requests, a client reads the slots of the hash index according to the hash value of the target key and then reads the KV pair on MNs according to the pointer in the slot. For `UPDATE`, `INSERT`, and `DELETE` requests, RACE hashing adopts an *out-of-place modification* scheme. It first writes a KV pair to MNs and then modifies the corresponding slot in the hash index to the address of the KV pair atomically with an `RDMA_CAS`. Nevertheless, the RACE hashing only supports a single replica.

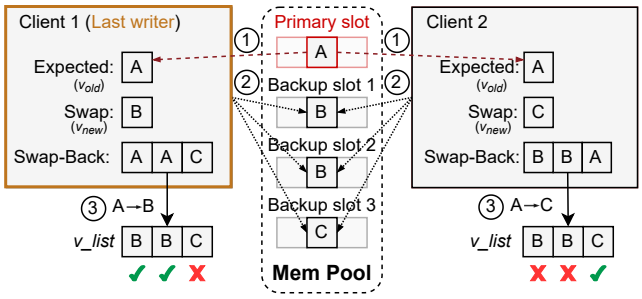


Figure 6: The SNAPSHOT replication protocol.

4.3 The SNAPSHOT Replication Protocol

In FUSEE, multiple clients concurrently read or write the same slot in the replicated hash index to execute `SEARCH` or `UPDATE` requests, as shown in Figure 6. To efficiently maintain the strong consistency of slot replicas in the replicated hash index, FUSEE proposes the SNAPSHOT replication protocol, a client-centric replication protocol that achieves linearizability without the expensive request serialization.

There are two main challenges to efficiently achieving linearizability under the fully memory-disaggregated setting. First, how to protect readers from reading incomplete states during read-write conflicts. Second, how to resolve write-write conflicts without expensively serializing all conflicting requests. To address the first challenge, SNAPSHOT splits the replicated hash index into a single primary replica and multiple backup replicas and uses backup replicas to resolve write conflicts. Hence, incomplete states during write conflicts only appear on backup replicas and the primary replica always contains the correct and complete value. Readers can simply read the contents in the primary replica without perceiving the incomplete states. To address the second challenge, SNAPSHOT adopts a last-writer-wins conflict resolution scheme similar to shared register protocols. SNAPSHOT leverages the *out-of-place modification* characteristic of RACE hashing that conflicting writers always write different values into the same slot because the values are pointers referring to KV pairs at different locations. Three conflict-resolution rules are thus defined based on the values written by conflicting writers in backup replicas, which enable clients collaboratively to decide on a single last writer under write conflicts.

Algorithm 1 shows the `READ` and `WRITE` processes of the SNAPSHOT replication protocol. Here we focus on the execution of SNAPSHOT when no failure occurs and leave the

Algorithm 1 The SNAPSHOT replication protocol

```
1:  procedure READ(slot)
2:      v = RDMA_READ_primary(slot)
3:      if v = FAIL then deal with failure
4:      return v
5:  procedure WRITE(slot, v_new)
6:      v_old = RDMA_READ_primary(slot)
7:      v_list = RDMA_CAS_backups(slot, v_old, v_new)
8:      // Change all the v_olds in the v_list to v_news.
9:      v_list = change_list_value(v_list, v_old, v_new)
10:     win = EVALUATE_RULES(v_list)   ▷ The last writer returns
        the winning rule while other writers return LOSE.
11:     if win = Rule_1 then
12:         RDMA_CAS_primary(slot, v_old, v_new)
13:     else if win ∈ {Rule_2, Rule_3} then
14:         RDMA_CAS_backups(slot, v_list, v_new)
15:         RDMA_CAS_primary(slot, v_old, v_new)
16:     else if win = LOSE then
17:         repeat
18:             sleep a little bit
19:             v_check = RDMA_READ_primary(slot)
20:             if notified failure then goto Line 24
21:         until v_check ≠ v_old
22:         if v_check = FAIL then goto Line 24
23:     else if win = FAIL then
24:         deal with failure
25:     return
```

discussion of failure handling in Section 5. We call the slots in the primary and backup hash indexes primary slots and backup slots, respectively.

For `READ` operations, clients directly read the values in the primary slots using `RDMA_READ`. For `WRITE` operations, SNAPSHOT first resolves write conflicts by letting conflicting writers collaboratively decide on a last writer with three conflict resolution rules and then let the decided last writer modify the primary slot. Figure 6 shows the process that two clients simultaneously `WRITE` the same slot. The corresponding algorithms are shown in Algorithms 1 and 2. Clients first read the value in the primary slot as v_{old} (①). Then each client modifies all backup slots by broadcasting `RDMA_CAS` operations (②) with v_{old} as the expected value and v_{new} as the swap value. On receiving an `RDMA_CAS`, the RNICs on MNs atomically modify the value in the target slot only if v_{old} matches the current value in the slot. Since all writers initiate `RDMA_CAS` operations with the same v_{old} and different v_{new}s and all backup slots initially hold v_{old}, the atomicity of `RDMA_CAS` ensures that each backup slot can only be modified once by a single writer. As a result, the values in all backup slots will be fixed after each of them has received one `RDMA_CAS` from one writer [1]. Meanwhile, since an `RDMA_CAS` returns the value in the slot before it is modified, all clients

[1]That the process that all conflicting clients broadcast `RDMA_CAS`es to modify backup slots is just like taking a snapshot, which is why the replication protocol is named SNAPSHOT.

Algorithm 2 The rule evaluation procedure of SNAPSHOT

```
1:  procedure EVALUATE_RULES(v_list, slot, v_new, v_old)
2:      v_maj = The majority value in v_list
3:      cnt_maj = The number of v_maj in v_list
4:      if FAIL ∈ v_list then
5:          return FAIL
6:      else if cnt_maj = Len(v_list) then
7:          return Rule 1 if v_maj = v_new else LOSE
8:      else if 2 * cnt_maj > Len(v_list) then
9:          return Rule 2 if v_maj = v_new else LOSE
10:     else if v_new ∉ v_list then
11:         return LOSE
12:     v_check = RDMA_READ(slot)
13:     if v_check = FAIL then
14:         return FAIL
15:     else if v_check ≠ v_old then
16:         return FINISH
17:     else if min(v_list) = v_new then
18:         return Rule 3
19:     return LOSE
```

can perceive the new values in the backup slots (③) through the return values of the broadcast of `RDMA_CAS` operations. The return values are denoted as *v_list* in Algorithm 1.

With *v_list*, SNAPSHOT defines the following three rules to let conflicting clients collaboratively decide on a last writer:

Rule 1: A client that has successfully modified all the backup slots is the last writer.

Rule 2: A client that has successfully modified a majority of backup slots is the last writer.

Rule 3: If no last writer can be decided with the former two rules, the client that has written the minimal target value (v_{new}) is considered as the last writer.

The three rules are evaluated sequentially as shown in Algorithm 2. **Rule 1** provides a fast path when there are no conflicting modifications. **Rule 2** preserves the most successful CAS operations to minimize the overhead of executing atomic operations on RNICs when conflicts are rare [30]. Finally, **Rule 3** ensures that the protocol can always decide on the last writer. To ensure the uniqueness of the last write, a client issues another `RDMA_READ` to check if the primary slot has been modified (Line 12, Algorithm 2) before evaluating **Rule 3**. If the primary slot has not been modified, then the `RDMA_CAS_backups` (Line 7, Algorithm 1) of the client must happen before the last writer modifies the primary slot. Hence, it is safe to evaluate **Rule 3** because the *v_list* must contain the value of the last writer if it has already been decided. Otherwise, **Rule 3** will not be evaluated because the modification of the primary slot means the decision of a last writer. Relying on the three rules, a unique last writer can be decided without any further network communications. For example, in Figure 6, Client 1 is the last writer according to **Rule 2**. Client 1 then modifies the backup slots that do not yet contain its proposed value using `RDMA_CAS`es and then modifies the primary slot. Other conflicting clients iteratively `READ` the value in the

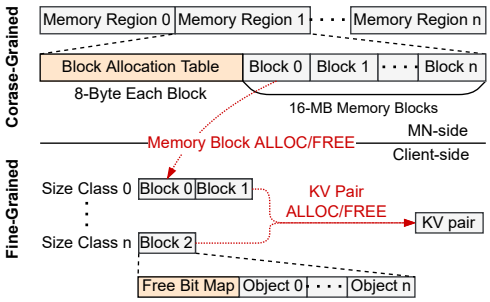


Figure 7: The two-level memory management scheme.

primary slot and return success after finding the change in the primary slot. The primary slot may remain unmodified only under the situation when the last writer crashed, which will be discussed in Section 5.

Correctness. The SNAPSHOT replication protocol guarantees linearizability of the replicated hash indexes with last-writer-wins conflict resolution like shared register protocols [7, 44]. We briefly demonstrate the correctness of SNAPSHOT using the notion of the linearizable point of KV operations. A formal proof is shown in Appendix A. A linearizable point is a point when an operation atomically takes effect in its invocation and completion [26]. For `READ`, the linearizable point happens when it gets the value in the primary slot. For `WRITE` operations, the linearizable point of the last writer happens when it modifies the primary slot. Linearizable points of other conflicting writers appear instantly before the last writer modifies the primary slot. Conflicts between readers and the last writer are resolved by RNICs because the last writer atomically modifies the primary slot using `RDMA_CAS` operations and readers access the primary slot using `RDMA_READ` operations.

Performance. SNAPSHOT guarantees a bounded worst-case latency when clients `WRITE` the hash index. Under the situation when **Rule 1** is triggered, 3 RTTs are required to finish a `WRITE` operation. Under situations when **Rule 2** or **Rule 3** is triggered, 4 or 5 RTTs are required, respectively.

4.4 Two-Level Memory Management

Memory management is responsible for allocating, replicating, and freeing memory spaces for KV pairs on MNs. As discussed in Section 3.2, the key challenge of DM management is that conducting the management tasks solely on clients or on MNs cannot satisfy the performance requirement of frequent memory allocation for KV requests. FUSEE addresses this issue via a two-level memory management scheme. The key idea is to split the server-centric memory management tasks into compute-light coarse-grained management and compute-heavy fine-grained management run on MNs and clients.

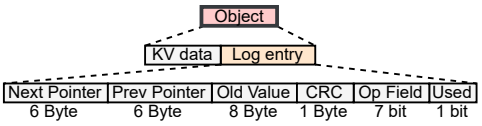
FUSEE first replicates and partitions the 48-bit memory space on multiple MNs. Similar to FaRM [18], FUSEE shards the memory space into 2GB memory regions and maps each region to r MNs with consistent hashing [33], where r is the replication factor. Specifically, consistent hashing maps a region to a position in a hash ring. The replicas are then stored at the r MNs successively following the position and the primary region is placed on the first of the r MN.

Figure 7 shows the two-level memory allocation of FUSEE. Allocating a memory space for a KV pair happens before writing the KV pair, as introduced in Section 4.1. The first level is the coarse-grained MN-side memory block allocation with low computation requirements. Each MN partitions its local memory regions into coarse-grained memory blocks, *e.g.*, 16 MB, and maintains a block allocation table ahead of each region. For each memory block, the block allocation table records a client ID (CID) that allocates it. Clients allocate memory blocks by sending `ALLOC` requests to MNs. On receiving an `ALLOC` request, an MN allocates a memory block from one of its primary memory regions, records the client ID in the block allocation tables of both primary and backup regions, and replies with the address of the memory block to the client. The coarse-grained memory allocation information is thus replicated on r MNs and can survive MN failures. The second level is the fine-grained client-side object allocation that allocates small objects to hold KV pairs. Specifically, clients manage the blocks allocated from MNs exclusively with slab allocators [6]. The client-side slab allocators split memory blocks into objects of distinct size classes. A KV pair is then allocated from the smallest size class that fits it.

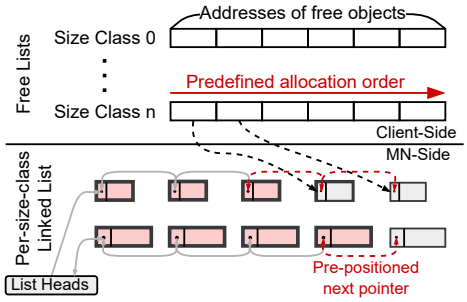
The allocated objects can be freed by any client. To efficiently reclaim freed memory objects on client sides, FUSEE stores a free bit map ahead of each memory block, as shown in Figure 7, where each bit indicates the allocation state of one object in the memory block. The free bit map is initialized to be all zeros when a block is allocated. To free an object, a client sets the corresponding bit to '1' in the free bit map with an `RDMA_FAA` operation. By reading the free bit map, clients can easily know the freed objects in their memory blocks and reclaim them locally. FUSEE frees and reclaims memory objects periodically using background threads in a batched manner to avoid the additional RDMA operations on the critical paths of KV accesses. The correctness of concurrently accessing KV pairs and reclaiming memory spaces is guaranteed by RACE hashing [73], where clients check the key and the CRC of the KV pair on data accesses.

4.5 Embedded Operation Log

Operation logs are generally adopted to repair the partially modified hash index incurred by crashed clients. Conventional operation logs record a log entry for each KV request that modifies the hash index. The log entries are generally written in an append-only manner so that the order of log entries reflects the execution order of KV requests. The recovery process can thus find the crashed request and fix the corrupted metadata by scanning the ordered log entries. However, constructing operation logs incurs high log maintenance overhead on DM because writing log entries adds remote memory accesses on the critical paths of KV requests.



(a) The embedded log entry.



(b) The organization of the embedded operation log.

Figure 8: The embedded operation log.

To reduce the log maintenance overhead on DM, FUSEE adopts an *embedded operation log* scheme that embeds log entries into KV pairs. The embedded log entry is written together with its corresponding KV pair with one `RDMA_WRITE` operation. The additional RTTs required for persisting log entries are thus eliminated. However, by embedding log entries in KV pairs, the execution order of KV requests cannot be maintained because the log entries are no longer continuous. To address this problem, the embedded operation log scheme maintains per-size-class linked lists to organize the log entries of a client in the execution order of KV requests. As shown in Figure 8b, a per-size-class linked list is a doubly linked list that links all allocated objects of the size class in the order of their allocations. The object allocation order reflects the execution order of KV requests because all KV requests that modify the hash index, *e.g.*, `INSERT` and `UPDATE`, need to allocate objects for new KV pairs. For `DELETE`, FUSEE allocates a temporary object recording the log entry and the target key and reclaims the object on finishing the `DELETE` request. FUSEE stores the list heads on MNs during the initialization of clients, which will be accessed during the recovery process of clients (Section 5).

An embedded log entry is a 22-byte data structure stored behind KV pairs, as shown in Figure 8a. It contains a 6-byte *next pointer*, a 6-byte *prev pointer*, an 8-byte *old value*, a 1-byte *CRC*, a 7-bit *opcode*, and a *used* bit. The *next pointer* points to the next object of the size class that will be allocated and the *prev pointer* points to the object allocated before the current one. The *old value* records the old value of the primary slot for recovery proposes, which will be discussed in Section 5. The 1-byte *CRC* is adopted to check the integrity of the *old value* under client failures. The *operation field* records the operation type, *i.e.*, `INSERT`, `UPDATE`, or `DELETE`, so that the crashed operation can be properly retried during recovery. The *used bit* indicates if an object is in-use or free. Storing the *used bit* at the end of the entire object can be used to check the integrity of an entire object. This is because the order-preserving nature of `RDMA_WRITE` operations ensures that the used bit is written only after all other contents in the object have been successfully written.

FUSEE efficiently organizes per-size-class linked lists by co-designing the linked list maintenance process with the memory allocation process. As shown in Figure 8b, for each size class, a client organizes the addresses of remote free objects locally as a free list. Since an object is always allocated from the head of a local free list, the allocation order of each size class is pre-determined. Based on the pre-determined order, for each allocation, a client pre-positions the *next pointer* to point to the free object in the head of the local free list and the *prev pointer* to point to the last allocated object of the size class. Both the *next pointer* and the *prev pointer* are thus known before each allocation and the entire log entry can be written to MNs with the KV pair in a single `RDMA_WRITE`.

Combined with the SNAPSHOT replication protocol, the execution process is shown as follows. First, for each writer, a log entry with an empty *old value* and *CRC* is written with the KV pair in a single `RDMA_WRITE`. Then, for the last writer of the SNAPSHOT replication protocol, the *old value* is modified to store the old value of the primary slot before the primary slot is modified. For other non-last writers, the used bits in their corresponding KV log entries are reset to '0' after finding the modification of the primary slot.

4.6 Optimizations

Adaptive index cache. Index caching is widely adopted on RDMA-based KV stores to reduce request RTTs [60, 66–68]. For a key, the index cache caches the remote addresses of the replicated index slots and the addresses of the KV pairs locally. With the cached KV pair addresses, `UPDATE`, `DELETE`, and `SEARCH` requests can read KV pairs in parallel with searching the hash index, reducing an RTT on cache hits. To guarantee cache coherence, an invalidation bit is stored together with each KV pair, which is used by clients to check whether the KV pair is valid or invalid. However, by accessing the index cache, invalid KV pairs (*e.g.*, outdated) can be fetched into clients, causing read amplification.

To attack the read amplification issue, FUSEE adaptive bypasses the index cache by distinguishing read-intensive and write-intensive keys. For each cached key, FUSEE maintains an access counter and an invalid counter which increases by 1 each time the key is accessed or found to be invalid. A client calculates an *invalid ratio* $I = \frac{\text{invalid counter}}{\text{access counter}}$ for each cached key. The index cache is bypassed when accessing a key with $I > threshold$ because the key is write-intensive and the cached key address points to an invalid KV pair with high probability. The *invalid ratio* can adapt to workload changes, *i.e.*, a write-intensive key becomes read-intensive, because the access counter of the key keeps increasing while the invalid counter stops. Besides, the adaptive scheme does not affect the `SEARCH` latency for most cases since only write-intensive keys bypass the cache.

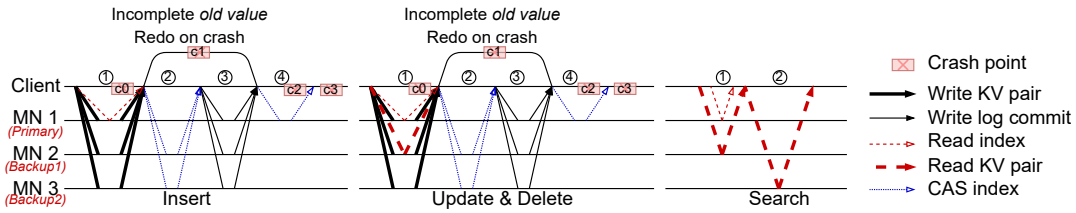


Figure 9: The workflows of different KV requests. `INSERT`: ① *write the KV pair to all replicas and read the primary index slot.* ② *CAS all backup slots.* ③ *write the old value to the log header.* ④ *CAS the primary slot.* `UPDATE` & `DELETE`: ① *write the KV pair, read the primary slot, and read the KV pair according to the index cache.* ② *CAS backup slots.* ③ *write the old value to the log header.* ④ *CAS the primary slot.* `SEARCH`: ① *read the primary slot and the KV pair according to the index cache.* ② *read the KV pair on cache misses.*

RDMA-related optimizations. KV requests require multiple remote memory accesses. FUSEE adopts doorbell batching and selective signaling [30] to reduce RDMA overhead. Figure 9 shows the procedures for executing different KV requests. Each request consists of multiple phases with multiple network operations. For each phase, FUSEE adopts doorbell batching [30] to reduce the overhead of transmitting network operations from user space to RNICs and selective signaling to reduce the overhead of polling RDMA completion queues. Consequently, each phase only incurs 1 network RTT. For `INSERT`, `DELETE`, and `UPDATE` requests, four RTTs are required in general cases. For `SEARCH` requests, at most two RTTs are required and only one RTT is required in the best case due to the index cache.

5 Failure Handling

Similar to existing replication protocols [34, 59, 62], FUSEE relies on a fault-tolerant master with a lease-based membership service [24] to handle failures. The master maintains a membership lease for both clients and MNs so that clients always know alive MNs by periodically extending their leases. The failures of both clients and MNs can be detected by the master when they no longer extend their leases. Master crashes are handled by replicating the master with state machine replication [24, 59, 62]. We formally verify FUSEE in TLA+ [36] for safety and absence of deadlocks under MN failures and the detailed illustration can be found in Appendix A.

5.1 Failure Model

We consider a partially synchronous system where processes, *i.e.*, clients and MNs, are equipped with loosely synchronized clocks [20, 24, 34]. FUSEE assumes *crash-stop* failures, where processes, *i.e.*, clients and MNs, may fail due to crashing and their operations are non-Byzantine.

Under this failure model, FUSEE guarantees linearizable operations, *i.e.*, each KV operation is atomically committed in a time between its invocation and completion [26]. All the objects of FUSEE are durable and available under an arbitrary number of client crashes and at most $r-1$ MN crashes, where r is the replication factor.

5.2 Memory Node Crashes

MN crashes lead to failed accesses to KV pairs and hash slots. For accesses to KV pairs, clients can access the backup replicas according to the consistent hashing scheme.

The complication comes from the unavailable primary and backup slots that affect the normal execution of index `READ` and `WRITE` operations. FUSEE relies on the fault-tolerant master to execute operations on clients' behalves under MN failures. We first introduce how clients `READ`/`WRITE` the replicated slots and then introduce the master's operations.

When executing index `WRITE` under MN crashes, FUSEE allows the last writer decided by the SNAPSHOT replication protocol to continue modifying all *alive* slots to the same value. Other writers send RPC requests to the master and wait for the master to reply with a correct value in the replicated slots. Under situations when no last writer can be decided, the master decides the last writer and modifies all the index slots on behalf of clients. For `READ` operations, executions are not affected under the following two cases. First, if the primary slot is still alive, clients can read the primary slot normally. Second, if the primary slot crashes, clients read all *alive* backup slots. If all *alive* backup slots contain the same value, reading this value is safe because there are no write conflicts. Otherwise, clients use RPCs and rely on the master to return a correct value for the crashed slot. Since `READ` operations are only affected under write conflicts, most `READ` can continue under the read-intensive workloads that dominate in real-world situations [9, 71].

On detecting MN crashes, the master first blocks clients from further modifying the crashed slots with the lease expiration. The master then acts as a representative last writer that modifies all *alive* slots to the same value. Specifically, the master selects a value v in an *alive* backup slot and modifies all *alive* slots to v. Since the SNAPSHOT protocol modifies the backup slots before the primary slot, the values in the backup slots are always newer than the primary slot. Hence, the master choosing a value from a backup slot is correct because it proceeds the conflicting write operations. In cases where all backup slots crash, the master selects the value in the primary slot. Clients that receive old values from the master retry their write operations to guarantee that their new value is written. The master then writes the *old value* in the operation log header to prevent clients from redoing operations when recovering from crashed clients (Section 5.3). Finally, the master reconfigures new primary and backup slots and returns the selected value to all clients that wait for a reply. After the

reconfiguration of the primary and backup slots, all KV requests can be executed normally without involving the master. During the whole process, only accesses to the crashed slots are affected and the blocking time can be short thanks to the microsecond-scale membership service [24].

5.3 Client Crashes

Crashed clients may result in two issues. First, their allocated memory blocks remain unmanaged, causing memory leakage. Second, other clients may be unable to modify a replicated index slot if the crashed client is the last writer. The master uses embedded operation logs to address these two issues.

The recovery process is executed in the compute pool and consists of two steps, *i.e.*, memory re-management and index repair. Memory re-management restores the coarse-grained memory blocks allocated by the client and the fine-grained object usage information of the client. The recovery process first gets all memory blocks managed by the crashed client by letting MNs search for their local block allocation tables. Then the recovery process traverses the per-size-class linked lists to find all used objects and log entries. With the used objects and the allocated memory blocks, the recovery process can easily restore the free object lists of the crashed client. Hence, all the memory spaces of the crashed client are re-managed.

The index repair procedure then fixes the partially modified hash index. FUSEE deems all requests at the end of per-size-class linked lists as potentially crashed requests. For incomplete log entries, *i.e.*, the *used bit* at the end of the log entry is not set, the client must crashes during writing the KV pair (c0 in Figure 9). The object is directly reclaimed without further operation since the writing of the object has not been completed. For a log entry with an incomplete *old value* according to the *CRC* field, FUSEE redoes the request according to the *operation field* and the KV pair. Under this situation, either the request belongs to the last writer that crashed before committing the log (c1 in Figure 9), or it belongs to other non-last writers. In the first case, the values in the backup slots may not be consistent and the primary slot has not been modified to a new value. Redoing the request can make the backup and primary slots consistent. In the second case, since the request of crashed non-last writers has not returned to clients, redoing the request does not violate the linearizability. For a request with a complete *old value*, the request must belong to a last writer. However, the request may finish (c3) or crash before the primary slot is modified (c2). The recovery process checks the value in the primary slot (v_p) and the value in the *old value* (v_{old}) to distinguish c2 from c3. If $v_p = v_{old}$, the request crashed before the primary was modified because v_{old} records the value before index modification. Since all backup slots are consistent, the recovery process modifies the primary slot to the new value and finishes the recovery. Otherwise, the request is finished and no further operation is required. After recovering the request, the master asynchronously checks content in the v_{old}s in log entries of the crashed client to recover its batched free operations.

5.4 Mixed Crashes

In situations where clients and MNs crash together, FUSEE recovers the failures separately. FUSEE first lets the master recover all MN crashes and then starts the recovery processes for failed clients. KV requests can proceed because the master acts as the last writer for all blocked KV requests. No request is committed twice because the master commits the operation logs on clients' behalves.

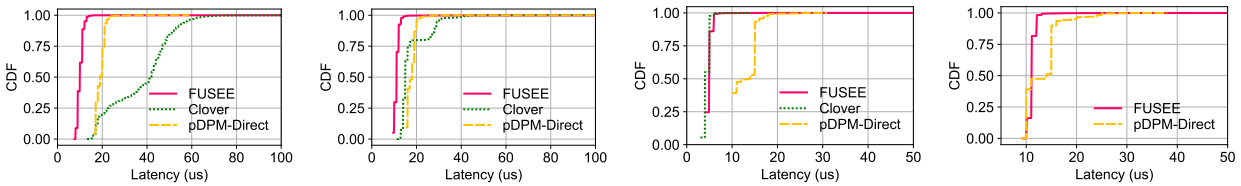
6 Evaluation

6.1 Experiment Setup

Implementation. We implement FUSEE from scratch in C++ with 13k LOC. We implement RACE hashing carefully according to the paper due to no available open-source implementations. Coroutines are employed on clients to hide the RDMA polling overhead, as suggested in [31, 73]. The design of FUSEE is agnostic to the lower-level memory media of memory nodes, *i.e.*, any memory node with either persistent memory (PM) or DRAM that provides `READ`, `WRITE`, and 8-byte `CAS` interfaces is compatible. We adopt monolithic servers with RNICs and DRAM to serve as MNs like Clover [60] since we do not have access to smartNICs and PM. Specifically, we start an MN process on a monolithic server to register RDMA memory regions and serve memory allocation RPCs with a UDP socket. MN processes serve memory allocation requests with UDP sockets. Since the socket *receive* is a blocking system call, the process will be in the blocked state with no CPU usage most of the time.

Testbed. We run all experiments on 22 physical machines (5 MNs and 17 CNs) on the APT cluster of CloudLab [19]. Each machine is equipped with an 8-core Intel Xeon E5-2450 processor, 16GB DRAM, and a 56Gbps Mellanox ConnectX-3 IB RNIC. These machines are interconnected with 56Gbps Mellanox SX6036G switches.

Comparison. We compare FUSEE with two state-of-the-art KV stores on DM, *i.e.*, pDPM-Direct and Clover [60]. pDPM-Direct stores and manages the KV index and memory space on the clients. It uses a distributed consensus protocol to ensure metadata consistency and locks to resolve data access conflicts. We extend the open-source version of pDPM-Direct to support string keys for fair comparison in our evaluation. Clover is a semi-disaggregated KV store that adopts monolithic servers to manage memory spaces and a hash index. All `UPDATE` and `INSERT` requests have to go through the metadata server, requiring additional compute power. For both pDPM-Direct and Clover, client-side caches are enabled following their default settings. To show the effectiveness of SNAPSHOT and the adaptive index cache, we implement FUSEE-CR and FUSEE-NC, two alternative versions of FUSEE. FUSEE-CR replicates index modifications by sequentially `CAS`ing all replicas to enforce sequential accesses. FUSEE-NC is the version of FUSEE without a client-side



(a) `INSERT` latency CDF. (b) `UPDATE` latency CDF. (c) `SEARCH` latency CDF. (d) `DELETE` latency CDF.

Figure 10: The CDFs of different KV request latency under the microbenchmark.

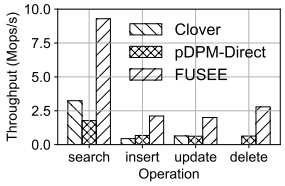


Figure 11: The throughputs of microbenchmark.

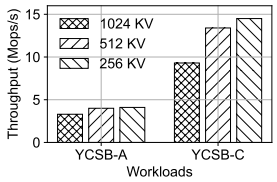


Figure 12: The throughput of FUSEE under different KV sizes.

cache. For all these methods, we evaluate their throughput and latency with both micro and YCSB [15] benchmarks.

Since the open-source version of Clover and pDPM-Direct only support one index replica, we compare FUSEE with these two approaches with a single index replica and two data replicas in the microbenchmark (Section 6.2) and YCSB performance (Section 6.3) evaluations. When evaluating FUSEE with a single index replica, the embedded log is constructed, but the commit of the log is skipped since committing the log is used to ensure the consistency of multiple index replicas. The performance of FUSEE with multiple replicas is evaluated in the fault-tolerance evaluation (Section 6.4).

6.2 Microbenchmark Performance

We use microbenchmarks to evaluate the operation throughput and latency of the three approaches. For FUSEE and pDPM-Direct, we use 16 CNs and 2 MNs. For Clover, we use 17 CNs and 2 MNs because it needs an additional metadata server, consuming 8 more CPU cores and an additional RNIC. We do not use multiple metadata servers for Clover because the current open-source implementation of Clover only supports a single metadata server. We run 128 client processes on the 16 CNs, where each CN holds 8 clients. The `DELETE` of Clover is not tested because Clover does not support it.

Latency. To evaluate the latency of KV requests, we use a single client to iteratively execute each operation 10,000 times. Figure 10 shows the cumulative distribution functions (CDFs) of the request latency. FUSEE performs the best on `INSERT` and `UPDATE`, since the SNAPSHOT replication protocol has bounded RTTs. FUSEE has a little higher `SEARCH` latency than Clover since FUSEE reads the hash index and the KV pair in a single RTT, which is slower than only reading the KV pair in Clover. FUSEE has slightly higher `DELETE` latency than pDPM-Direct because FUSEE writes a log entry and reads the hash index in a single RTT, which is slower than just reading the hash index in pDPM-Direct.

Throughput. Figure 11 shows the throughput of the three approaches. The throughput of pDPM-Direct is limited by its remote lock, which causes extensive lock contention as the number of clients grows. For Clover, even though it consumes more hardware resources, *i.e.*, 8 additional CPU cores and an RNIC, the scalability is still lower than FUSEE. This is because the CPU processing power of the metadata server bottlenecks its throughput. On the contrary, FUSEE improves the overall throughput by eliminating the computation bottleneck of the metadata server and efficiently resolving conflicts with the SNAPSHOT replication protocol.

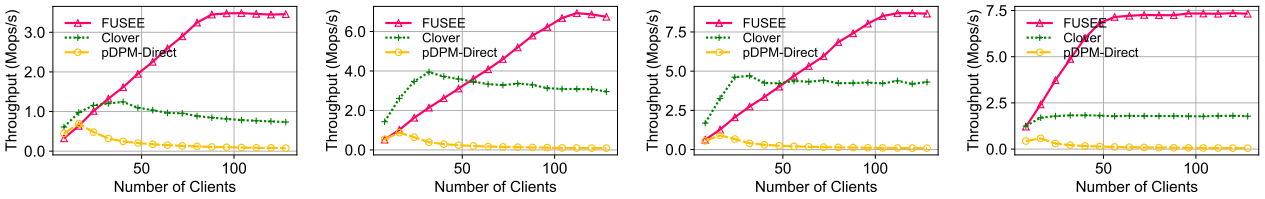
6.3 YCSB Performance

For YCSB benchmarks [15], we generate 100,000 keys with the Zipfian distribution ($\theta = 0.99$). We use 1024-byte KV pairs, which is representative of real-world workloads [9, 15, 17]. The hardware setup is the same as microbenchmarks.

YCSB Throughput. Figure 13 shows the throughput of three approaches with different numbers of clients. Clover performs the best under a small number of clients since adopting the metadata server simplifies KV operations. Compared with Clover, pDPM-Direct and FUSEE require more RDMA operations to resolve index modification conflicts. As the number of clients grows, the throughput of Clover and pDPM-Direct does not increase because the throughput is bottlenecked by the metadata server and the lock contention, respectively. Compared with Clover, FUSEE scales better with the growing number of clients while consuming fewer resources. Compared with pDPM-Direct, FUSEE improves the throughput by avoiding lock contention. When the number of clients reaches 128, the throughput of FUSEE is 4.9× and 117× higher than Clover and pDPM-Direct, respectively.

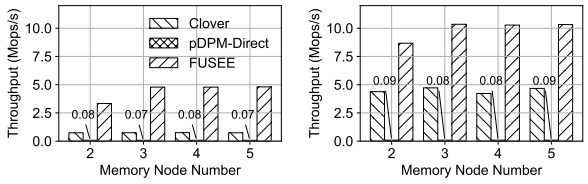
Figure 14 shows the throughput of the three approaches with a write-intensive workload (YCSB-A) and a read-intensive workload (YCSB-C) when varying numbers of MNs from 2 to 5 using 128 clients. The throughput of pDPM-Direct and Clover does not increase due to being limited by lock contention and the limited compute power of the metadata server, respectively. As for FUSEE, the throughput improves as the number of memory nodes increases from 2 to 3. There is no further throughput improvement because the total throughput is limited by the number of compute nodes.

Figure 12 shows the throughput of FUSEE under smaller KV sizes. Since the throughput of FUSEE is limited by the bandwidth of MN-side RNICs, the YCSB-C throughput of FUSEE increases by 44.1% and 55.9% with 512B and 256B



(a) A (SEARCH:UPDATE = 0.5:0.5). (b) B (SEARCH:UPDATE = 0.95:0.05). (c) C (100% SEARCH). (d) D (SEARCH:INSERT = 0.95:0.05).

Figure 13: The scalability of FUSEE under different YCSB workloads.



(a) YCSB-A throughput. (b) YCSB-C throughput.

Figure 14: The throughput with different numbers of MNs.

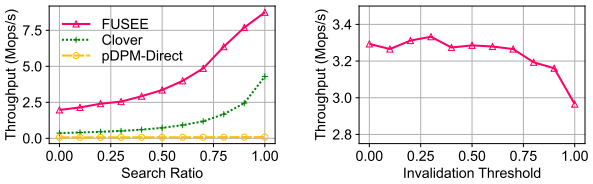


Figure 15: Throughput under different SEARCH-UPDATE ratios.

Figure 16: Throughput under different adaptive cache thresholds.

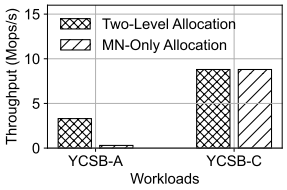


Figure 17: The throughput of different memory allocation methods.

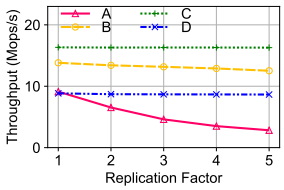


Figure 18: YCSB throughput under different replication factors.

KV pairs, respectively. The performance of FUSEE is not affected by the dataset size because the performance depends only on the number of RTTs of KV requests, which is deterministic as presented in Section 4.

Read-write performance. Figure 15 shows the throughput of the three approaches under different SEARCH-UPDATE ratios. As the portion of UPDATE grows, the throughput of all three methods decreases because UPDATE requests involve more RTTs. However, FUSEE exhibits the best throughput due to eliminating the computation bottleneck of metadata servers.

Adaptive index cache performance. Figure 16 shows the YCSB-A throughput of FUSEE with different adaptive index cache thresholds. The throughput of FUSEE decreases with the increasing thresholds because more bandwidth is wasted on fetching invalidated KV pairs with a high threshold.

Two-level memory allocation performance. To show the effectiveness of the two-level memory allocation scheme, we compare FUSEE with an MN-centric memory allocation scheme, as shown in Figure 17. The YCSB-A throughput drops 90.9% due to the limited compute power on MNs, while the YCSB-C throughput remains the same since no memory allocation is involved in the read-only setting.

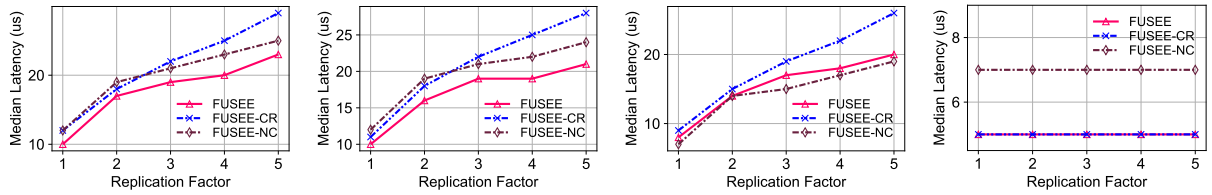
6.4 Fault Tolerance & Elasticity

SNAPSHOT Replication Protocol. Figure 19 shows the median latency of FUSEE, FUSEE-NC, and FUSEE-CR with different replication factors under microbenchmarks. We set both the numbers of index replicas and data replicas to r where r is the replication factor. The latency of FUSEE-CR on INSERT, UPDATE, and DELETE grows linearly as the replication factor because it modifies index replicas sequentially, and the number of RTTs equals the replication factor. Differently, the latency of FUSEE grows slightly with the replication factor because SNAPSHOT has a bounded number of RTTs. For SEARCH requests, FUSEE and FUSEE-CR have comparable latency since they execute SEARCH similarly. Compared with FUSEE-NC, FUSEE has lower latency for UPDATE, DELETE, and SEARCH due to fewer RTTs. The INSERT latency is slightly higher than that of FUSEE-NC because FUSEE spends additional time to maintain the local cache. Figure 18 shows the throughput of FUSEE under different replication factors. For YCSB-A and YCSB-B, the throughput drops as the replication factor grows. The YCSB-D throughput slightly drops from 8.8 Mops to 8.6 Mops due to the read-intensive nature of YCSB-D. The YCSB-C throughput remains the same due to no index modifications.

Search under Crashed MNs. FUSEE allows SEARCH requests to continue when MNs crash under read-intensive workloads. Figure 20 shows the throughput of 9 seconds of execution, where memory node 1 crashes at the 5th second. The overall throughput drops to half of the peak throughput because all data accesses come to one MN. The throughput is then limited by the network bandwidth of a single RNIC.

Recover from Crashed Clients. To evaluate the efficiency of a client recovering from failures, we crash and recover a client after UPDATE 1,000 times. As shown in Table 1, FUSEE takes 177 milliseconds to recover from a client failure. The memory registration and connection re-establishment account for 92% of the total recovery time. The log traversal and KV request recovery only account for 4% of the recovery time, which implies the affordable overhead of log traversal.

Elasticity. FUSEE supports dynamically adding and shrinking clients. We show the elasticity of FUSEE by dynamically adding and removing 16 clients when running the YCSB-C workload. As shown in Figure 21, the throughput increases when the number of clients increases from 16 to 32 and re-



(a) UPDATE median latency. (b) DELETE median latency. (c) INSERT median latency. (d) SEARCH median latency.

Figure 19: Median operation latency of FUSEE, FUSEE-NC and FUSEE-CR under different replication factors.

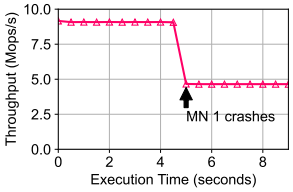


Figure 20: YCSB-C throughput under a crashed memory node.

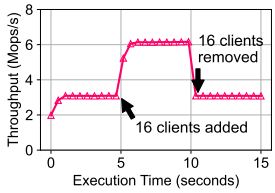


Figure 21: The elasticity of FUSEE.

Table 1: Client recovery time breakdown.

Step	Time (ms)	Percentage
Recover connection & MR	163.1	92.1%
Get Metadata	0.3	0.2%
Traverse Log	3.5	2.0%
Recover KV Requests	3.5	2.0%
Construct Free List	6.6	3.7%
Total	177.0	100%

sumes to the previous level after removing 16 clients.

7 Related Work

Disaggregated Memory. Existing approaches can be classified into software-based, hardware-based, and co-design-based memory disaggregation. Software-based approaches hide the DM abstraction by modifying operating systems [3, 23, 48, 56, 61], virtual machine monitors [42], or runtimes [55, 63]. Hardware-based ones design memory buses [14, 41] to enable efficient remote memory access. Co-design-based approaches co-design software and hardware [8, 25, 39, 65] to gain better application throughput and latency on DM. The design of FUSEE is agnostic to the low-level implementations of all these DM approaches.

Disaggregated Memory Management. MIND [39] and Clio [25] are the two state-of-the-art memory management approaches on DM. But they both rely on special hardware to manage memory spaces. The two-level memory management of FUSEE resembles the hierarchical memory management of The Machine [21, 35]. The difference is that FUSEE focuses on fine-grained KV allocation with commodity RNICs, while The Machine relies on special SoCs and directly manages physical memory devices.

Memory-disaggregated KV stores. Clover [60] and Dinomo [38] are the most related memory-disaggregated KV stores. Compared with Clover [60], FUSEE brings disaggregation to metadata management and gains better resource efficiency and scalability. Finally, Dinomo [38] is a fully-disaggregated KV store that was developed concurrently with our system. Dinomo proposes ownership partitioning to reduce coordination overheads of managing disaggregated metadata. However, it assumes that the disaggregated memory pool is fault-tolerant, and hence its design does not consider MN failures. In contrast, FUSEE addresses the challenges of handling MN failures with the SNAPSHOT replication protocol. There are many related RDMA-based KV stores [18, 29, 31, 32, 46, 49, 52, 57, 60, 66–68]. They are infeasible on DM since they rely on server-side CPUs to execute KV requests. Besides, there are emerging approaches that use SmartNICs to construct KV stores [40, 53]. FUSEE can also benefit from the additional compute power by offloading the memory management to SmartNICs.

Replication. Both traditional [2, 22, 37, 44, 47, 51, 59, 62] and RDMA-based [34, 58, 72] replication protocols are designed to ensure data durability. However, all these approaches are server-centric replication protocols designed for monolithic servers. Differently, SNAPSHOT is a client-centric replication protocol designed for the DM architecture and achieves high scalability with collaborative conflict resolution.

8 Conclusion

This paper proposes FUSEE, a fully memory-disaggregated KV store, that achieves both resource efficiency and high performance by disaggregating metadata management. FUSEE adopts a client-centric replication protocol, a two-level memory management scheme, and an embedded log scheme to attack the challenges of weak MN-side compute power and complex failure situations on DM. Experimental results show that FUSEE outperforms the state-of-the-art approaches by up to 4.5× with less resource consumption.

Acknowledgments

We sincerely thank our shepherd Kimberly Keeton and the anonymous reviewers for their constructive comments and suggestions. This work was supported by the National Natural Science Foundation of China (Nos. 62202511 & 61971145), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210920 of the General Research Fund), and Huawei Cloud. Pengfei Zuo is the corresponding author (pfzuo.cs@gmail.com).

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[2] Peter Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*, pages 562–570. IEEE Computer Society, 1976.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[4] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 49–60. IEEE Computer Society, 2017.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pages 363–375. ACM, 1990.

[6] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, June 6-10, 1994, Conference Proceeding*, pages 87–98. USENIX Association, 1994.

[7] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 591–617. USENIX Association, 2020.

[8] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 79–92. ACM, 2021.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 209–223. USENIX Association, 2020.

[10] cgroups. cgroups. https://man7.org/linux/man-pages/man7/cgroups.7.html, 2022.

[11] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 19:1–19:14. ACM, 2019.

[12] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 539–551. USENIX Association, 2017.

[13] Zhiguang Chen, Yu-Bo Liu, Yong-Feng Wang, and Yutong Lu. A gpu-accelerated in-memory metadata management scheme for large-scale parallel file systems. *J. Comput. Sci. Technol.*, 36(1):44–55, 2021.

[14] Gen-Z Consortium. Gen-z technology. https://genzconsortium.org/.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[16] Intel Corporation. Driving exascale computing and hpc with intel. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html.

[17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 33–49. USENIX Association, 2021.

[18] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[20] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[21] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan S. Milojicic. Beyond processor-centric operating systems. In George Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[22] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[24] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, 2022. USENIX Association.

[25] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. *CoRR*, abs/2108.03492, 2021.

[26] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[27] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[28] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, 2019.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306. ACM, 2014.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16. USENIX Association, 2019.

[33] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 654–663. ACM, 1997.

[34] Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 201–217. ACM, 2020.

[35] HP Labs. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2014.

[36] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[37] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[38] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022.

[39] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[40] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 137–152. ACM, 2017.

[41] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 267–278. ACM, 2009.

[42] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[43] Compute Express Link. Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/.

[44] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 272–281. IEEE Computer Society, 1997.

[45] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[46] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.

[47] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[48] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 16:1–16:12. ACM, 2018.

[49] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 97–108. ACM, 2019.

[50] Brian M. Oki and Barbara Liskov. Viewstamped replication: A general primary copy. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, pages 8–17. ACM, 1988.

[51] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[52] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[53] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodík, and Thomas E. Anderson. Floem: A programming system for nic-accelerated network applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 663–679. USENIX Association, 2018.

[54] Kai Ren, Qing Zheng, Swapnil Patil, and Garth A. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In Trish Damkroger and Jack J. Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 237–248. IEEE Computer Society, 2014.

[55] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[56] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[57] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: when RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 1–15. ACM, 2017.

[58] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 851–863. USENIX Association, 2018.

[59] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*. USENIX Association, 2009.

[60] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[61] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324. ACM, 2017.

[62] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 91–104. USENIX Association, 2004.

[63] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 261–280. USENIX Association, 2020.

[64] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. *CoRR*, abs/2112.07320, 2021.

[65] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[66] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 117–135. USENIX Association, 2020.

[67] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[68] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[69] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 307–320. USENIX Association, 2006.

[70] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, 2021.

[71] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[72] Yiying Zhang, Jian Yang, Amir Saman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 3–18. ACM, 2015.

[73] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.

A Proof of Correctness

Since SNAPSHOT focuses on the linearizability of replicated index slots, it is equivalent to proving that SNAPSHOT is correct on a single replicated slot. The proof consists of four parts. First, we formally define the system model and the correctness standard, *i.e.*, linearizability. We then specify the behavior of a legal replicated slot. After that, we formally prove that SNAPSHOT is correct under failure-free executions. Finally, we illustrate that SNAPSHOT is correct when failures occur with a fault-tolerant management master.

A.1 Formal System Model

Formally, the system is comprised of a set C of *clients* $\{c_1,...,c_n\}$ and a set S of *replicated slots* $\{s_1,...,s_r\}$, where n is the number of clients and r is the replication factor. Each slot s_i stores a value denoted by v_{s_i}. Without loss of generality, we assume s_1 is the primary slot and $\{s_2,...,s_r\}$ are backup slots. *Clients* can access the contents in the *replicated slots* via `RDMA_READ`(s_i), and `RDMA_CAS`(s_i, v_{old}, v_{new}) operations. `RDMA_READ`(s_i) returns the content in slot s_i. `RDMA_CAS`(s_i, v_{old}, v_{new}) atomically compares v_{s_i} with v_{old} and updates $v'_{s_i} = v_{new}$ if $v_{s_i} = v_{old}$. All these operations are synchronous and reliable, *i.e.*, client-initiated RDMA operations will not be re-ordered and operations reply a `FAIL` state only if the corresponding slot crashes.

Both *clients* and *slots* may fail according to the *crash-stop model*: a failed client stops executing instructions and a failed slot returns `FAIL` to clients when performing RDMA operations. Slots are shared memory that provide only `RDMA_READ` and `RDMA_CAS` operations for *clients* to manipulate its content. Clients are state machines that deterministically transition between states when events occur. Each operation op contains two events, $inv(op)$ indicating the invocation of the operation and $resp(op)$ indicating the completion of the operation.

We use the definitions of *history* from Herlihy and Wing [27] to facilitate our proof. A history h of an execution e is an infinite sequence of operation invocation and response events in the same order as they appear in e. We denote by $ops(h)$ the set of all operations whose invocations appear in h.

A history h is *sequential* if (1) the first event of h is an invocation and (2) each invocation, except the last is immediately followed by a matching response and each response is immediately followed by an invocation. We use $h|o$ to denote an object subhistory, where all events in h occurred at object o. A set S of histories is *prefix-closed* if whenever h is in S, every prefix of h is also in S. A *sequential specification* of an object o is a prefix-closed set of single-object sequential histories for o. A sequential history h is legal if $\forall o \in O : h|o$ belongs to the sequential specification for o. A history induces an irreflexible partial order on $ops(h)$, denoted as $<_h$, where $op_1 <_h op_2$ if and only if $resp(op_1) < inv(op_2)$ in h.

A.2 Replicated Slot

A replicated slot is a data type that supports the following operations:

- READ(): returns the value of the replicated slot.
- WRITE(v): updates the value of the slot to v.

Different from traditional replicated objects, the values different clients write to a replicated slot are not duplicated, which is guaranteed by the out-of-place modification scheme of FUSEE. We use $reads(h)$ and $writes(h)$ to denote the set of all operations that reads or writes in $ops(h)$ respectively.

Definition 1 (Replicated Slot Specification)**.** *A sequential object history $h|o$ belongs to the sequential specification of a replicated slot if for each $op \in reads(h|o)$ such that $resp(op) \in h|o$, $resp(op)$ contains the value of the latest preceding operation $u \in write(h|o)$ or if there is no preceding update, then $resp(op)$ contains the initial value of o.*

A.3 Proof of Linearizability

In this subsection, we prove that the failure-free executions of SNAPSHOT satisfy linearizability. The proof is based on Algorithm 1 and Algorithm 2. Linearizability [26] is defined as follows.

Definition 2 (Linearizability)**.** *A complete history h satisfies linearizability if there exists a legal total order τ of $ops(h)$ such that $\forall op_1, op_2 \in ops(h), op_1 <_h op_2 \implies op_1 <_\tau op_2$.*

Proof Sketch. We prove the linearizability of SNAPSHOT by first attaching a *virtual label* to the replicated slot. Then we formally define the rules to update the slot virtual label. We assign each READ or WRITE operation an operation label according to the slot virtual label. Finally, we define a total order relationship on the operation virtual label and prove that the label total order satisfies τ in Definition 2.

Notations Without further specification, we use var^{op} and $func^{op}$ to denote the variable var and the function call $func$ in the execution of the algorithms of op.

More Definitions. We first give more definitions to facilitate our proof.

Definition 3. *A complete operation $op \in reads(h)$ observes an update $u \in writes(h)$ if the value returned in $resp(op)$ was written by u.*

Definition 4 (Write Winner)**.** *An $op \in writes(h)$ is a write winner, denoted by $Win(op) = True$, if $win^{op} \in \{\texttt{Rule\_1}, \texttt{Rule\_2}, \texttt{Rule\_3}\}$ in Line 10 of Algorithm 1.*

Definition 5 (Slot Virtual Label)**.** *A virtual label of a replicated slot is of the form $(u,v,w) \in \mathbb{N}^3$. The initial virtual slot label is $(0,0,0)$.*

Definition 6 (Slot Virtual Label Update Rules)**.** *For a label with value (u,v,w) its update rules are defined as follows:*

- *$(u,v,w) \leftarrow (u,v,w+1)$ for each `RDMA_READ` operation in Line 2 of Algorithm 1.*
- *$(u,v,w) \leftarrow (u+1,0,0)$ for each `RDMA_CAS` that modifies the primary slot in Line 12 and Line 15 of Algorithm 1.*

Definition 7 (Operation Virtual Label)**.** *For each operation $op \in ops(h)$, its virtual label $L_{op} = (u,v,w) \in \mathbb{N}^3$ is assigned as follows:*

- *If $op \in reads(h)$, $L_{op} = (u,0,w+1)$, where (u,v,w) is the slot virtual label when it initiates the `RDMA_READ` in Line 2 of Algorithm 1.*
- *If $op \in writes(h), Win(op) = True$, then $L_{op} = (u+1,0,0)$, where (u,v,w) is the slot virtual label when it initiates the `RDMA_READ` in Line 6 of Algorithm 1.*
- *If $op \in writes(h), Win(op) = False$, then $L_{op} = (u,1,c_i)$, where (u,v,w) is the slot virtual label when it initiates the `RDMA_READ` in Line 6 of Algorithm 1 and c_i is the client that performs the operation.*

Definition 8 (Operation Label Order)**.** *Two operation virtual labels $L_1 = (u_1,v_1,w_1), L_2 = (u_2,v_2,w_2)$, $L_1 <_l L_2$ if and only if $u_1 < u_2 \bigvee (u_1 = u_2 \bigwedge v_1 < v_2) \bigvee (u_1 = u_2 \bigwedge v_1 = v_2 \bigwedge w_1 < w_2)$.*

Definition 9 (Write Round)**.** *For an $op \in writes(h)$, its write round $R_{op} = u$, where u is the label (u,v,w) when Line 6 of Algorithm 1 is executed.*

Definition 10 (Round Start Time)**.** *The start time of a write round k is defined as $t^k_{start} = inv(op) : \forall op_j \in writes(h) : R(op) = R(op_j) \implies inv(op) \leq inv(op_j)$.*

Lemma 1. *Values in the primary slot is uniquely associated with a write round.*

Proof. By Definition 6 and Line 12 and Line 15 of Algorithm 1. The primary slot is modified with different values and with a strictly monotonic u, where (u,v,w) is its virtual label. □

Lemma 2 (At least one winner)**.** *$\forall$ write round k if $v_{s_1} = ... = v_{s_r}$ at t^k_{start}, then $\exists op \in writes(h) : R(op) = k \bigwedge win(op) = True$.*

Proof. Since $v_{s_1} = ... = v_{s_r}$, by Lemma 1 $\forall op \in writes(h) : R(op) = k, v^{op}_{old} = v_{s_1}$. Since values in the backup slots all equals v_{s_1} and different op proposes different v^{op}_{new}, the atomicity of `RDMA_CAS` (Line 7, Algorithm 1) ensures that each backup can only be modified once and all backups must have be modified once after some op reaches Line 9. Suppose $\forall op \in writes(h) : R(op) = k \bigwedge Win(op) = False$. The primary slot will not be modified as indicated by the if condition in Line 16 of Algorithm 1, which implies that all op has $v^{op}_{check} = v_{old}$ in Line 12 of Algorithm 2. The if condition of Line 17 (Algorithm 2) must be false, and thus $\forall op : v^{op}_{new} \neq min(v\_list)$, which is impossible because all the backup slot must have be modified by some op and v^{op}_{new} are distinct. □

Lemma 3 (At most one winner)**.** *$\forall$ write round k if $v_{s_1} = ... = v_{s_r}$ at t^k_{start}, then $\forall op_1, op_2 \in writes(h) : R(op_1) = R(op_2) = k \bigwedge Win(op_1) = Win(op_2) = True \implies op_1 = op_2$.*

Proof. 1. No two client can both win by `Rule 1` and `Rule 2` because (1) all backup slots can only be modified once and (2) majority cannot be overlapped because different op modifies with different v_{new}.

2. Suppose there are two winners op_1 and op_2.

CASE 1: $win^{op_1} \in \{\texttt{Rule\_1}, \texttt{Rule\_2}\} \bigwedge win^{op_2} = \texttt{Rule\_3}$.
If the finish of `RDMA_CAS_backups` (Line 7, Algorithm 1) of op_2 happens before op_1 executes `RDMA_CAS_primary` (Line 12 or Line 15, Algorithm 1), then guaranteed by the atomicity of `RDMA_CAS`, op_2 knows the majority and cannot win. If the finish of `RDMA_CAS_backups` of op_2 happens after op_1 executes `RDMA_CAS_primary`, then the `RDMA_READ` (Line 12, Algorithm 2) must return $v^{op_2}_{check} \neq v^{op_2}_{orig}$. Then $win^{op_2} = \texttt{FINISH}$ by Line 16 of Algorithm 2.

CASE 2: $win^{op_1} = win^{op_2} = \texttt{Rule 3}$.
Since there is no majority, Line 17 of Algorithm 2 indicates that $v^{op_1}_{new} = v^{op_2}_{new}$, contradicting with the assumption that no clients write duplicated value. □

Definition 11 (Round finish time)**.** *The round finish time t^k_{fini} of a write round k is the time when its only winner initiates `RDMA_CAS_primary` in Line 12 or Line 15 of Algorithm 1.*

Lemma 4. *$\forall k$ at t^k_{start}, we have $v_{s_1} = ... = v_{s_r}$.*

Proof. We prove this by induction.

1. Initially, when $k = 0$, all slots have the same value.

2. Suppose all slots have the same value at t^k_{start}. By Lemma 2 and Lemma 3, there must be a single write winner

op. Guaranteed by the if-condition in Line 11 and Line 13 of Algorithm 1, only the op with $Win(op) = True$ can further modify the replicated slot. If $win^{op} =$ `Rule_1`, then Line 7 of Algorithm 1 guarantees that all backups $v_{s_2} = ... = v_{s_r} = v^{op}_{new}$ before the initiation of `RDMA_CAS_primary` (Line 12). If $win^{op} \in \{$`Rule_2`,`Rule_3`$\}$, then the `RDMA_CAS_backups` (Line 14, Algorithm 1) guarantees that all backups $v_{s_2} = ... = v_{s_r} = v^{op}_{new}$ before the `RDMA_CAS_primary` (Line 15). Since op modifies the primary slot also to v^{op}_{new} (Line 12 or Line 15), after t^0_{fini} we have $v_{s_1} = ... = v_{s_r} = v^{op}_{new}$. By Definition 10, all modifications happens after t^{k+1}_{start}, which implies $v_{s_1} = ... = v_{s_r}$ at t^{k+1}_{start}. □

Lemma 5 (Exactly one write winner)**.** $\forall k \in \mathbb{N}$*, we have:*

- $\exists op \in writes(h) : R(op) = k, Win(op) = True.$
- $\forall op_1, op_2 \in writes(h) : R(op_1) = R(op_2) = k \bigwedge Win(op_1) = Win(op_2) = True \implies op_1 = op_2.$

Proof. By Lemma 2 + Lemma 3 + Lemma 4. □

Lemma 6 (Label order respects history)**.** $\forall op_1, op_2 \in ops(h), op_1 <_h op_2 \implies op_1 <_l op_2.$

Proof. Suppose $\exists op_1, op_2 \in ops(h), op_1 <_h op_2 \bigwedge op_2 <_l op_1$. Their labels are $l_1 = (u_1, v_1, w_1)$ and $l_2 = (u_2, v_2, w_2)$.
CASE 1: $op_1, op_2 \in reads(h)$.
If $u_1 = u_2$, then $op_1 <_h op_2$ implies that the `RDMA_READ` (Line 2, Algorithm 1) of op_1 happens before the `RDMA_READ` of op_2. By Definition 6, `RDMA_READ`s in Line 2 of Algorithm 1 monotonically increase w of the virtual slot label. Then we must have $w_1 < w_2$. Since $u_1 = u_2, v_1 = v_2 = 0, w_1 < w_2$, $op_1 <_l op_2$, contradicts with the assumption. If $u_1 > u_2$, then $\exists t^{u_2}_{fini} : resp(op_1) < t^{u_2}_{fini} < inv(op_2)$. By Definition 6, $u_1 \leq u_2$, contradicts with $u_1 > u_2$.
CASE 2: $op_1 \in reads(h), op_2 \in writes(h)$.
If $Win(op_2) = True$, then $op_1 <_h op_2$ implies that $resp(op_1) < t^{u_2-1}_{fini}$. Then $u_1 \leq u_2 - 1$ contradicts with $op_2 <_l op_1$. Otherwise $Win(op_2) = False$, then $op_1 <_h op_2$ implies that $resp(op_1) < t^{u_2}_{fini}$. Since $u_1 \leq u_2$ and $0 = v_1 < v_2 = 1$ by Definition 7, there is a contradiction with $op_2 <_l op_1$.
CASE 3: $op_1 \in writes(h), op_2 \in reads(h)$.
Line 17-22 of Algorithm 1 ensures that $\forall op \in writes(h), R(op) = k \implies resp(op) > t^k_{fini}$. If $Win(op_1) = True$, then $op_1 <_h op_2$ implies that $t^{u_1-1}_{fini} < inv(op_2)$ and $l_1 = (u_1, 0, 0)$. Consequently, $u_1 \leq u_2 \bigwedge v_1 = v_2 = 0 \bigwedge 0 = w_1 < 1 \leq w_2$ contradicts with $op_2 <_l op_1$. Otherwise $Win(op_1) = False$, then $op_1 <_h op_2$ implies that $t^{u_1}_{fini} < inv(op_2)$. Then $u_1 + 1 \leq u_2$ which contradicts with $op_2 <_l op_1$.
CASE 4: $op_1, op_2 \in writes(h)$.
If $Win(op_1) = True$, then $op_1 <_h op_2 \implies t^{u_1-1}_{fini} < inv(op_2) \implies u_1 \leq u_2$. Also $Win(op_1) = True \implies l_1 = (u_1, 0, 0)$. Then $u_1 \leq u_2, 0 = v_1 < v_2 = 1$ contradicts with $op_2 <_l op_1$. Otherwise $Win(op_1) = False$, then $op_1 <_h op_2 \implies t^{u_1}_{fini} < inv(op_2) \implies u_1 + 1 \leq u_2 \implies u_1 < u_2$, which contradicts with $op_2 <_l op_1$. □

Lemma 7 (Label order is a legal order)**.** *The label order* $<_l$ *is a legal total order of* $ops(h)$.

Proof. By Definition 1, let $r \in reads(h)$ be an operation that observes a write $k \in writes(h)$, r is completed. Suppose $\exists k' \in writes(h)$ such that $k <_l k' <_l r$. Let $l_k = (u_k, v_k, w_k), l_{k'} = (u'_k, v'_k, w'_k), l_r = (u_r, v_r, w_r)$. By Lemma 1, r observes a value if and only if it returns the corresponding virtual label in Line 2 of Algorithm 1. By Definition 6 and Lemma 5, the label is exclusively updated by a single writer, which implies $Win(k) = True, t^{u_k-1}_{fini} < inv(r)$ and $u_k = u_r$. Since $k <_l k' <_l r$ and $u_k = u_r, u_k = u'_k = u_r$. If $Win(k') = True$, then $Win(k) = Win(k') = True \bigwedge R(k) = R(k') = u_k - 1$, which contradicts with Lemma 5. Otherwise $Win(k') = False$ then by Definition 7 $v'_k = 1$. $u'_k = u_r, v'_k = 1 > 0 = v_r$ contradicts with $k' <_l r$. As a result, there is no k' such that $k <_l k' <_l r$. □

Theorem 1. *FUSEE implements a replicated slot with linearizability.*

Proof. By Lemma 7 and Lemma 6. □

Algorithm 3 The master process.

```
1:  procedure MASTER
2:      if MN failed or received client fail_query then
3:          send member_prepare_change to all clients
4:          wait all clients reply or membership lease expires
5:          select a slot s randomly in alive backups
6:          modify all slots s_i = s
7:          commit the operation log of write(s)
8:          select new primary and backup
9:          reply all clients' fail_query with a new value s
10:         send member_commit_change to all clients with
        the new membership
11:     if Clients failed then
12:         start RecoverClient(log) thread.
13: procedure RECOVERCLIENT(log)
14:     if The log is committed then
15:         return
16:     slot is the slot in the log
17:     v_old is the old value in the log
18:     v_new is the new value in the log
19:     s_0 = RDMA_READ_primary(slot)
20:     if FAIL ∈ {s_0, bk_list} then
21:         send FailReq to master.
22:     else if v_old is incomplete then
23:         retry the operation
24:     else if v_old = s_0 then
25:         RDMA_CAS_primary(slot, s_0, v_new)
26:     return
```

Algorithm 4 SNAPSHOT with failure handling

```
1:  procedure READ(slot)
2:      v = RDMA_READ_primary(slot)
3:      if v = FAIL then
4:          v_list = RDMA_READ_backups(slot)
5:          if All backups have the same value v' then
6:              v = v'
7:          else
8:              v = RPC_fail_query(master, slot)
9:              wait for membership change
10:     return v
11: procedure WRITE(slot, v_new)
12:     v_old = RDMA_READ_primary(slot)
13:     if v_old = FAIL then
14:         wait for membership change
15:         retry WRITE
16:     v_list = RDMA_CAS_backups(slot, v_old, v_new)
17:     // Change all the v_olds in the v_list to v_news.
18:     v_list = change_list_value(v_list, v_old, v_new)
19:     win = evaluate_rules(v_list)
20:     if win = Rule_1 then
21:         RDMA_CAS_primary(slot, v_old, v_new
22:     else if win ∈ {Rule_2, Rule_3} then
23:         RDMA_CAS_backups(slot, v_list, v_new)
24:         RDMA_CAS_primary(slot, v_old, v_new)
25:     else if win = LOSE then
26:         repeat
27:             sleep a little bit
28:             v_check = RDMA_READ_primary(slot)
29:             if receive member_prepare_change() then
30:                 goto Line 35
31:         until v_check ≠ v_old
32:         if v_check = FAIL then
33:             goto Line 35
34:     else if win = FAIL then
35:         v_RPC = RPC_fail_query(master, slot)
36:         wait for membership change
37:         if v_RPC = v_old then
38:             retry WRITE
39:     return
```

A.4 Correctness of Failure Recovery

The recovery of FUSEE relies on a fault-tolerant management master, which is a common assumption on membership-based replication protocols like Chain Replication [59, 62] and Hermes [34]. The master adopts a lease-based membership service [24] for clients and MNs. The lease-based membership service provides a view of all alive MNs to clients. Clients check and extend their leases before performing each *read* and *write*. The master can detect the failures of clients and MNs when a client or MN no longer extend their leases. The pseudo-code of the master is shown in Algorithm 3, and the full process of clients is shown in Algorithm 4. The key to guaranteeing correctness under failures is that the single-winner condition is not violated.

A.4.1 Handling MN Crashes

The failure handling process consists of three phases, *i.e.*, a disconnection phase (Line 2-4, Algorithm 3), a slot-modification phase (Line 5-7, Algorithm 3), and a notification phase (Line 8-10, Algorithm 3). The disconnection phase starts with the master sending a *member_prepare_change* to all clients and waiting for a lease expiration. On receiving the *member_prepare_change*, clients stop their future *write*s to the failed slot. After the lease expires, no clients can further modify the failed slot because the failed slot is excluded from the membership view. The currently executing write operations will also be stopped and wait for the final value to be decided by the master (Line 35-28, Algorithm 4). Hence, all clients are disconnected to the crashed slot.

During the modification phase, the master randomly selects a value in an alive backup slot and uses the selected value to make all alive slots consistent. Choosing values from backup slots is always safe because backup slots contain no older values than the latest committed value (value in the primary slot before it crashed). This is guaranteed by the SNAPSHOT replication protocol that the write conflicts are resolved in the backup slots and then written to the primary slot. If there are no alive backup slots, then there must have only one survivor as the primary slot. In both cases, either a latest committed value or a fresh uncommitted value (value written by conflicting writers) is chosen. When a fresh uncommitted value is chosen, the master is the representative last writer and finishes the last writer's job on a client's behalf. Since no other client can modify the slot, the master is the only last writer. When a committed value is chosen, the value will be replied to clients and clients will retry their newer WRITEs (Line 38, Algorithm 4) on receiving an old value. In this case, the new write round of clients is considered not started. The retry of *write*s guarantees that the clients' newer write operations are executed before they are returned. The single last write will then be decided among clients through the normal execution of the SNAPSHOT replication protocol. After modifying all the values in the replicated slots, the master commits the operation by writing a special value (1) to the *old value* field of the log header. This guarantees that clients will never retry an operation finished by the master.

The attribute of exactly one winner on each write round is not violated. Line 4 of Algorithm 2 ensures that a winner can only be decided when no backup slot crashes. Line 29 and Line 35-38 of Algorithm 4 ensure that all losers and failed operations wait for the decided value returned by the master. If there is a winner and the winner finishes execution before the disconnection phase of the master, then the master can only choose no older value than the winner's committed value because all old values are modified to the winner's proposed new value. If the winner does not finish execution before the disconnection phase of the master, then it will be disconnected and deemed also as a loser. In this case, the master becomes a representative winner that decides a unique winner value to all

other waiting losers. If there is no winner due to the backup slot failures, then the master will also become a representative winner that decides a unique winner.

During MN crashes, `reads` can also get the latest committed value in the slot, thus ensuring linearizability. The latest committed value is defined as the last value stored in the primary slot before it crashes. Since the primary slot is lastly modified in a write round (Line 21 and Line 24, Algorithm 4), it always contains the latest committed value. If the primary slot is alive, then `reads` execute normally by reading the content in the primary slot (Line 2, Algorithm 4). Linearizability is guaranteed by the SNAPSHOT replication protocol. Otherwise, `reads` use `RDMA_READs` to read all backup slots and return a value only if all the values in the backup slot are the same. This reflects the situation when there are no write conflicts, and hence the value must be committed. Under the situation when values in backup slots are not the same, clients send RPCs to the master to let the master commit a value and return it to clients.

A.4.2 Handling Client Crashes

FUSEE uses per-client operation logs to recover crashed client operations. The v_{old} is written to the log header by the write winner before the execution of `RDMA_CAS_primary` (Line 21 and Line 24, Algorithm 4). 0 is written to the *used bit* of the log headers before the losers return their operations. During the recovery of a crashed client, memory objects with unset *used* bits are reclaimed because they are either incomplete data or free data. Operations in the operation log with an incomplete v_{old} are redone. These operations may belong to a loser or a winner. Redoing a crashed operation of a loser is safe because the losers' operations have not been returned to clients and cannot be observed by other clients. Redoing such operations of a winner is also safe because the winner's operation also cannot be observed by other clients due to the unmodified primary slot. Under this situation, the retried operation will also become a winner due to Lemma 5.

If the *old value* field is set, then the crashed client must be a write winner since only the winner commits the log 4.5. However, for a winner, it is possible that it crashed before the primary slot was modified. Under this situation, all backup slots contain v_{new}, and the primary slot contains the v_{old}. Hence, FUSEE checks if the primary slot equals v_{old} and modifies the primary slot to be v_{new} if it is the case. Otherwise, the operation must have been finished because the primary slot is modified in a new write round.

A.4.3 Handling Client and MN Crashes

When handling concurrent client and MN failures, the master reconfigures MNs to use a decreased replication factor. After the reconfiguration, the master recovers the crashed client. The only conflict between recovering MNs and clients is the case that the master decides the value written by a client, and the client is crashed. In this situation, a winner client is crashed and ignorant of its winning. The operations may be retired, causing old values to be written to the slots. FUSEE addresses this issue by letting the master commit the operation by writing a $v_{old} = 0$ to the *old value* field of the log header. As a result, when recovering the winner client using the log, it will not execute the operation twice.