FUSEE: A Fully Memory-Disaggregated Key-Value Store

21st USENIX Conference on File and Storage Technologies

Jiacheng Shen; Pengfei Zuo; Xuchuan Luo; Tianyi Yang; Yuxin Su; Yangfan Zhou and Michael R. Lyu

Paper Notes

By JeongHa Lee

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 FUlly memory-diSaggrEgated KV StorE 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.

Problem Statement and Research Objectives

- **Client-centric index replication**: To tolerate memory node failures, clients need to replicate the index on memory nodes.
  - In existing replication approaches, 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.
- **Remote memory allocation**
  - Existing semi-disaggregated KV stores manage memory spaces with monolithic metadata servers.
  - However, in the fully memory-disaggregated setting,
    - **Memory nodes cannot handle the compute-heavy** fine-grained memory allocation for KV pairs due to their poor compute power.
    - **Clients cannot efficiently allocate** memory spaces because **multiple RTTs are required to modify the memory management information** stored on memory nodes.
- **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, in the fully memory-disaggregated setting,
    - **Clients directly access and modify metadata** on memory nodes. ➔ client failures can leave **partially modified metadata**

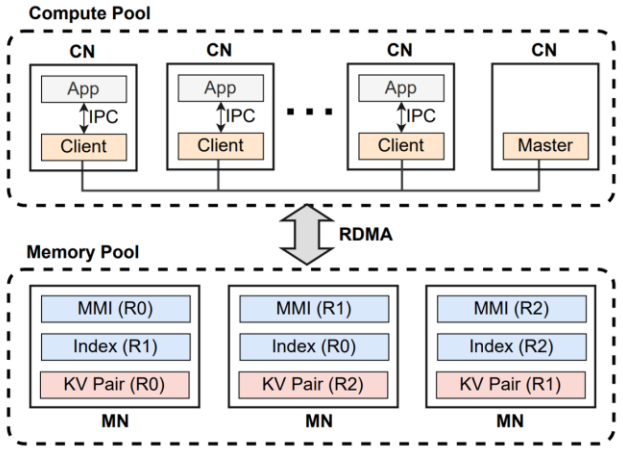


Figure 4: The FUSEE overview *(MMI, Index, and KV pairs have multiple replicas, i.e., R_0, R_1, and R_2. R_0 is the primary replica.).*

Proposed Method

1. SNAPSHOT replication protocol
   - to maintain the strong consistency of the replicated index
   - The key to achieving scalability is to **resolve write conflicts without** involving the expensive **request serialization**.
     ① Clients **first read the value** in the primary slot as v_{old}.
     ② Then each client **modifies all backup slots** by broadcasting `RDMA_CAS` operations with v_{old} as the expected value and v_{new} as the swap value.
     ③ Since an `RDMA_CAS` returns the value in the slot before it is modified, **all clients can perceive the new values in the backup slots** through the return values of the broadcast of `RDMA_CAS` operations.

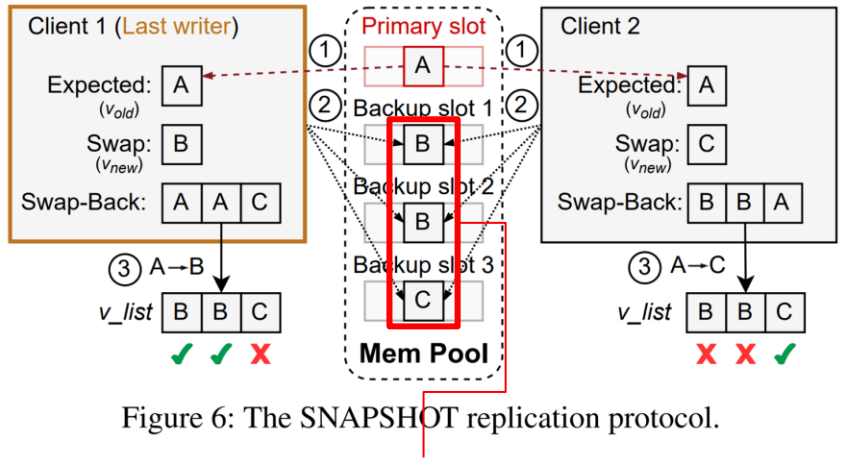


Figure 6: The SNAPSHOT replication protocol.

In Figure 6, Client 1 is the last writer according to Rule 2.

- With v_{list}, SNAPSHOT defines the following three rules to decide 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

Proposed Method

2. Two-level memory management scheme
   - To achieve efficient remote memory management
   - Splits the server-centric memory management process
     - The compute-light tasks: coarse-grained memory blocks are managed by the memory nodes
     - The compute-heavy tasks: fine-grained objects are handled by clients.
3. Embedded operation log scheme
   - To deal with the problem of metadata corruption
   - 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.

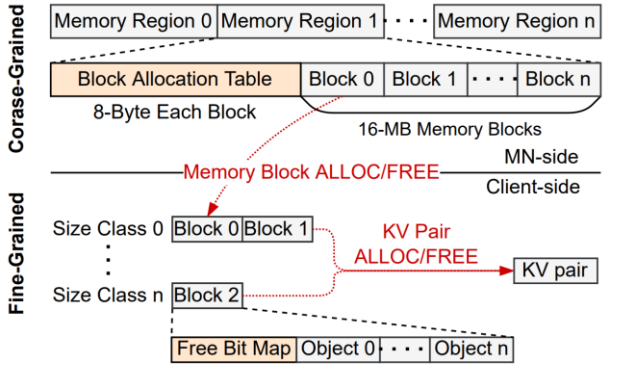
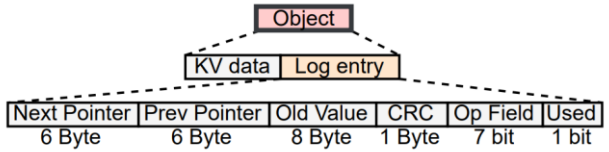
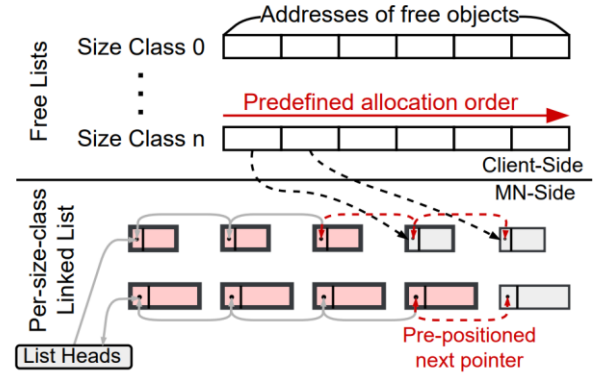


Figure 7: The two-level memory management scheme.



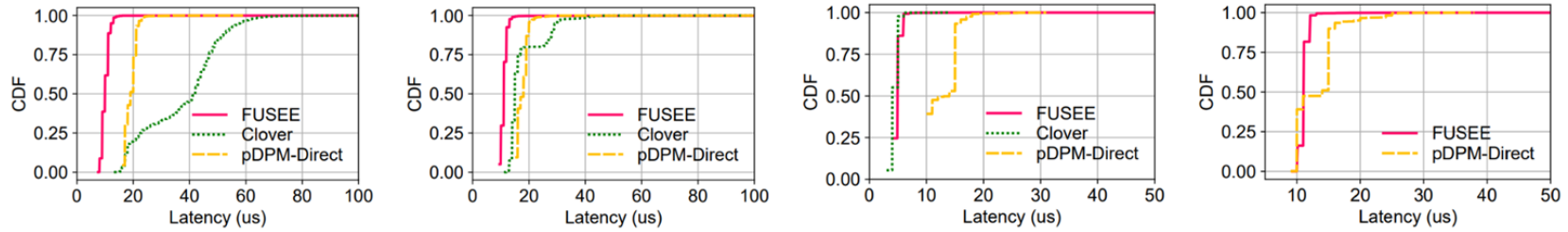
(a) The embedded log entry.



(b) The organization of the embedded operation log.

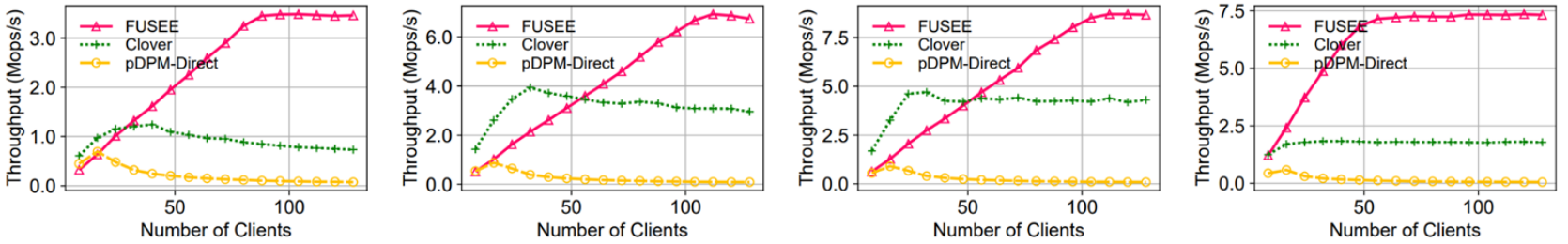
Figure 8: The embedded operation log.

Evaluation and Results



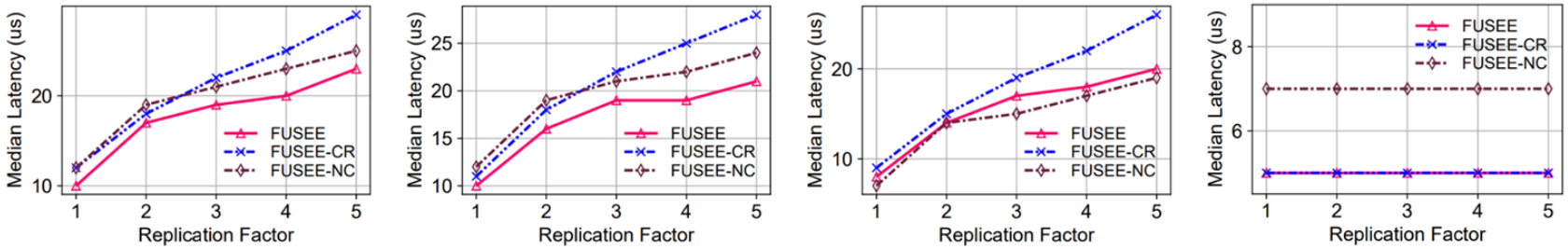
(a) INSERT latency CDF. (b) UPDATE latency CDF. (c) SEARCH latency CDF. (d) DELETE latency CDF.

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



(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) 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.

Evaluation and Results

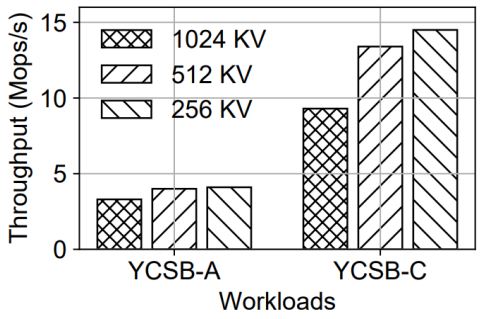


Figure 12: The throughput of FUSEE under different KV sizes.

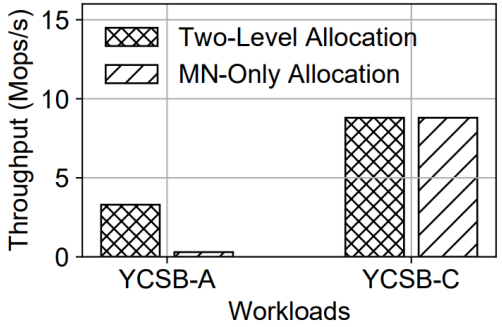
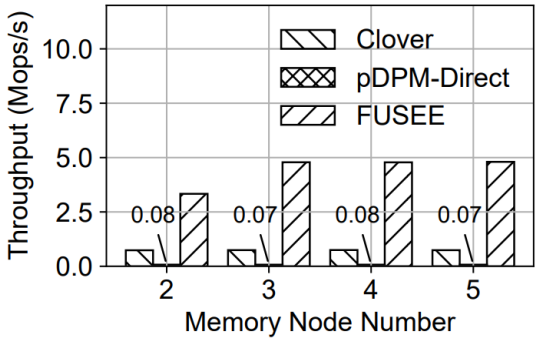
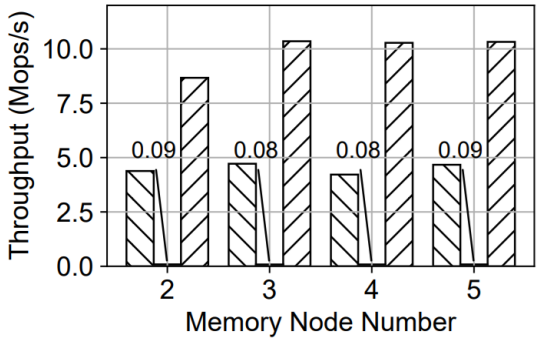


Figure 17: The throughput of different memory allocation methods.



(a) YCSB-A throughput.

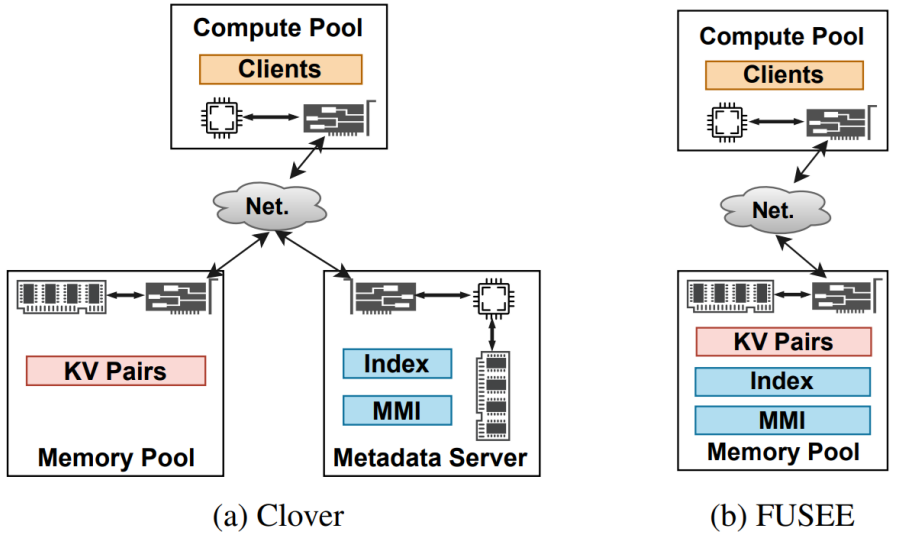


(b) YCSB-C throughput.

Figure 14: The throughput with different numbers of MNs.

Notes

- **The Disaggregated Memory Architecture**: DM **separates CPUs and memory** of monolithic servers into two independent hardware resource pools containing **compute nodes (CNs)** and **memory nodes (MNs)**.
  - CNs have abundant CPU cores and a small amount of memory as local caches.
  - MNs host various memory media, e.g., DRAM and persistent memory, to accommodate different application requirements with weak compute powe
- However, constructing KV stores on DM is challenging because the memory pool generally lacks the compute power to manage data and metadata.
  - Existing work proposes a *semi-disaggregated design* that stores KV pairs in the disaggregated memory pool but retains metadata management on monolithic servers.



(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.