Enabling Efficient Erasure Coding in Disaggregated Memory Systems

Qiliang Li, Liangliang Xu, Yongkun Li, *Member, IEEE*, Min Lyu, *Member, IEEE*, Wei Wang, Pengfei Zuo, and Yinlong Xu

***Abstract*—Disaggregated memory (DM) separates compute and memory resources to build a huge memory pool. Erasure coding (EC) is expected to provide fault tolerance in DM with low memory cost. In DM with EC, objects are first coded in compute servers, then directly written to memory servers via high-speed networks like one-sided RDMA. However, as the one-sided RDMA latency goes down to the microsecond level, coding overhead degrades the performance in DM with EC. To enable efficient EC in DM, we thoroughly analyze the coding stack from the perspective of cache efficiency and RDMA transmission. We develop MicroEC, which optimizes the coding workflow by reusing the auxiliary coding data and coordinates the coding and RDMA transmission with an exponential pipeline, as well as carefully adjusting the coding and transmission threads to minimize the latency. We implement a prototype supporting common basic operations, such as write/read/degraded read/recovery. Experiments show that MicroEC reduces the write latency by up to 44.35% and 42.14% and achieves up to $1.80\times$ and $1.73\times$ write throughput, compared with the state-of-the-art DM systems with EC and 3-way replication for objects not smaller than 1 MB, respectively. For small objects, MicroEC also evidently reduces the variation of latency, e.g., it reduces the P99 latency of writing 1 KB objects by 27.81%.**

***Index Terms*—Disaggregated memory, erasure coding, pipeline, reliability.**

I. Introduction

In recent years, there has been a notable shift towards utilizing in-memory solutions as the primary toolchain for enhancing the performance of data-intensive applications. This trend is evident in various AI application systems [53], [81] and data analysis systems [15], [28], [93]. Thus, there is a growing demand for the development of memory pools to accommodate large memory requirements for data-intensive applications. Due to the development of the high-speed network, such as remote direct memory access (RDMA), both the bandwidth and latency of network transmission become close to memory access. For example, the throughput of 200 Gb/s Mellanox ConnectX-6 IB RNIC [54] (RDMA NIC) is close to that of local memory access of DDR4 DRAM [50]. Thus, it is possible to develop a huge memory pool with microsecond-level access latency in a scale-out manner. Sharing memory not only scales out the memory capacity but also increases the memory utilization and reduces the memory cost [1], [23], [63]. There are two ways to provide a large-scale shared memory, namely swap-based *remote memory* [2], [23], [49], [63] and object-based *disaggregated memory* [35], [67], [75], [95], [98].

With remote memory, a cluster consists of many monolithic servers, where each server is configurated with individual CPUs and memory. All servers contribute their spare memory space which can be shared by other servers [2], [23], [34], [49], [97]. From the perspective of application developers, a program is allowed to flexibly allocate and deallocate memory of varied size [97], transparently using the local RAM and remote RAM. When local memory pressure is high, the system asynchronously swaps cold local memory out to remote memory. Alternatively, disaggregated memory decouples a system into the compute pool and the memory pool, where the compute pool is configurated with powerful CPUs but with limited memory only for cache, and the memory pool provides a large volume of storage space but with limited computing power [35], [67], [75], [79], [95], [98]. Different from remote memory systems, which swap local memory data to remote memory only if local memory is insufficient, the cache in compute servers only serves the data processing, and once the processing is finished, the data will be directly written to memory servers [35], [67], [75], [95], [98]. This brings the benefit that after a client running on a compute server successfully writes an object to memory servers, the object is visible to other clients, regardless of whether the client is failed or not in the future.

A large-scale memory pool also makes the disaggregated memory prone to failures [19], [60]. To enable fault tolerance, the widely used *replication* scheme provides a simple approach [17], [30], but it introduces high memory cost and extra write latency because of writing redundant copies [34].

Manuscript received 17 August 2023; revised 1 November 2023; accepted 9 November 2023. Date of publication 15 November 2023; date of current version 8 December 2023. This work was supported in part by the National Key Research and Development Program of China for Young Scholars under Grant 2021YFB0301400, and in part by the National Nature Science Foundation of China under Grants 61832011 and 62172382. Recommended for acceptance by D. Li. *(Qiliang Li and Liangliang Xu equally contributed to this work.) (Corresponding author: Min Lyu.)*

Qiliang Li and Wei Wang are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China (e-mail: leeql@mail.ustc.edu.cn; wangwww@mail.ustc.edu.cn).

Liangliang Xu is with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China, and also with the Huawei Cloud, Shanghai 201206, China (e-mail: llxu@mail.ustc.edu.cn).

Yongkun Li, Min Lyu, and Yinlong Xu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China, and also with the Anhui Province Key Laboratory of High Performance Computing, Hefei 230026, China (e-mail: ykli@ustc.edu.cn; lvmin05@ustc.edu.cn; ylxu@ustc.edu.cn).

Pengfei Zuo is with the Huawei Cloud, Shenzhen 518100, China (e-mail: pfzuo@hust.edu.cn).

Digital Object Identifier 10.1109/TPDS.2023.3332782

Erasure coding (EC) provides an alternative that offers the same level of fault tolerance with an order of magnitude less extra storage overhead. There are some works deploying EC in remote memory systems [34], [97], but only for the data swapped from local memory to remote memory. In the event of a local host failure, the data stored in local memory becomes lost. Application-level fault tolerance schemes such as checkpointing can be employed to ensure the recoverability of the local data, but they come with the drawbacks of time overhead because of writing checkpoints and interrupting the running application during recovery. These factors ultimately degrade the performance of applications. However, in disaggregated memory systems with EC being enabled, data is encoded within the compute pool and directly written to the memory pool, so that all successfully written data can be recovered online even in the event of failures. This paper focuses on deploying EC within disaggregated memory systems, aiming to ensure robust availability for the data successfully written to the memory pool. This entails skillfully orchestrating the interplay between coding and RDMA transmission to effectively reduce the overall latency associated with data write and degraded read scenarios.

To efficiently deploy EC in disaggregated memory, an intuitive approach is pipelining the coding and RDMA transmission. However, realizing an efficient pipeline in disaggregated memory with EC faces multiple challenges. First, realizing an efficient pipeline needs to split a stripe into multiple substripes. However, coding many small substripes independently increases the total coding latency due to reloading the auxiliary coefficients (see Section III-A). We should design the coding workflow by reusing the auxiliary coding data of the first substripe in cache, e.g., the instruction set and multiplication table, so as to reduce the coding time of subsequent substripes. Second, we should use dedicated cores to execute the coding tasks, which will avoid cache pollution by other threads. Last but not least, the coding speed and RDMA transmission throughput highly depend on the object size. Only when the coding and RDMA transmission is well coordinated, we can efficiently pipeline them and minimize the total latency of write/read in disaggregated memory.

Based on the above insights, we develop a new EC workflow, MicroEC, for disaggregated memory. We design the coding workflow by leveraging the on-chip cache access locality to minimize the coding latency, and also carefully coordinate the coding and RDMA transmission to realize an efficient pipeline. With a careful implementation of these techniques, MicroEC makes the coding and RDMA transmission well-pipelined. Thus, MicroEC achieves a lower write latency than existing EC and replication policies. We point out that due to the inherent latency of the software stack, our pipeline scheme only achieves marginal reduction of write/read latency for small objects (e.g., <64 KB). However, many industrial systems show that small objects only occupy a small portion of the storage space [4], [58], [78], [82], [88]. So we argue that it is reasonable to use replication for fault-tolerance of small objects with limited extra storage overhead. Our contributions are summarized as follows.

- We thoroughly analyze the coding stack from the perspective of cache efficiency and RDMA transmission. We find out the critical performance degradation when deploying EC in disaggregated memory. We provide two key insights for optimizing the coding and RDMA transmission workflow at the system level.
- We design the coding workflow by reusing the auxiliary data to accelerate coding. We also propose efficient data structures to support the design. Besides, we coordinate the coding and RDMA transmission with a non-blocking pipeline, and carefully adjust the coding and transmission threads to minimize the latency.
- We implement a prototype, MicroEC, on top of Apache Crail [73] with various optimizations. Extensive experiments show that compared to the state-of-the-art EC approaches and 3-way replication, MicroEC achieves up to 1.80× and 1.73× write throughput for objects not smaller than 1 MB, respectively. Furthermore, compared to the system without fault tolerance, MicroEC reduces the write throughput only by 19.8% including the parity writes. For small objects, e.g., 1 KB objects, MicroEC reduces the P99 latency by 27.81%. In short, MicroEC enables efficient EC in disaggregated memory with high performance and low memory cost. The source code of MicroEC is available at https://github.com/ADSL-EC/MicroEC.

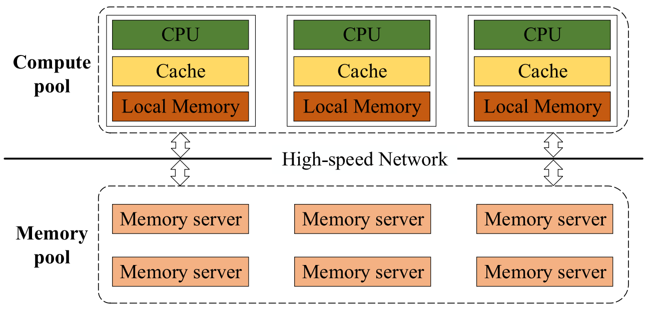


Fig. 1. Architecture of disaggregated memory.

II. Background and Motivation

A. Disaggregated Memory

A disaggregated memory system [35], [67], [75], [79], [95], [98] separates its resources into a *compute pool* and a *memory pool*, as shown in Fig. 1. Each pool is managed and scaled independently as well as failure-isolated. Different from the monolithic servers in traditional data centers and remote memory systems, the compute servers in compute pool host substantial CPU cores to perform applications, but only a small reserved memory as cache. And the memory servers in the memory pool contain many memory blades, such as DRAM DIMMs, to store the application data, and are configured with weak compute units only for memory allocations and network interconnections. To provide low access latency without the involvement of CPU in the memory pool, the compute servers and memory servers are connected by a high-speed network with remote access techniques, such as one-sided RDMA, Omni-path [5], CXL [42] and Gen-Z [11]. This paper focuses on the widely used one-sided RDMA, which provides `WRITE` and `READ` with similar throughput to local memory access [8] and microsecond (μs) level latency.

The capacity of local memory is a key factor of performance for remote memory systems. If the capacity of local memory in remote memory systems is restricted to relatively small like in disaggregated memory systems, its performance may drop dramatically. For example, when the capacity of local memory is only 10% of the working set size, the throughputs of Hydra [34] and Carbink [97] decrease over 50% compared with their peaks in transactional KV-store evaluation [97]. So disaggregated memory is a potential approach to build a huge memory pool with high elasticity.

B. Erasure Coding

A (k, m) systematic code encodes k data blocks $B_1, B_2, \ldots, B_k$ into m parity blocks $P_1, P_2, \ldots, P_m$, forming a stripe. We focus on (k, m) codes with Maximum Distance Separable (MDS) [48], [62] property, that any block can be reconstructed from surviving blocks in the same stripe in case of m blocks being failed. So a (k, m) MDS code tolerates up to m failures with m/k times extra storage overhead, while $(m + 1)$-way replication needs m times. So EC can provide the same level of fault tolerance with less extra storage overhead. For example, (10, 2) code needs 20% extra storage, while 3-way replication needs 2×. So EC is widely used in storage systems to provide fault tolerance.

Limited by the computational power in memory servers, when writing an object in disaggregated memory with EC, we should encode it by a compute server and then write it into memory servers via RDMA `WRITE`. For degraded read in case of failures, we should read surviving blocks of the same stripe from memory servers to a compute server via RDMA `READ` and decode the failed blocks. To efficiently provide fault tolerance with EC in disaggregated memory, we should overlap the latency of coding and RDMA `WRITE/READ` by leveraging an efficient pipeline. Otherwise, the end-to-end write/read latency will be largely increased. Though there are some works on EC to provide in-memory fault tolerance recently [34], [74], [78], [86], [94], [97], they mainly focus on metadata or memory management of EC.

C. Coding Latency Vs. RDMA Latency

We conduct experiments to evaluate the coding latency with RDMA `WRITE` latency on a cluster of 17 nodes. Each node is configured with two Xeon(R) E5-2650 CPUs and 64 GB DRAM and connected with other nodes via 56 Gbps RDMA network. We take the widely used (4,2) Cauchy Reed-Solomon (CRS) code [19], [77], [78] as an example, and use the Intel ISA-L library [14], the most popular library with highly optimized instructions. The latency of RDMA `WRITE` is measured with `ib_write_lat`. We present the average coding latency of successively encoding 10 K objects with the same thread. Note that the objects to be encoded already reside in the memory of a compute server.

From Fig. 2, we find that the average coding latency is comparable with the latency of RDMA `WRITE`. For instance, for an object of 1 MB, the latency of RDMA `WRITE` is 1.26× the average coding latency. It provides us an opportunity of designing a pipeline to overlap the latency of coding and RDMA transmission.

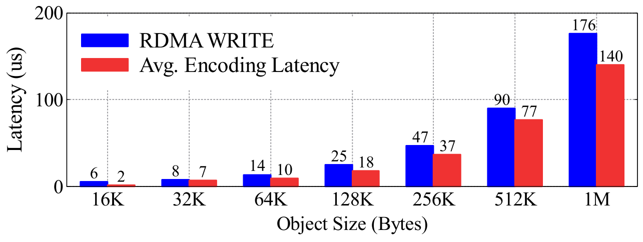


Fig. 2. Latencies of RDMA `WRITE` and encoding.

D. Issues of Realizing Pipeline

We face the following key issues to efficiently pipeline the coding and RDMA transmission.

No pipeline support: In the existing implementation of in-memory systems with erasure coding [10], [34], [37], [58], [91], [94], [97], a stripe is encoded and transmitted entirely without further splitting, which implies that the coding and network transmission are executed in synchronization. To overlap the latency of coding and network transmission, the stripe should be split into substripes, and the transmission of a substripe should be fully asynchronous with the coding of the following substripes. As a result, the progress of coding and RDMA transmission should be decoupled with asynchronous threads. MicroEC targets the stripe formed by encoding a single object, which will be further split into subobjects, and then coded into substripes.

Tradeoff of subobject size: Note that RDMA `WRITE` of the first subobject must wait for the completion of its coding, so its coding latency can not be overlapped. Besides, the last subobject is written to the memory pool by RDMA `WRITE` only after its coding is completed, resulting in the inability to overlap its network latency as well. So too large subobjects will induce long first-round and last-round latency in the pipelined workflow. However, small subobjects bring more RDMA round trips, and also incur extra cost degrading the performance [29], [31], [51], [76], such as creating work queue elements in the client host and receiving completion entries from memory servers. Therefore, we should carefully adjust the subobject size for the coding and RDMA transmission.

III. Key Insights and Design Opportunities

We investigate the coding stack of EC workflow to support the pipeline and present key insights via experiments. To write an object with a (k, m) code, the workflow consists of three steps as shown in Fig. 3: *split*, *encode* and RDMA `WRITE`. First, in the *split* step, the object is logically split into k data blocks with very low and even negligible overhead. Second, in the *encode* step, a compute server executes the encoding computation to generate m parity blocks with the k data blocks. Finally, in the RDMA `WRITE` step, the compute server writes the $k + m$ data and parity blocks to the memory pool. To enable encoding, the compute

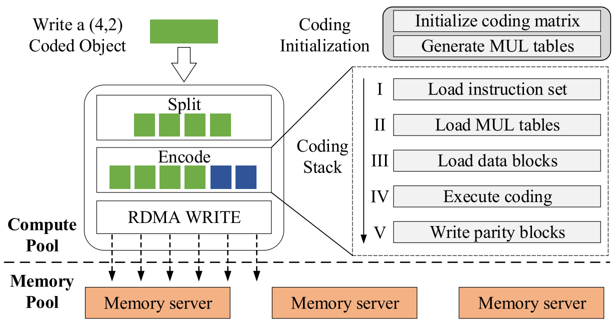


Fig. 3. Write workflow in EC-enabled DM systems.

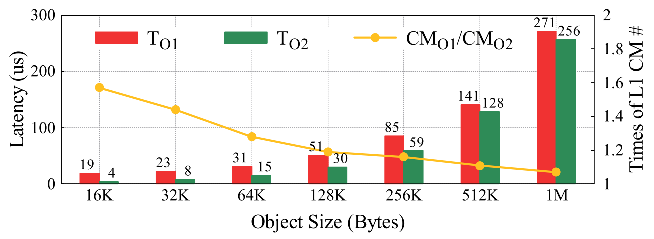


Fig. 4. Latency and L1 cache miss of coding two successive objects. T_{O1} and T_{O2} denote the latencies of coding O1 and O2, respectively. CM$_{O1}$/CM$_{O2}$ denotes the times of L1 cache miss number in coding O1 and O2.

server needs to initialize the coding matrix and generate the MUL table for multiplication (some coding algorithms use bit matrices [6], [57], [96] instead of MUL tables) at system startup and keeps the MUL table in memory to avoid re-initialization if the parameters (k, m) are not changed. We point out that there is a long coding stack, as shown in Fig. 3. It consists of five stages: (I) *loading the instruction set* designed for efficient coding, (II) *loading MUL tables* from memory, (III) *loading data blocks*, (IV) *executing coding* to generate parity blocks, and (V) *writing parity blocks* to memory. In the following, we present our key insights on accelerating the coding and RDMA `WRITE` and also analyze the challenges to leverage these optimization opportunities.

A. The Effect of Coding Locality

The coding latency in Fig. 2 is the average of successively encoding 10 K objects with the same thread, which dilutes the coding's cold start time 10 K times. To study the coding latency in practical implementation, we encode only two successive objects, denoted as "O1" and "O2", which have the same size ranging from 16 KB to 1 MB, with the same EC code and coding matrix. We encode two objects at the same core to study coding stack efficiency. To avoid the influence of outliers, we repeat the test 1 K times. The average latencies of the coding stack in Fig. 3 are shown in Fig. 4.

We find that the latencies for coding the second object are significantly reduced, especially for objects of small size like 16 KB. As a result, the total latency of coding the first 16 KB object is 19 μs, which is $4.75\times$ that of coding the second one. The reason is that when the thread encodes the first object, it needs to load the instruction set and MUL table to the L1 cache. After coding the first object, the instruction set and MUL table are still in the cache and can be reused for coding the next one, so the latency of coding the second object is greatly reduced. The differences in coding O1 and O2 on the number of L1 cache misses, including dcache and icache measured via `perf` [43], are shown in Fig. 4, which further validate our analysis. Note that as the object size increases, the cache misses induced by loading data blocks dominate the total number of cache misses, which makes the difference in coding O1 and O2 on the number of L1 cache misses gradually decreasing.

Implication 1: The results imply that we can split an object into multiple subobjects, and keep the auxiliary data for encoding the first subobject in cache, then reuse the auxiliary data to encode subsequent subobjects. Reusing the auxiliary data in cache will greatly reduce the coding latency of subsequent subobjects, and avoid increasing the coding latency due to splitting objects. Only by this, we can efficiently realize the pipeline of the coding and RDMA write. *The key challenge* is that it necessitates carefully designing the coding workflow so as to guarantee that the auxiliary data for coding in the L1 cache can be reused.

B. The Effect of Splitting for Pipeline

The above analysis implies that large objects can be split into small ones for pipelining. As the latency of coding with optimizations already becomes comparable with RDMA `WRITE` latency, the end-to-end write latency can be greatly reduced with carefully coordination. However, there are two factors to be considered in utilizing coding cache locality. On the one hand, the coupling of coding and data transmission, such as using a single thread in traditional pipeline implementation like HDFS recovery pipeline [25], will reset the related buffers after coding each subobject for easy memory management, and thus make it impossible to reuse the auxiliary data. On the other hand, the coding operation may be interfered by other applications or OS tasks running on the same CPU core. To overcome this difficulty, we leverage two asynchronous threads for the coding and RDMA transmission separately (details in Section IV-C), and also isolate dedicated CPU cores for the coding (details in Section IV-E).

Furthermore, splitting objects induces many small subobjects, which inevitably increase RDMA transmission rounds. Thus, there is a conflict between the optimal object sizes for maximizing the coding speed and minimizing the RDMA `WRITE` latency. The simple approach of packing as many small subobjects as possible for RDMA `WRITE` is inefficient. The rationale is that as the object size increases, the marginal latency reduction of RDMA `WRITE` decreases. For example, as shown in Fig. 2, when we double the object size from 16 KB to 32 KB, the latency of RDMA `WRITE` increases from 6 μs to 8 μs, only $1.33\times$. From 128 KB to 1 MB, the latency increases near $2\times$ when the object size is doubled. In addition, packing a too-large packet for RDMA `WRITE` hinders the pipeline efficiency, as it induces a long time to wait for the coded data before starting RDMA `WRITE`.

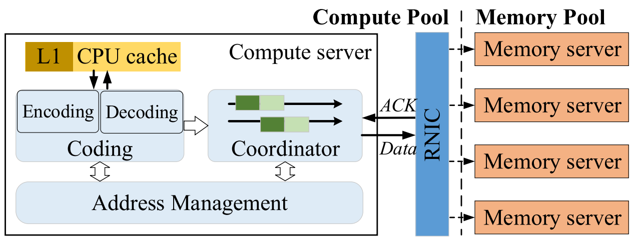


Fig. 5. Overview of MicroEC design.

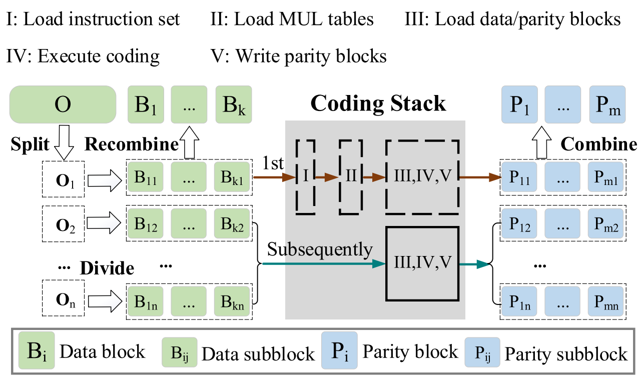


Fig. 6. Coding procedure of MicroEC.

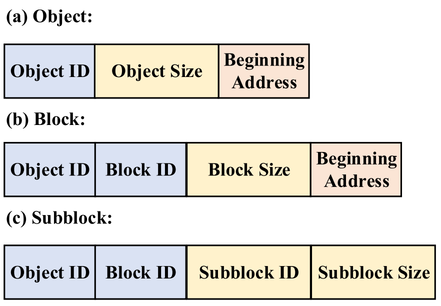


Fig. 7. MicroEC data structures.

Implication 2: The results imply that with an optimized coding workflow, there is an opportunity to pipeline the coding and RDMA transmission. This motivates us to design an efficient pipeline to minimize the total delay of EC writes. *The key challenge* is that it requires careful coordination of the coding and RDMA transmission, as well as an efficient way to adjust the mismatched object sizes between the two operations, so as to realize the maximum parallelism.

IV. Design

We develop a prototype, MicroEC, to enable efficient EC in disaggregated memory. As illustrated in Fig. 5, MicroEC consists of three modules, *coding*, *address management* and *coordinator*. The *coding* module implements the coding operation with optimized coding workflow. The *address management* module is responsible for efficient access to the objects for coding and RDMA transmission, and supports object splitting, subobjects accessing and combining. The *coordinator* module coordinates coding and RDMA transmission to realize an efficient pipeline.

A. Coding

Coding Procedure: We first present the coding procedure from the view of L1 cache. Suppose that we encode an object using a (k, m) code with coding matrix $M = (m_{ij})_{k\times m}$. The object is first divided into k data blocks, say $B_1, B_2, \ldots, B_k$, and then encoded into m parity blocks $P_1, P_2, \ldots, P_m$. We take the Intel ISA-L [14] library, which is the start-of-the-art coding library, as an example to illustrate the detailed coding process. To encode, the *auxiliary data*, including the multiplication tables and the AVX vectorization instruction sets, are first loaded into L1 cache, and then the data blocks, $B_1, B_2, \ldots, B_k$ are loaded into L1 cache one by one for encoding as L1 cache has limited space. Once B_i $(1 \leq i \leq k)$ is loaded into L1 cache, the multiplication of $m_{ij} \times B_i$ is executed, and all parity blocks P_j's $(1 \leq j \leq m)$ are updated as $P_j \oplus (m_{ij} \times B_i)$. When all data blocks of the same object are processed, the coding process terminates and all parity blocks are evicted from L1 cache to memory.

Pipeline Oriented Coding (`POC` *):* Suppose that we are to write an object O with a (k, m) code in disaggregated memory. To realize the pipeline of coding and RDMA transmission, we first split O into n subobjects O_1, O_2,..., O_n, and then encode and RDMA `WRITE` them one by one in pipeline. Fig. 6 illustrates the procedure of `POC`. To encode O_j, it is first divided into k subblocks $B_{1j}, B_{2j}, \ldots, B_{kj}$, and then they are encoded into m subblocks of parities $P_{1j}, P_{2j}, \ldots, P_{mj}$ to form a substripe S_j. Once all subobjects $O_1, O_2, \ldots, O_n$ are encoded, we recombine subblocks $B_{i1}, B_{i2}, \ldots, B_{in}$ into B_i for $1 \leq i \leq k$ and combine $P_{i1}, P_{i2}, \ldots, P_{in}$ into P_i for $1 \leq i \leq m$. So we divide O into k data blocks $B_1, B_2, \ldots, B_k$, and encode them into m parity blocks $P_1, P_2, \ldots, P_m$.

To increase the coding throughput, we create a thread that codes an object on an isolated core until the coding process is completed. Unlike the traditional coding stack, we do not evict the auxiliary data for encoding the first subobject and still keep them in cache even if the coding process of the first subobject is completed, and reuse the auxiliary data to code the subsequent subobjects until the whole object is coded. So the coding latencies of subsequent subobjects and the object are greatly reduced. We set the default subblock size as 4 KB for memory page alignment and RDMA packet coordination.

B. Address Management for `POC`

We design lightweight data structures to avoid overhead caused by splitting objects. As the objects already reside in memory, if we record the in-memory beginning address of an object O, denoted by $\mathcal{A}_O$, we can calculate the beginning address of each block and each subblock, and thus realize logical split. As shown in Fig. 7, the data structure of object O records an object ID, object size s and its beginning address $\mathcal{A}_O$. For the split block B_i, its data structure consists of an object ID, a block ID, block size and its beginning address. The data structure of

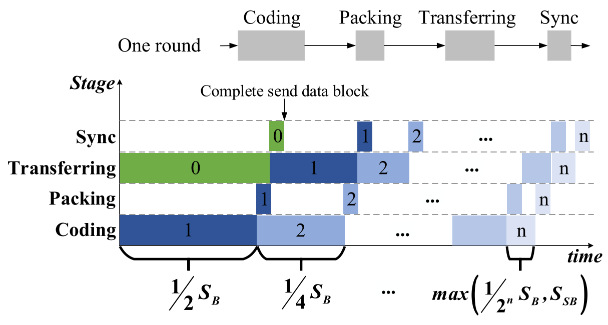


Fig. 8. Illustration of the non-blocking pipeline, where S_B is the size of a data/parity block and S_{SB} is the size of a data/parity subblock.

a subblock consists of object ID, block ID, subblock ID and subblock size.

When we encode a subobject O_j for $1 \leq j \leq n$, we can read its k subblocks $B_{1j}, B_{2j}, \ldots, B_{kj}$ according to their beginning addresses and the subblock size, and then encode them into m subblocks of parities, $P_{1j}, P_{2j}, \ldots, P_{mj}$. The process of writing parity blocks into memory is similar. Note that the extra costs for supporting the pipeline oriented coding are only the above introduced data structures. Besides, the data structures of subblocks are freed after completing the write of each object. So the overhead is very low.

C. Asynchronous Coding and RDMA Write

Asynchronous Threads: We leverage two asynchronous threads, one thread (*the coding thread*) is for coding, and the other thread (*the packing thread*) is for packing the coded subblocks for RDMA transmission. To coordinate the two threads, we define a semaphore, `Counter`, which is shared by the two threads. Once a subobject is coded, the coding thread increases `Counter` by one. When sufficient subobjects are coded, the packing thread reads data from each data/parity block, which belongs to the just coded subobjects, and sends it to the network buffer for RDMA transmission. Meanwhile, it resets `Counter` to zero. We point out that packing subblocks into a packet is efficiently supported by our designed data structures (see Section IV-B). It is similar to reading subblocks for coding subobjects.

Non-Blocking Pipeline: We design a non-blocking pipeline protocol to minimize the total latency of the write procedure, which consists of four stages, *coding*, *packing*, *transferring* and *sync*, as shown in Fig. 8. Specifically, *coding* means asynchronously encoding multiple subobjects and updating the corresponding data structures in real-time, *packing* means preparing the data for RDMA `WRITE` by logically combining small subblocks into large packets, *transferring* means executing data writing to memory servers via asynchronous RDMA `WRITE`, and finally *sync* means acknowledging the success of the RDMA `WRITE`. Once a compute server receives the ACKs from all memory servers, one round of transferring and data writing finishes successfully. Finally, we carefully coordinate the four stages to make them work perfectly with *exponential pipeline* (see Section IV-D).

D. Exponential Pipeline

To write an object with an erasure code in disaggregated memory, we should write all data subblocks and all parity subblocks into memory pool by RDMA `WRITE`. From Fig. 2, we know that RDMA `WRITE` with larger size of packets will achieve higher throughput. So from the view of RDMA transmission throughput, we should wait for more parity subblocks to be generated and pack them into a large packet to achieve high transmission throughput. However, waiting for more parity subblocks will lead to an extended wait time for transmission in the first round, as well as an elongated transmission latency during the last round, because the RDMA `WRITE` of the packets of parity blocks in the first round must wait for the completion of its coding and the packets of parity blocks in the last round is written to the memory pool only after its coding is completed. So we should carefully design the packet size selection of RDMA `WRITE` for an efficient pipeline.

Coordination of the Exponential Pipeline: Note that at the beginning of coding, we can immediately pack the data blocks and write them into the memory pool. So there is no wait time for data transmission in the first round. In the following, we only discuss the RDMA `WRITE` of parity blocks for an efficient pipeline. We propose an *Exponential Rule* for the selection of packet size at each round of RDMA `WRITE` as follows.

- At the first round, we pack parity subblocks of size $\alpha^{-1} \times S_B$ into a packet for RDMA `WRITE`, where S_B is the size of a data/parity block. For example, $\alpha = 2$ in our testbed.
- If all parity blocks are submitted to RDMA `WRITE`, then all data/parity blocks have been written to the memory pool. The total write process of the current object is finished. Otherwise, start the next round of RDMA `WRITE`.
- At each subsequent round, we reduce the packet size by α^{-1} times until it equals to the size of a parity subblock.

Example: Take writing a 1 MB object with (4,2) code as an example, where the data/parity block size S_B is 256 KB. To enhance clarity, we build a correspondence between the outlined steps and the phrases in Fig. 8. In the first round, the first $1/2 \times S_B = 128$ KB of each data block is encoded following the coding procedure in Section IV-A (see the dark blue box marked with `1` in the coding stage), and asynchronously four 256 KB data blocks are sent to the memory pool at the same time (see the green box marked with `0` in the transferring stage), therefore hiding the first-round coding latency in the transmission of data blocks. In the second round, the next $1/2^2 \times S_B = 64$ KB of each data block is encoded (see the blue box marked with `2` in the coding stage), and in the meantime, two 128 KB parity packets encoded in the first round are packed (see the dark blue box marked with `1` in the packing stage) and transferred (see the dark blue box marked with `1` in the transferring stage). The following rounds go on in the same way, except the size of the transferred parity packets is reduced by $1/2$ until it equals the size of a parity subblock 4 KB. The way of exponentially decreasing the parity packet size guarantees fewer rounds for transmission and makes the last-round latency short enough. The breakdown analysis in Section VI-C validates the coding and network transmission are well pipelined.

Analysis: The configuration of the parameter α is primarily determined by the computational performance of CPU processors and the network speed. If CPU processors are able to support high-performance coding instruction sets, such as AVX512, we should select a small α to increase the first-round packet size for reducing the network transmission rounds. In the case of a higher network transmission speed, such as 200 Gbps RNIC or CXL, we should select a large α to write the coded subobjects into the memory pool timely. Thus, the relation between α and the speeds of coding and network transmission satisfies $\alpha \propto \frac{T_n}{T_c}$, where T_n is the speed of network transmission and T_c is the coding throughput. In our hardware configurations, we choose $\alpha = 2$ by default, as shown in Fig. 8.

Support for Degraded Read and Recovery: Degraded read and recovery need to read k source blocks from the memory pool by RDMA `READ`, and execute decoding for the unavailable/failed block(s). Unlike the fixed encoding matrix of (k, m) code, the decoding matrix changes based on both the unavailable/failed block(s) and the k source blocks participating in decoding. This consequently introduces the overhead of initializing the decoding matrix and correlated MUL table in the decoding workflow for each degraded read and recovery request. To hide such overhead, we pre-activate the coding thread to initialize the auxiliary data for decoding during the data packet transmission in the first round. And for the successive rounds, the reading of k source blocks is in parallel with the decoding of the packets received from the preceding round. In addition, since the recovery requests need to write the recovered block(s) back to the memory pool, we write the recovered packet(s) to the memory pool in parallel with the next-round reading and decoding process.

E. Optimizations

Dispatch Coding Tasks for Multi-Client: In our design, each coding thread prefers to monopolize a CPU core, so as to avoid the auxiliary coding data being evicted from L1 cache due to running other threads on the same core. To achieve this, we isolate a subset of CPU cores using Linux `cgroup`, specifically the `isolcpus` option, and employ shared memory to maintain a `flag` for each isolated core for recording the usage status, where `0` indicates idle state and `1` indicates busy state. The coding threads are dispatched into these isolated CPU cores, which first poll the `flags` to get an idle isolated core, then set the corresponding `flag` to `1` and assign the coding thread to the core, while the application clients run on the non-isolated cores based on the default process scheduler provided by operating systems. When all `flags` are set to `1`, which means all isolated cores are actively involved in coding, and a new coding thread attempts to execute, the new thread will be blocked and will continue polling the `flags` until an isolated core becomes available.

The principle of configuring the number n of isolated CPU cores is to make the isolated ones not be underused or overloaded. The most crucial factor that determines n is the number of concurrent coding threads. In case of more than n concurrent coding threads, it may degrade the coding performance due to the contention and blocking between multiple coding threads. However, the number of concurrent coding threads is determined by the number of concurrent clients and the ratio of requests with erasure coding. In common, the write/degraded read/recovery requests are involved in encoding/decoding stages, while normal read requests just fetch the corresponding data blocks without coding. The ratio of those requests involved in coding together with the number of clients determine the number of concurrent coding threads. In addition, the size of requests involving coding also reflects the time of a coding thread occupying the isolated CPU core. In summary, it is hard to precisely determine n, and we conduct experiments to further study the configuration of n in Section VI-D.

Optimization on Small Objects: Small objects commonly exist in practical systems, such as key-value systems [18], [64] and in-memory cache systems [16], [46], [89]. Too small objects only have one substripe, so *exponential pipeline* can only benefit from the overlapping on sending k data blocks and coding within one substripe. Meanwhile, as a lot of requests with small sizes usually come successively [88], [90], it gives us a chance to leverage the coding locality with contiguous requests. We dispatch the coding tasks of consecutive requests into the same cores to further share the auxiliary coefficients in L1 cache. For example, when a client consecutively writes multiple small objects, we can reuse the coding matrix and MUL tables between objects. Performance evaluation in Section VI-C shows that the performance of writing small objects becomes more stable with this optimization.

We also evaluated the latency of writing small objects (detailed in Section VI-B1), even though MicroEC still incurs higher latency (24 μs) than REP3 (15 μs) for writing small objects. The data path of MicroEC includes submitting the asynchronous network transmission of data/parity blocks (16.5 μs), which dominates the latency and can not be optimized anymore. Besides, we point out that many object stores have megabyte-level objects, e.g., most files are larger than 20 MB in the Yahoo! trace [92]. The docker image registry service uses an object store to store large-size container images [4], [44], e.g., more than 20% objects are larger than 10 MB in an IBM data center. Besides, there are also many studies focusing on large objects ($>$1 MB) [58], [78]. Although in some workloads, the requests with small sizes may occupy the majority of all requests, their total size in terms of storage is not large. Take the widely used IBM Docker registry production workload [4], [78] as an example, the requests with smaller than 64 KB account for over 70%, but their total size of storage footprint accounts for less than 0.1%. Thus, we suggest a hybrid way of replicating the objects smaller than 64 KB and storing objects no smaller than 64 KB with EC, as in recent work [88].

Despite the inability to further reduce the latency of writing small objects due to the software stack overhead, it is possible to enhance the throughput of small objects with EC through various engineering techniques. One widely adopted approach is interleaved request processing [95], [99], which employs multiple coroutines [30], [84], [95], [99] within a single thread to process different requests in a pipelined fashion. Each coroutine acts as a coordinator to execute requests. After issuing the submission of one request, a coroutine yields its CPU core to another coroutine

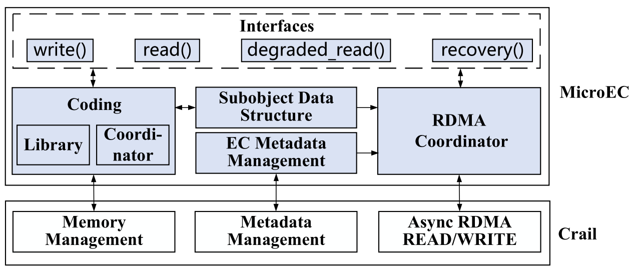


Fig. 9. MicroEC prototype on top of Crail. The shadow boxes are the new components introduced by MicroEC.

to process the next request. This technique increases the throughput by overlapping the software stack overhead among multiple requests, and it is orthogonal to the optimizations in MicroEC. Note that our current prototype of MicroEC primarily focuses on the pipeline within a single request without integrating the interleaved request processing technique.

V. Implementation

MicroEC Coding Library: We implement the coding procedure from scratch with around 1.2 K lines of C code and also leverage Intel ISA-L [14]. Based on the data structures proposed in Section IV-B, we can read a segment from a specific memory region and realize the coding. MicroEC provides two simple coding APIs, `encode()` and `decode()`. `encode()` takes data blocks and coding parameters (e.g., k, m and object size) as its input, and generates parity blocks as its output. As for `decode()`, it needs extra `indices` of surviving blocks for decoding the lost blocks. MicroEC provides its coding functions as a shared library, which can be dynamically linked into the address space of client processes and deployed in disaggregated memory systems.

MicroEC Prototype: We implement MicroEC on top of Crail [73] by modifying/adding around 6 K lines of JAVA code, as shown in Fig. 9. Crail is an in-memory store, providing a unified memory space, and supporting one-sided RDMA and concurrent access to the memory pool. Based on the *metadata management* in Crail, MicroEC realizes the *EC metadata management* to record coding information, e.g., data/parity block ID and coding parameters, etc. Based on the asynchronous RDMA `READ/WRITE` in Crail, combined with the data structures and metadata management, we realize the coordinator in MicroEC. To write an object with the format of `CrailBuffer`, a client first calls `CrailStore.create` to create object metadata in the namespace and get a Crail `file`. Then, it calls the coding APIs with input `CrailBuffer` and executes the coding procedure of MicroEC. We leverage `slice()` to read subblocks from `CrailBuffer`, and put them into the stream of `file`, which writes the subblocks to the memory pool via asynchronous RDMA `WRITE` and returns `future` as the acknowledgment of finishing RDMA `WRITE`.

Note that as for data durability, disaggregated memory systems employ various methods including logging writes [83], persistent memory [75], [95], and battery-backup systems [17], while still maintaining efficient performance. These approaches are orthogonal to MicroEC and can be used in conjunction with it.

MicroEC Interfaces: MicroEC supports four general interfaces, `write`, `read`, `degraded read` and `recovery`, and they can be used by client programs in the following way. `CrailStore.create` creates a file with a coding policy, and `file.getMircoECIOStream` creates a tunnel to send/receive packets of an encoded file ready for RDMA `READ/WRITE`.

```
file=CrailStore.create(filename,policy);
s=file.getMircoECIOStream(filesize);
s.write/read/degraded_read/recovery(buf);
```

Optimizations: After one round of coding, the packing thread will prepare $k+m$ subblocks in the local memory buffer and wait for writing to the memory pool. A simple solution may execute multiple times of `memcpy()` to pack the data from the original data buffer for RDMA transmission. To eliminate `memcpy()` overhead, we leverage `ByteBuffer slice()` to get the data in a shared data buffer and logically pack data for RDMA transmission. MicroEC achieves object buffer sharing without incurring additional memory copies through the inclusion of individual indices and offsets. Note that all data processing and computations in the critical path take place within the user space, ensuring that MicroEC does not necessitate memory copying between the user space and kernel space.

VI. Performance Evaluation

A. Setup

Testbed: We run all experiments on a cluster of 17 nodes. Each node is equipped with two Intel(R) Xeon E5-2650 V4 CPUs, 64 GB DDR4-2400 memory, and 56 Gbps Mellanox ConnectX-3 IB RNIC, which is connected to a 56 Gbps Mellanox InfiniBand switch. Each node runs CentOS 7.8.2003 based on Linux kernel 3.10.0. We assign 4 nodes as compute servers and the remaining 12 nodes as memory servers to create the memory pool, whose total memory capacity is 768 GB. We use 1 node as a namenode, which maintains the hierarchical storage namespace and records file metadata. CPUs of the memory servers are only used to register memory to RNICs in the initialization, and they are not involved in coding. The data transmission between the compute pool and the memory pool leverages one-sided RDMA.

Workloads: We consider both microbenchmark and application benchmarks. For microbenchmark, we adopt the Crail iobench [72], [73], and use it to evaluate the performance of individual operations, e.g., write, read, degraded read, and recovery. For application benchmarks, we consider both a production workload from IBM Docker registry [4], [78] and YCSB [12] with the Zipfian distribution ($\theta = 0.99$).

Baselines: We integrate three state-of-the-art fault tolerance policies into Crail, and they all use one-sided RDMA for fair comparison. (1) **Replication (REP).** It is the simplest approach used in various storage systems [16], [55], and we implement primary-backup replication [7]. The widely accepted industry standard is 3-way replication, and we denote it as REP3. To study how much performance loss is introduced by fault tolerance,

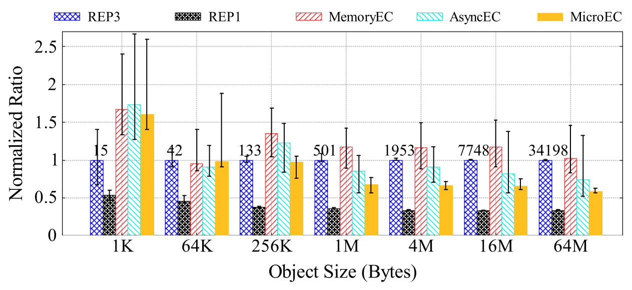


Fig. 10. Write latency (normalized over REP3).

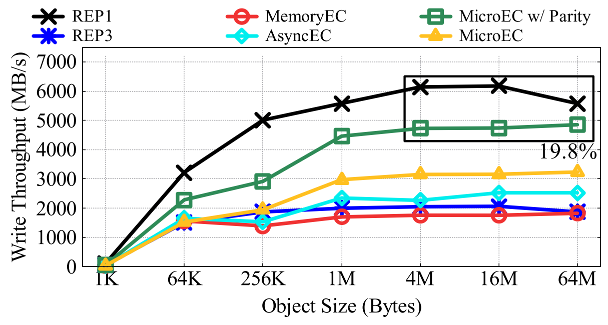


Fig. 11. Write throughput.

we also take into consideration the case of no redundancy (i.e., REP1). (2) **MemoryEC.** It is the most widely used EC scheme in distributed in-memory systems [58], [65], [74], [94], e.g., EC-Cache [58], Cocytus [94] and Carbink [97]. It calls the ISA-L library for encoding and then transmits the coded blocks to memory servers without pipelining. (3) **AsyncEC.** It is the stripe write design used in Hydra [34]. It starts writing data blocks and asynchronously executing encoding, then transmits the parity blocks to memory servers after the coding is finished.

Default Parameters: Note that we use (4,2) code by default, as it is widely used in many systems [19], [77], [78], and we also study different codes in Section VI-D. By default, the coding block size S_b of MicroEC is 4 KB. All schemes use an equal number of CPU cores for fair comparison. For MicroEC, we isolate 3 physical CPU cores from all cores for coding and study the impact of the number of isolated CPU cores in Section VI-D. MemoryEC and AsyncEC bind each coding thread to a randomly selected set of CPU cores to avoid interruptions.

B. Microbenchmarks

We now evaluate the performance of each individual operation, including write, read, degraded read, and recovery. We test 1 K times for each operation and present the average. For setting the object size, as illustrated in Section IV-E, many object stores have megabyte-level objects. So we conduct experiments by varying the object size from 1 KB to 64 MB, and the default object size is 1 MB.

1) Write Performance. Comparison With MemoryEC: As shown in Figs. 10 and 11, we have the following conclusions. First, MicroEC outperforms MemoryEC, e.g., the latency is reduced by up to 44.35% and the throughput increases by up to 79.71%. Second, MicroEC benefits more for large objects, and the improvement becomes significant as long as the object size is no smaller than 256 KB. The reason is that when the object is too small, the pipeline benefit is limited. Furthermore, Fig. 11 shows that the throughput of MicroEC becomes stable at around 3200 MB/s when the object size is larger than 16 MB, and it is around 1.75× that of MemoryEC.

Comparison With AsyncEC: The write latency of AsyncEC can be modeled as $\max\{T_{data}, T_{encode} + T_{parity}\}$, where T_{data} is the latency of writing data blocks, and $T_{encode} + T_{parity}$ is the total latency of encoding and writing parity blocks. Due to the overlap between writing data blocks and encoding, AsyncEC outperforms MemoryEC, reducing the latency by up to 30.45%. However, coding and writing parity can not be overlapped. With the exponential pipeline, compared with AsyncEC, MicroEC further overlaps the coding and writing parity, and thus it reduces the write latency by up to 26.69%.

Comparison With REP3: Compared with REP3, MicroEC with (4,2) code provides the same reliability with only half of the memory cost, while it achieves higher performance for objects larger than 256 KB, e.g., it reduces the write latency by 42.14% and increases the write throughput by 72.83% for objects of 64 MB. REP3 has lower throughput because of more network bandwidth consumption for writing extra copies.

For small objects like 1 KB, MicroEC incurs higher latency (24 μs) than REP3 (15 μs). By breaking down the write workflow of MicroEC with 1 KB request, we find that although the coding latency (3 μs) matches the RDMA latency, the extra latency incurred on the data path, e.g., submitting the network requests of transferring data/parity blocks (16.5 μs), dominates the overall latency. Thus, deploying EC for objects of small size (e.g., <64 KB) inevitably increases the end-to-end latency, and we suggest replicating small objects as they often occupy only a small portion of the storage space [4], [78].

Comparison With REP1: REP1 can be regarded as the performance upper bound as it writes only one copy. MicroEC increases the write latency as it needs extra coding computation and parity writes. However, we emphasize that MicroEC is already sufficiently optimized, and this can be demonstrated by the almost fully utilized RDMA bandwidth. Specifically, MicroEC achieves 3234 MB/s throughput of writing client data, so the total RDMA transmission throughput, including both data and parity blocks, is $3234 \times 1.5 = 4851$ MB/s (denoted by "MicroEC w/ Parity" in Figs. 11). The total throughput reaches around 73% of the bandwidth limit of 56 Gbps RNIC, and is only 19.8% lower than that of REP1.

2) Read and Recovery Performance: Normal/Degraded Reads: Fig. 12 shows the average latency of normal and degraded reads. Due to the same implementation of normal/degraded read for MemoryEC and AsyncEC, they have the same performance and we show them in the same bar. Limited by space, we omit the throughput results as they show similar conclusions. As normal read does not involve coding, EC policies perform almost the same. Compared with REP3, MicroEC splits large objects into small blocks, which brings benefits with asynchronous RDMA, but incurs overhead for each RDMA `READ` due to the smaller packet size. Putting the two factors together, MicroEC

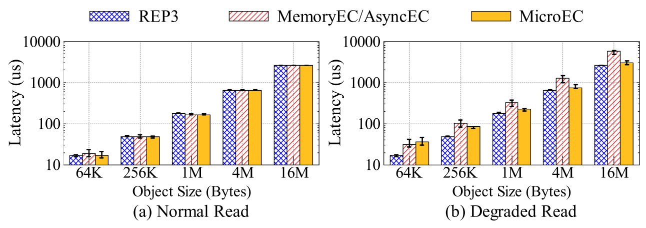


Fig. 12. Normal read and degraded read latency.

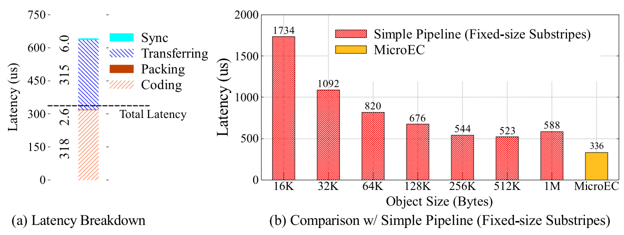


Fig. 14. Performance breakdown for writing a 1 MB object.

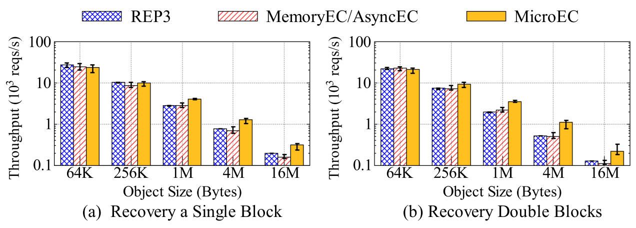


Fig. 13. Recovery throughput.

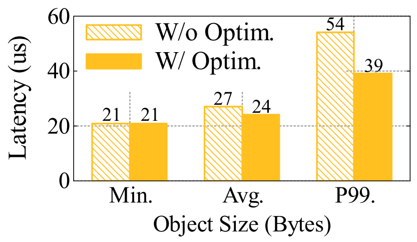


Fig. 15. Performance of small objects optimization.

achieves almost the same performance as REP3. For degraded read latency, MicroEC outperforms MemoryEC/AsyncEC and reduces the average latency by up to 47.67%. Compared with REP3, all EC policies have higher latencies as they need to read surviving blocks and perform decoding. However, the latency gap between MicroEC and REP3 is reasonable for large objects, e.g., for 16 MB objects, the degraded read latency of MicroEC is 2.96 ms, and that of REP3 is 2.58 ms.

Recovery: For recovery, throughout is a more critical factor, so we show the throughput of recovering a single block and double blocks in Fig. 13. Due to the same implementation of recovery for MemoryEC and AsyncEC, we show their performance in the same bar. We find that MicroEC achieves higher throughput than MemoryEC/AsyncEC, e.g., it increases the throughput by up to 96.28% and 121.59% for recovering a single block and double blocks, respectively. In addition, compared with REP3, MicroEC also increases the throughput by up to 65.56% and 115.71% for recovering a single block and double blocks, respectively. The reason is that for a failed memory node, EC needs to recover one block for a failed stripe, while REP3 needs to recover the whole large object, thus MicroEC leverages an efficient non-blocking pipeline in all phases and shows high throughput.

C. *Performance Breakdown*

Now we conduct breakdown analysis to show the efficiency of MicroEC. Note that the whole write process of MicroEC consists of four stages, *coding*, *packing*, *transferring* and *sync*, and we measure the elapsed time in each stage by ignoring the pipelining effect. The results are shown in Fig. 14(a). We find that the coding phase takes only 318 μs, which already matches the transmission time (315 μs), implying that the coding and RDMA transmission are well pipelined with negligible waiting time. We also find that the packing phase only costs 2.6 μs, which is negligible to the total time. This implies that the address management by maintaining the data structures in MicroEC adds very limited overhead. In summary, with MicroEC, most of the time cost is contributed by the coding and RDMA transmission, which are pipelined very well. This breakdown analysis also justifies why MicroEC performs even much better than REP3.

To further study the benefit of the exponential pipeline (see Section IV-D), we also compare MicroEC with a baseline that implements pipeline with fixed-size substripes. Fig. 14(b) shows the results of writing 1 MB objects, and we vary the size of substripes from 16 KB to 1 MB. We find that MicroEC still reduces the latency by 35.76% even compared with the baseline under the best setting (i.e., 512 KB substripe). This improvement comes from the exponential pipeline design, which minimizes the latency of the first and last rounds in the pipeline workflow. Besides, we find that for pipeline with fixed-size substripes, splitting the object into too many small subobjects does not reduce the end-to-end latency, but significantly increases the latency. This is because it induces too many RDMA round trips and thus increases the overall latency.

Finally, we study the performance of MicroEC in optimizing small objects, as described in Section IV-E. We use a client to send 1 K write requests for 1 KB objects with and without dispatching the coding tasks of consecutive requests to the same core, and the minimum/average/P99 latencies are shown in Fig. 15. While the average latency exhibits only a modest reduction of 10.17% with the small objects optimization, the P99 latency reduction is more significant, e.g., 27.81%. This implies that the performance of writing small objects becomes more stable with the implemented optimization.

D. *Sensitivity Analysis*

Impact of Coding Parameters: Fig. 16(a)–(b) show the impact of different codes by varying k and m. We also compare with replication realizing the same level of fault tolerance, i.e., REP represents 3-way replication in Fig. 16(a) and 2-way/3-way/4-way replication in Fig. 16(b) when $m = 1, 2, 3$, respectively.

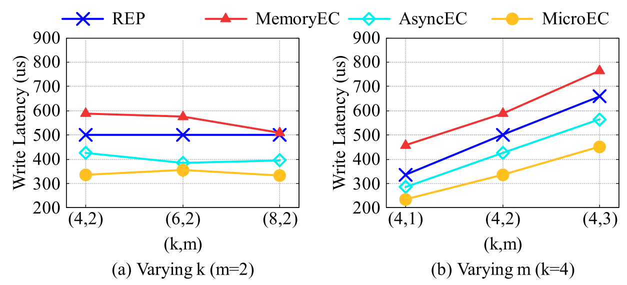


Fig. 16. Varying the code parameters (k, m).

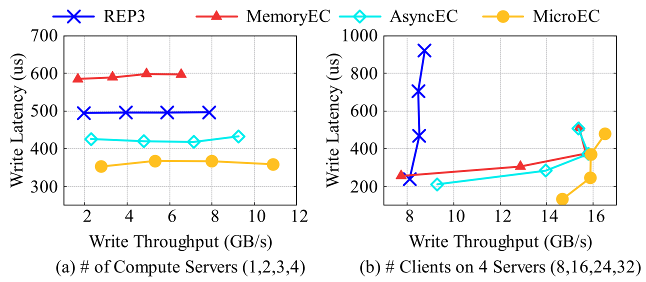


Fig. 17. Varying the number of compute servers and clients.

Results show that increasing k reduces the improvement ratio while increasing m enlarges the improvement ratio. Thus, MicroEC benefits more from a higher level of fault tolerance. This demonstrates the usefulness of MicroEC as more and more applications demand higher reliability than that of 3-way replication.

Impact of Multiple Clients: We first let each compute server run only one client, so each client uses a dedicated RNIC for data transmission. Fig. 17(a) shows the latency (y-axis) and throughput (x-axis) by varying the number of servers from 1 to 4. We see that using more servers can continuously increase the throughput without sacrificing the latency too much for all designs. This is because the clients use different RNICs, the write competition is not serious. MicroEC outperforms all existing designs, and the improvement of throughput becomes larger as the number of compute servers increases.

Fig. 17(b) further shows the results of running multiple clients on each server, and each client keeps performing requests of writing 256 KB objects. We deploy 8, 16, 24, and 32 clients on 4 computer servers. Each server isolates two CPU cores for coding threads, and they share the same RNIC. Note that the throughput of replication keeps stable for 16, 24, and 32 clients because writing redundant copies also consumes network bandwidth. Although existing EC designs improve the throughput with multiple clients, the write latency also increases due to network competition. Besides, compared with MemoryEC/AsyncEC with 24 clients, MicroEC with 16 clients achieves 1.01×/1.005× throughput, while reducing the latency by 34.27%/34.05%. That is, MicroEC can achieve the same throughput as that of MemoryEC/AsyncEC with fewer clients, so it mitigates the competition and realizes lower latency. Finally, when using 24 clients, as the network bandwidth becomes the bottleneck, the improvement of MicroEC diminishes.

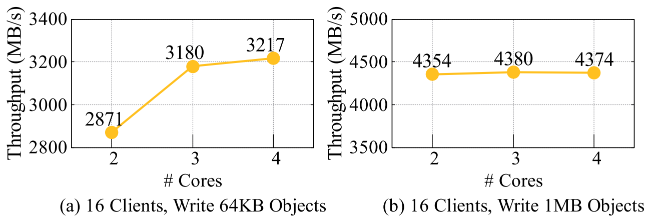


Fig. 18. Impact of the number of isolated cores.

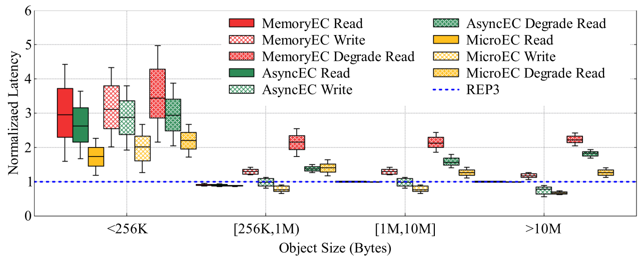


Fig. 19. Performance under IBM production workload.

Impact of the Number of Isolated CPU Cores: We set 16 clients running on one server to keep issuing write requests and evaluate the throughput by varying the number of isolated cores. Fig. 18 shows the throughput when writing small and large objects, respectively. We find that concurrent small I/Os require more isolated cores for coding. For example, the throughput increases by 10.76% when the number of isolated cores increases from two to three, as the contention among coding threads is relaxed with more isolated cores. When the number of isolated cores keeps increasing, the write throughput keeps almost stable, because the network bandwidth becomes the bottleneck in this case. For writing large objects of 1 MB, using two isolated cores is adequate to realize a stable performance. That is, using two cores to process requests of writing 1 MB objects from 16 clients already reaches the upper bound of the network bandwidth. Thus, we can isolate fewer cores for coding in the case of large objects.

E. Application Benchmarks

IBM Docker Registry Production Workload [4], [78]: We focus on the trace of Dallas data center as it has the highest load, and replay the requests in an 18-hour duration. The object size varies from 1 KB to 660.2 MB, and the average size is 11.6 MB. The total data size is 124.13 GB, and 96% of requests are reads. To include evaluating the performance of degraded read, we randomly disable a memory server and the correlated read requests become degraded read requests. Fig. 19 shows the latency. We categorize requests into multiple groups according to the object size, and normalize the average latency of REP3 as one, as it always has the lowest latency for degraded reads. For each EC policy, the closer to the blue line, the better its performance is. We see that MicroEC always outperforms MemoryEC and AsyncEC, e.g., for the group of >10 MB objects, MicroEC reduces the average latency by 26.58% and 18.28%, respectively. In addition, MicroEC also achieves comparable performance with REP3 for large objects.

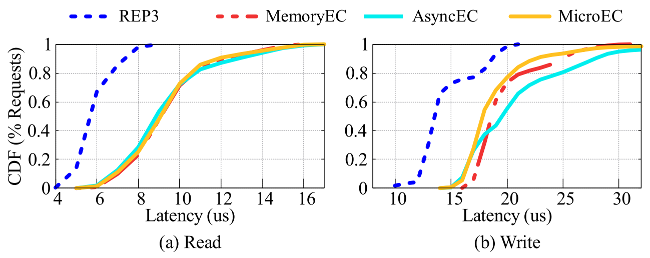


Fig. 20. Performance under YCSB benchmark.

YCSB Workload [12]: To show the performance of MicroEC under small objects, we run YCSB, which is a general-purpose benchmark widely used in key-value stores. Limited by space, we only show the results under YCSB A, which is a read-write mixed workload containing 50% reads and 50% writes, so as to study the influence of both read and write. We configure the workload to issue 40 K requests and set the object size as 4 KB as previous works [68], [94]. We run four concurrent clients on four compute servers, which share the same memory pool. Fig. 20 shows the CDF of the read/write latencies. All EC schemes achieve similar performance and have higher latency compared to the REP3 scheme for both read and write requests. Specifically, the latency of MicroEC is approximately $1.39\times$ higher for read requests and $1.23\times$ higher for write requests compared to REP3. This latency increase is primarily attributed to the additional software stack overhead introduced by the EC schemes, such as the submission and synchronization of $k+m$ connections, as analyzed in Section VI-B1.

VII. Related Work

Disaggregated Memory: The disaggregated memory has received widespread attention due to providing significant benefits for modern datacenters on elasticity and resource utilization. Existing works explore disaggregated memory in hardware architectures [40], [41], networks [20], [22], [70], indexes [35], [47], [80], [99], KV stores [32], [67], [75], transactions [95], caching [66], and memory management [24], [33], [76], [79]. MicroEC focuses on the design of efficient write/read workflows while deploying EC in disaggregated memory, which is orthogonal to these works.

EC for in-Memory Storage Systems: There are many works on deploying EC in in-memory systems. For example, some works explore EC for in-memory key-value stores, including EC designs for efficiency and consistency [39], [65], [94], redundancy transitioning of resilient fault-tolerance [74], [86] and parity update optimization [10], [37]. There are also some works leveraging EC in memory for object caching [58], [78]. These works focus on the cost-effectiveness of EC, while MicroEC mainly optimizes the pipeline of the coding and RDMA transmission so as to minimize the total delay of write/read for deploying EC in disaggregated memory.

EC in Remote Memory: Hydra [34] and Carbink [97] are two recent works addressing fault tolerance in remote memory systems, where all servers are configured with individual memory and contribute their spare memory forming a memory pool. They treat remote memory as a swap device for local memory and swap erasure-coded local data to remote when local memory pressure is high. MicroEC differs from them in two aspects. First, MicroEC focuses on disaggregated memory where compute servers are configured with limited memory, and data should be directly written/read to/from remote memory servers with one-sided RDMA. Second, the two approaches only provide fault tolerance for the swapped-out data, while the data in local memory should be protected by additional techniques, such as writing checkpoints to a persistent storage device like SSD, which causes the periodically pause of applications due to recording checkpoints and the slow recovery of local memory data due to accessing the persistent storage device [45]. In MicroEC, all successfully written erasure-coded data is directly stored on memory servers, which can be instantly accessed by other clients.

EC for Secondary Storage Systems: EC is widely studied and optimized for secondary storage systems, including RAIDs and distributed storage systems like GFS [21], Ceph [85], HDFS [71] and Azure [9]. In secondary storage systems, I/O usually is the bottleneck of system performance. So the major efforts of these works focus on reducing the high overhead of cross-node network traffic and disk I/Os [27], [56], [59], [61], [77], [87]. However, MicroEC focuses on the timely write/read latency in disaggregated memory.

EC Pipelines: Towards making EC viable and efficient for large-scale storage systems, multiple works focusing on designing pipelines along different directions have been proposed [36], [38], [52], [68], [69]. One direction is to design repair pipelining, which divides the repair of a block into partial operations [52] or small slices [36], [38] and parallelizes them for improved repair performance. Another direction is to design EC offloading pipelines coping with modern SmartNICs for taking full advantage of SmartNICs and reducing CPU involvement [68], [69]. Different from the background and challenges of the existing pipelining, MicroEC focuses on designing the pipeline of the write/read for deploying EC in disaggregated memory to overlap the latency of coding and RDMA transmission.

Efficient Code Schemes: MicroEC focuses on efficiently deploying CRS codes in disaggregated memory with the widely used Intel ISA-L library [14], because it provides highly optimized instructions and fast coding speed [13]. CRS codes are widely studied in previous works and deployed in many industrial systems [19], [26], [34], [77], [78], [96]. There are some codes with lower computational complexity and lower coding latency, such as the industry-standard RAID6 codes [3] based on XOR operations. However, RAID6 only tolerates double failures. And because it strongly depends on the system scale, it is not so flexible, especially not applicable in systems with dynamic scale and resilient fault tolerance.

VIII. Conclusion

This paper presented MicroEC, which realized an efficient pipeline of the coding and RDMA transmission and evidently reduced the total delay of write/read with EC in disaggregated memory. Its efficient pipeline comes from the designed coding workflow and the careful coordination of the coding and RDMA transmission. Experiments demonstrate that MicroEC improves the write, degraded read, and recovery performance, and outperforms existing EC designs significantly. However, it is a great challenge to limit the latency of writing and reading small objects with EC.

REFERENCES

[1] M. K. Aguilera et al., "Remote memory in the age of fast networks," in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 121–127.

[2] E. Amaro et al., "Can far memory improve job throughput?," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, Art. no. 14.

[3] H. P. Anvin, "The mathematics of RAID-6," 2011. [Online]. Available: http://alamos.math.arizona.edu/RTG16/ECC/raid6.pdf

[4] A. Anwar et al., "Improving docker registry design based on production workload analysis," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 265–278.

[5] M. S. Birrittella et al., "Intel omni-path architecture: Enabling scalable, high performance fabrics," in *Proc. 23rd Annu. Symp. High- Perform. Interconnects*, 2015, pp. 1–9.

[6] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," Univ. California at Berkeley, Berkeley, CA, Tech. Rep. TR-95–048, 1995.

[7] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 80–107, 1996.

[8] Q. Cai et al., "Efficient distributed memory management with RDMA and caching," *VLDB Endowment*, vol. 11, no. 11, pp. 1604–1617, 2018.

[9] B. Calder et al., "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 143–157.

[10] L. Cheng, Y. Hu, and P. P. Lee, "Coupling decentralized key-value stores with erasure coding," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 377–389.

[11] G.-Z. Consortium, Gen-Z, 2019. [Online]. Available: https://genzconsortium.org/

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 1430–154.

[13] I. Corporation, "Intel(R) intelligent storage acceleration library performance report," 2017. [Online]. Available: https://01.org/sites/default/files/documentation/intel_isa-l_2.19_performance_report_0_0.pdf

[14] I. Corporation, "Intel(R) intelligent storage acceleration library," 2021. [Online]. Available: https://github.com/intel/isa-l

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating System Des. Implementation*, 2004, pp. 137–150.

[16] Dormando, "Memcached," 2018. [Online]. Available: https://memcached.org/,

[17] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Symp. Networked Syst. Des. Implementation*, 2014, pp. 401–414.

[18] Facebook, "RocksDB," 2022. [Online]. Available: https://rocksdb.org

[19] D. Ford et al., "Availability in globally distributed storage systems," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 61–74.

[20] P. X. Gao et al., "Network requirements for resource disaggregation," in *Proc. 12th USENIX Symp. operating Syst. Des. implementation*, 2016, pp. 249–264.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.

[22] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, high-performance memory disaggregation with directcxl," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 287–294.

[23] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *Proc. 14th USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 649–667.

[24] Z. Guo, Y. Shan, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proc. 27th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 417–433.

[25] Hadoop, "HDFS erasure coding," 2020. [Online]. Available: https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html

[26] Y. Hu, L. Cheng, Q. Yao, P. P. Lee, W. Wang, and W. Chen, "Exploiting combined locality for wide-stripe erasure coding in distributed storage," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021.

[27] C. Huang et al., "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 15–26.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.

[29] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 437–450.

[30] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 185–201.

[31] Y. Le, B. Stephens, A. Singhvi, A. Akella, and M. M. Swift, "RoGUE: RDMA over generic unconverged ethernet," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 225–236.

[32] S. K. Lee, S. Ponnapalli, S. Singhal, M. K. Aguilera, K. Keeton, and V. Chidambaram, "DINOMO: An elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, vol. 15, no. 13, pp. 4023–4037, 2022.

[33] S.-S. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "MIND: In-network memory management for disaggregated data centers," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 488–504.

[34] Y. Lee, H. A. Maruf, M. Chowdhury, A. Cidon, and K. G. Shin, "Hydra: Resilient and highly available remote memory," in *Proc. 20th USENIX Conf. File Storage Technol.*, 2022, pp. 181–198.

[35] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems," in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023, pp. 99–114.

[36] R. Li, X. Li, P. P. Lee, and Q. Huang, "Repair pipelining for erasure-coded storage," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 567–579.

[37] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "BCStore: Bandwidth-efficient in-memory KV-store with batch coding," in *Proc. IEEE 33rd Int. Conf. Massive Storage Syst. Technol.*, 2017.

[38] X. Li et al., "ParaRC: Embracing sub-packetization for repair parallelization in MSR-coded storage," in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023.

[39] Y. Li, J. Zhou, W. Wang, and Y. Chen, "RE-store: Reliable and efficient KV-store with erasure coding and replication," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2019, pp. 1–12.

[40] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH Comput. Archit. news*, vol. 37, no. 3, pp. 267–278, 2009.

[41] K. Lim et al., "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High- Perform. Comp Archit.*, 2012, pp. 1–12.

[42] C. E. Link, "Compute express link: The breakthrough cpu-to-device interconnect," 2022. [Online]. Available: https://www.computeexpresslink.org/

[43] Linux, "Perf," 2009. https://perf.wiki.kernel.org/

[44] M. Littley et al., "Bolt: Towards a scalable docker registry via hyperconvergence," in *Proc. 12th Int. Conf. Cloud Comput.*, 2019, pp. 358–366.

[45] K. Liu, J. Kosaian, and K. V. Rashmi, "ECRM: Efficient fault tolerance for recommendation model training via erasure coding," 2021, *arXiv:2104.01981*.

[46] R. Ltd, "Redis," 2022. [Online]. Available: https://redis.io/

[47] X. Luo et al., "SMART: A high-performance adaptive radix tree for disaggregated memory," in *Proc. 17th USENIX Symp. Operating Syst. Des. Implementation*, 2023, pp. 553–571.

[48] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam, Netherlands: Elsevier, 1977.

[49] H. A. Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proc. USENIX Annu. Tech. Conf.*, 2020.

[50] D. Memory, "DDR4 Memory Standard," 2014. [Online]. Available: https://www.kingston.com/en/memory/ddr4-overview

[51] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 103–114.

[52] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 30:1–30:16.

[53] P. Moritz et al., “Ray: A distributed framework for emerging AI applications,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 561–577.

[54] NVIDIA, “The NVIDIA mellanox connectX-6 SmartNIC,” 2022. [Online]. Available: https://www.nvidia.com/en-us/networking/ethernet/connectx-6

[55] J. Ousterhout et al., “The RAMCloud storage system,” *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 1–55, 2015.

[56] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic, “Opening the chrysalis: On the real repair performance of MSR codes,” in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 81–94.

[57] J. S. Plank and L. Xu, “Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications,” in *Proc. IEEE 5th Int. Symp. Netw. Comput. Appl.*, 2006, pp. 173–180.

[58] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, “EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding,” in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 401–417.

[59] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, “Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth,” in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 81–94.

[60] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, “A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster,” in *Proc. 5th USENIX Workshop Hot Topics Storage File Syst.*, 2013, pp. 8–8.

[61] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, “A “hitchhiker’s” guide to fast and efficient data reconstruction in erasure-coded data centers,” in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2014, pp. 331–342.

[62] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.

[63] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, “AIFM: High-performance, application-integrated far memory,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 315–332.

[64] J. D. S. Ghemawat, “LevelDB,”2011. [Online]. Available: https://github.com/google/leveldb

[65] D. Shankar, X. Lu, and D. K. Panda, “High-performance and resilient key-value store with online erasure coding for Big Data workloads,” in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 527–537.

[66] J. Shen et al., “Ditto: An elastic and adaptive memory-disaggregated caching system,” in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 675–691.

[67] J. Shen et al., “FUSEE: A fully memory-disaggregated key-value store,” in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023, pp. 81–97.

[68] H. Shi and X. Lu, “TriEC: Tripartite graph based erasure coding NIC offload,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 44:1–44:34.

[69] H. Shi and X. Lu, “INEC: Fast and coherent in-network erasure coding,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–17.

[70] V. Shrivastav et al., “Shoal: A network architecture for disaggregated racks,” in *Proc. 16th USENIX Symp. Networked Syst. Des. Implementation*, 2019, pp. 1–17.

[71] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[72] A. Software Foundation, “Crail iobench,” 2018. [Online]. Available: https://incubator-crail.readthedocs.io/en/latest/iobench.html

[73] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, “Unification of temporary storage in the nodekernel architecture,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 767–781.

[74] K. Taranov, G. Alonso, and T. Hoefler, “Fast and strongly-consistent per-item resilience in key-value stores,” in *Proc. 13th Eur. Conf. Comput. Syst.*, 2018, Art. no. 39.

[75] S.-Y. Tsai, Y. Shan, and Y. Zhang, “Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores,” in *Proc. Annu. Tech. Conf.*, 2020, pp. 33–48.

[76] S.-Y. Tsai and Y. Zhang, “LITE kernel RDMA support for datacenter applications,” in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 306–324.

[77] M. Vajha et al., “Clay codes: Moulding MDS codes to yield an MSR code,” in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 139–153.

[78] A. Wang et al., “INFINICACHE: Exploiting ephemeral serverless functions to build a cost-effective memory cache,” in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 267–282.

[79] C. Wang et al., “Semeru: A memory-disaggregated managed runtime,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, Art. no. 15.

[80] Q. Wang, Y. Lu, and J. Shu, “Sherman: A write-optimized distributed b+ tree index on disaggregated memory,” in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1033–1048.

[81] S. Wang et al., “Ownership: A distributed futures system for fine-grained tasks,” in *Proc. 18th USENIX Symp. Networked Syst. Des. Implementation*, 2021, pp. 671–686.

[82] Z. Wang, G. Zhang, Y. Wang, Q. Yang, and J. Zhu, “Dayu: Fast and low-interference data recovery in very-large storage systems,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 993–1008.

[83] X. Wei, R. Chen, and H. Chen, “Fast RDMA-based ordered key-value store using remote learned cache,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 117–135.

[84] X. Wei, Z. Dong, R. Chen, and H. Chen, “Deconstructing RDMA-enabled distributed transactions: Hybrid is better!,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 233–251.

[85] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.

[86] S. Wu, Z. Shen, and P. P. Lee, “Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic reed-solomon codes,” in *Proc. IEEE Int. Symp. Reliable Distrib. Syst.*, 2020, pp. 246–255.

[87] L. Xiang, Y. Xu, J. C. Lui, and Q. Chang, “Optimal recovery of single disk failure in RDP code storage systems,” *ACM Special Int. Group Perform. Eval. Rev.*, vol. 38, no. 1, pp. 119–130, 2010.

[88] J. Yang, A. Sabnis, D. S. Berger, K. Rashmi, and R. K. Sitaraman, “C2DN: How to harness erasure codes at the edge for efficient content delivery,” in *Proc. 19th USENIX Symp. Networked Syst. Des. Implementation*, 2022, pp. 1159–1177.

[89] J. Yang, Y. Yue, and K. Rashmi, “A large scale analysis of hundreds of in-memory cache clusters at twitter,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 1–35.

[90] J. Yang, Y. Yue, and R. Vinayak, “Segcache: A memory-efficient and scalable in-memory key-value cache for small objects,” in *Proc. 18th USENIX Symp. Networked Syst. Des. Implementation*, 2021, pp. 503–518.

[91] M. M. Yiu, H. H. Chan, and P. P. Lee, “Erasure coding for small objects in in-memory KV storage,” in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, Art. no. 14.

[92] Y. Yu, R. Huang, W. Wang, J. Zhang, and K. B. Letaief, “SP-cache: Load-balanced, redundancy-free cluster caching with selective partition,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 1–13.

[93] M. Zaharia et al., “Apache spark: A unified engine for Big Data processing,” *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[94] H. Zhang, M. Dong, and H. Chen, “Efficient and available in-memory KV-store with hybrid erasure coding and replication,” in *Proc. 14th USENIX Conf. File Storage Technol.*, 2017, Art. no. 25.

[95] M. Zhang, Y. Hua, P. Zuo, and L. Liu, “FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory,” in *Proc. 20th USENIX Conf. File Storage Technol.*, 2022, Art. no. 31.

[96] T. Zhou and C. Tian, “Fast erasure coding for data storage: A comprehensive study of the acceleration techniques,” in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, Art. no. 7.

[97] Y. Zhou et al., “Carbink: Fault-tolerant far memory,” in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 55–71.

[98] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, “One-sided RDMA-conscious extendible hashing for disaggregated memory,” in *Proc. USENIX Annu. Tech. Conf.*, 2021, Art. no. 11.

[99] P. Zuo et al., “Race: One-sided RDMA-conscious extendible hashing,” *ACM Trans. Storage*, vol. 18, no. 2, pp. 11:1–11:29, 2022.

Qiliang Li received the BS degree from the School of Computer Science and Technology, Anhui University, in 2019. He is currently working toward the PhD degree with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include designing and building dependable techniques for large-scale storage systems.

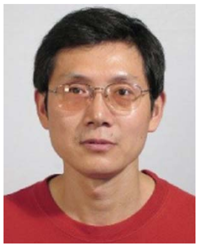
Liangliang Xu received the BS degree from the Department of Information and Computational Science, Anhui University, in 2017, and the PhD degree from the School of Computer Science and Technology, University of Science and Technology of China, in 2022. His research interests include distributed storage systems, data recovery, and erasure coding.

Yongkun Li (Member, IEEE) received the BEng degree in computer science from USTC, in 2008, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong, in 2012. He is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. After that, he worked as a postdoctoral fellow with the Institute of Network Coding, Chinese University of Hong Kong. His research interests include focuses on memory and file systems, including key-value systems, distributed file systems, as well as memory and I/O optimization for virtualized systems, and cloud computing.

Min Lyu (Member, IEEE) received the BS and MS degrees in applied mathematics from Anhui University, in 1999 and 2002, respectively, and the PhD degree in applied mathematics from the University of Science and Technology of China, in 2005. She is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. Her research interests include distributed storage systems, social networks, and security and privacy.

Wei Wang received the BS degree from the School of Software Engineering, South China University of Technology, China, in 2019. He is currently working toward the PhD degree with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include distributed storage systems, erasure coding, and fault-tolerance.

Pengfei Zuo received the BS and PhD degrees in computer science and technology from the Huazhong University of Science and Technology, Wuhan, in 2014 and 2019, respectively. His research interests include memory systems, storage systems and techniques, and security.



Yinlong Xu received the BS degree in mathematics from Peking University, Beijing, China, in 1983, and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC), Hefei, China, in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, USTC. He served the Department of Computer Science and Technology, USTC as an assistant professor, a lecturer, and an associate professor. He is currently leading a group in doing some networking, storage and high performance computing research. His current research interests include storage system, file system, social network, and high performance I/O. He was a recipient of the Excellent Ph.D. Advisor Award of the Chinese Academy of Sciences in 2006 and Baosteel Excellent Teacher Award in 2014.