FineMem: Breaking the Allocation Overhead vs. Memory Waste Dilemma in Fine-Grained Disaggregated Memory Management

Paper ID: 686

Abstract

RDMA-enabled memory disaggregation has emerged as an attractive approach to reducing memory costs in modern data center. While RDMA enables efficient remote read/write operations, it presents challenges in remote memory (de)allocation. Consequently, existing systems adopt coarse-grained allocations (in units of GBs), leading to memory waste.

We introduce FineMem, an RDMA-connected remote memory management system that enables high-performance, fine-grained memory allocation. FineMem addresses latency and scalability challenges for fine-grained allocations. It removes RDMA memory region (MR) registration costs from allocation paths through per-compute node MR pre-registration, while ensuring remote memory isolation using RDMA memory windows and a trusted allocation service on each compute node. It employs a lock-free, one-sided RDMA-based protocol to allocate memory chunks (e.g. 4KB, 2MB) without involving the memory node's CPU and maintains metadata consistency during compute node failures via logging. We show that FineMem reduces remote memory allocation latency by up to 95% compared to state-of-the-art remote memory management systems. It enables applications, Key-Value stores, and swap systems running on FineMem to achieve low memory waste with minimal overhead.

1 Introduction

The growing need for hyperscalers to reduce memory costs [17, 35, 38] has made memory disaggregation an attractive system design. Unlike traditional architectures that tightly couple memory with compute resources – often leading to memory underutilization [35], disaggregated memory (DM) decouples memory (memory nodes) from compute (compute nodes), and enables dynamic memory allocation across multiple compute nodes. Over the past decade, memory disaggregation has emerged as a pivotal area of systems research [8, 22, 31, 35, 53, 54, 56, 65].

Advancements in high-speed interconnect technologies, particularly Remote Direct Memory Access (RDMA) [45], have made it feasible to implement DM systems effectively. RDMA, facilitated by RDMA-capable NICs (RNICs [25, 41]), allows compute nodes to perform direct memory read/write operations on remote memory nodes using one-sided operations, bypassing the memory node's CPU entirely. Minimizing CPU involvement on memory nodes is critical for achieving low latency and high throughput, especially given their limited processing power [36, 50, 57, 65].

Despite RDMA's strengths in data access, it presents significant challenges in efficient memory allocation and deallocation. Allocating memory from a memory node involves several operations: the memory node OS needs to mmap() (pre-fault) physical memory pages, registering/copying page table entries to the memory transaction table (MTT) in the RNIC, and generate on-device memory management units with capabilities, named as memory regions (MR [4]) with rkey [47]. These operations are notably time-consuming and must run on memory node CPUs — for instance, registering a 4MB memory region can take upwards of 480 µs.

To work around RDMA's inefficiencies in memory (de)allocation, existing systems [3, 14, 50, 62, 65] resort to coarse allocation strategies, often grabbing chunks of 1 GB or more at a time. While this approach amortizes the overhead of allocations, it comes at a steep cost. Large memory chunks lead to considerable memory underutilization, as unused portions of these coarse chunks cannot be reclaimed or shared. This leaves RDMA-based DM systems stuck in a hard dilemma: either suffer crippling allocation overheads or tolerate memory waste.

In this paper, we introduce FineMem, a distributed remote memory management system that delivers fine-grained memory allocation and deallocation (e.g. 4KB/2MB/...) with minimal memory waste and high performance, leveraging one-sided RDMA. FineMem supports flexible allocation granularities from diverse applications within a shared disaggregated memory pool. By using one-sided RDMA, FineMem bypasses the memory node's CPU during allocations, ensuring it does not become a scalability bottleneck [36, 50, 57, 62, 65]. FineMem tackles three key challenges: 1) removing MR reg-

istration latency from the application's allocation path, 2) enabling efficient and highly concurrent allocations for free remote memory chunks, and 3) ensuring metadata consistency in the face of compute node failures.

To remove MR registration overheads during allocation, FineMem builds upon a strawman approach while addressing critical isolation challenges. The strawman method pre-registers the entire remote memory space as a single MR when a compute node boots. When an application process starts on the compute node, the memory node shares the MR rkey with the process, allowing it to allocate/access chunks from the pre-registered MR. While this strawman approach removes MR registration from application, it introduces significant isolation issues. An application can access any remote memory belonging to other applications. Additionally, coordinating (de)allocation across applications requires sharing the same memory management metadata region (which contains information on free or used memory chunks), allowing any application to read/modify this metadata. This lack of protection poses serious security risks to the entire DM system.

FineMem first uses the memory window (MW [13]) feature in RDMA to ensure memory access isolation across applications. The memory node pre-binds a MW to each chunk with specific rkey. When (de)allocating fine-grained chunks, FineMem uses one-sided operations to acquire/invalidate rkeys, thus removes bind-related RPC at allocation critical-path. Second, FineMem employs a per-compute node allocation service to protect remote memory management metadata (e.g. index, redo-log and rkey). Applications use inter-process communication to interact with the allocation service for (de)allocation, and only the allocation service holds the rkey necessary to read/modify the memory management metadata region.

To enable efficient, highly concurrent allocations for remote memory chunks across compute nodes, FineMem employs a carefully designed two-layer bitmap tree to accelerate allocation. It effectively limits the number of RDMA round-trips to find a free chunk. Additionally, FineMem embeds contention control information into the bitmap, avoids further contention, and reduces the number of retries for allocations' compare-and-swap (CAS) update, resulting in more predictable latency.

Finally, to ensure metadata crash consistency in the event of compute node failures, FineMem executes allocation in two steps: 1) a commit point for allocation success, combined with temporary redo-log information; and 2) flushing logs and updating related metadata while detecting inconsistencies using timestamps and commit point metadata. This mechanism guarantees metadata consistency for the search index and redo-log, enabling reliable failure recovery.

We have implemented FineMem on Linux and we show its portability across various systems, including application user-space allocation libraries (such as jemalloc [18], mimalloc [33]), DM-native applications (such as Key-Value stores [36, 50, 65]) and kernel swap systems [2, 3, 22, 56]. Based on mimalloc [33], we built FineMem-User, a user-space DM object allocation library that uses FineMem's DM allocation APIs. We also ported a popular swap system, FastSwap [3], and a DM-native Key-Value store [50] to use FineMem (Sec. 5). Our evaluations show that FineMem reduces up to 95% latency for remote memory allocation compared to state-of-the-art. For applications, FineMem improves memory utilization by 2.25× to 28× compared to coarse-grained memory management, all while introducing minimal overheads of about only 2.5%-4.1%.

In the rest of this paper, we demonstrate existing memory management systems for DM and their limits (Sec. 2); discuss the FineMem design (Sec. 3, 4, 5); characterize FineMem's performance with allocator benchmarks and end-to-end applications (Sec. 6); discuss limitations and potential extensions of FineMem (Sec. 7); review of related work (Sec. 8); and conclude (Sec. 9). Our implementation can be accessed at `https://anonymous.4open.science/r/FineMem-3C02/` (an anonymous open-source repository, which will be replaced with GitHub repo upon publication).

2 Background and Motivation

2.1 Remote Memory Management and RDMA

Memory disaggregation introduces a remote memory pool that is shared by compute nodes. In this work, we focus on RDMA-connected DM, where processes on compute nodes can access a shared memory node using RDMA primitives.

RDMA supports two communication patterns: Remote Procedure Call (RPC) using send/receive operations, and direct remote memory access through one-sided read/write that bypass the receiver cores. One-sided operations require external memory authorities on RDMA NICs. Before clients can perform one-sided operations, a memory region (MR) must be registered [4] with the RNIC. During registration, the driver pins the physical memory pages and maintains page mappings in an on-chip page table [14]. The registration process returns a region capability (rkey) that allows clients to directly access the MR. Clients within the same Protected Domain (PD) and with the appropriate rkey can freely read/write the contents of the MR. Consequently, malicious clients possessing the correct rkey could launch attacks by modifying the MR contents.

DM requires a memory management system to allocate and free memory according to applications demands. Similar to memory management in single-node operating systems, processes interact with the DM memory management system to (de)allocate memory chunks (e.g. 4KB or 2MB). However, unlike single-node OS memory management, memory (de)allocation over RDMA are costly.

2.2 Remote Memory Allocation Challenges with RDMA

Memory allocation over RDMA differs from local memory allocation in two significant ways: the need for MR registration and the (unsurprising) fact that allocations occur over the network. In this section, we quantitatively demonstrate how these two factors impact application performance.

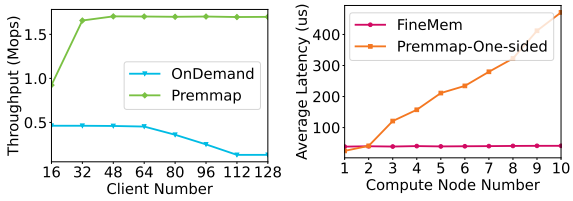
Memory Region Registration is Costly. The overhead associated with RDMA MR registration is a well-documented problem [14, 23, 31, 40, 49, 54]. MR registration involves pinning physical memory pages and updating the RNIC's page tables, which incurs significant overhead (e.g. a 4MB registration takes 480us).

In Fig. 1(a), we illustrate the impact of MR registration costs to end-to-end application performance using a DM KV store, FUSEE [50]. We compare on-demand MR registration for each 4KB chunk allocation with a baseline using a pre-registered large memory region (Premmap, no MR registration during runtime). In a YCSB-A workload (search:update=50:50), the on-demand approach achieves only 26.7% of the throughput of Premmap (64 clients), highlighting the impracticality of frequent MR registration during runtime for remote memory allocations.

Frequent Remote Allocations Over the Network are Costly. Beyond MR registration, handling remote allocations presents challenges because the operations occur over the network, introducing additional latency and potential bottlenecks.

Systems that delegate memory allocation to memory nodes via RPC (e.g. FUSEE [50]) face scalability issues, as memory nodes often have limited processing power. Memory allocation involves traversing metadata, and frequent (de)allocations impose significant processing demands. Similar to prior works [36, 57, 65], we found that RPC-based approach becomes bottlenecked by the memory node's compute power as more clients are added. For example, in Fig. 1(a), FUSEE's throughput (even with Premmap) plateaus after 32 clients; FUSEE OnDemand encounters additional scalability issues when the number of clients exceeds 64. This highlights the need for a one-sided RDMA-based allocation approach, as used by systems like CXL-SHM [62] and our proposed FineMem.

While avoiding bottlenecks at memory nodes, existing systems using one-sided access for allocations still experience unpredictable network round-trips due to their metadata designs. In Fig. 1(b), we evaluated a general allocation benchmark from ThreadTest [6], where multiple threads repeatedly allocate and free chunks from the memory node. To isolate the effects, we pre-registered sufficient memory to eliminate MR registration costs. As shown, the latency of existing one-sided approaches (e.g. premmap-one-sided built on CXL-SHM [62]) deteriorates significantly as the number of clients increases. This degradation is caused by a high number of retries over the network for each allocation, which occur when there are many concurrent allocations (details in Sec. 3.1).



(a) MR Registration impact to KV store.

(b) Chunks Allocation Latency of One-sided Approach.

Figure 1: Remote Memory Allocation Challenges with RDMA: (a) MR registration overheads. (b) Scalability issues in existing one-sided RDMA-based approach (due to metadata design).

In summary, remote memory allocation and deallocation over RDMA face both MR registration cost challenge and scalability issues due to network overhead and the limited processing power of memory nodes.

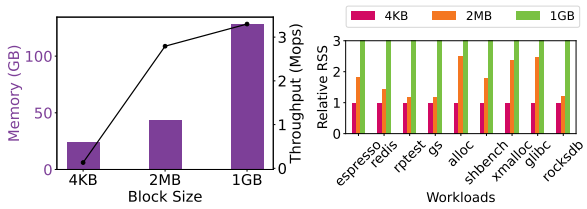
2.3 The Allocation Overhead vs. Memory Waste Dilemma in Existing Works.

Due to the challenges with MR registration and other operations, existing systems either rely on statically pre-mmaped regions [26, 48, 48, 50, 62, 65] or allocate memory from the remote memory pool at a coarse granularity (e.g. GBs) [2, 3, 8, 14, 22, 43, 51, 56, 60]. However, these approaches can introduce significant memory waste. Next, we characterize the tradeoff between allocation overhead and memory waste using KV store and general workloads.

KV store. Using the YCSB-A workload (50% updates, with all key-value pairs residing in remote memory), we evaluated FUSEE's performance and remote memory usage with different chunk allocation units (4KB, 2MB, and 1GB), as shown in Fig. 2(a). In FUSEE, updates are performed out-of-place, triggering memory allocation and deallocation, as well as memory fragmentation.

The results present a clear tradeoff between allocation overhead and memory waste. For instance, 2MB allocation granularity significantly reduces memory fragmentation (hence memory usage) compared to 1GB (a key optimization also optimized for in FUSEE). However, even with FUSEE's optimizations, reducing the granularity from 1GB to 2MB results in a nearly 17% drop in KV store throughput. This performance decline is attributed to the increased frequency of (de)allocations, each incurring significant overhead and impacting end-to-end performance. Similarly, 4KB allocation granularity further reduces memory fragmentation but exacerbates the tradeoff due to even higher allocation overhead.

General workloads. Fragmentation caused by different allocation granularities is a well-studied issue for general workloads, too. For instance, numerous single-node memory allocator studies have shown that larger page sizes (e.g. 2MB or 1GB) typically result in higher memory fragmentation and



(a) Trade-off on block sizes. (b) Mimalloc with huegpages.

Figure 2: Dilemma in KV Store and General Workloads: (a) KV store reaches better performance with memory wastes. (b) Memory fragmentation with hugepage granularities (1GB's RSS is collapsed to no more than 3.).

waste [33, 42, 46, 51, 52, 64].

In Fig. 2 (b), we used mimalloc to evaluate several benchmarks and applications, illustrating this behavior. The results show that fine-grained allocations (e.g. 2MB) significantly reduce memory fragmentation compared to larger granularities like 1GB. For specific workloads, such as shbench, even finer-grained allocations (4KB) are particularly effective in minimizing memory waste.

Summary. Existing RDMA-based DM systems are caught in a difficult dilemma: either endure significant allocation overheads or accept substantial memory waste.

In this paper, we tackle the question: can fine-grained memory allocation be achieved with minimal overheads? Successfully addressing this challenge would reduce memory waste in disaggregated memory pools, leading to lower memory spending in modern data centers. It would also allow systems under high load to run more tasks concurrently, thereby improving overall throughput, as we will demonstrate in swap system evaluations (Sec. 6.3). Additionally, fine-grained memory allocation simplifies porting emerging applications to disaggregated memory setups. For instance, applications using local memory allocators (e.g. jemalloc, tcmalloc) could seamlessly integrate into disaggregated memory systems by replacing local mmap() calls (usually in units of KB/MB) with remote memory allocations (Sec. 6.3).

Some DM memory management systems have explored on-chip authorization mechanisms for fine-grained memory allocation. For example, MIND [31] offloads memory management to on-path programmable switches [7] and introduces a VMA-based capability model for different processes. In contrast, our goal is to develop a flexible, software-based memory management solution.

3 FineMem Overview

3.1 Design Goals

FineMem is a distributed remote memory management system designed to support fine-grained, high-performance memory (de)allocation in RDMA-connected DM. Its design is driven by three key goals: 1) removing MR registration latency from the application's allocation path; 2) enabling efficient and highly scalable remote memory chunk allocation; and 3) maintaining metadata consistency during compute node failures.

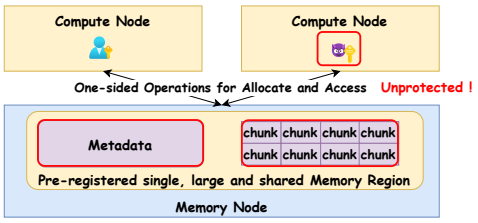


Figure 3: Isolation/Protection Challenges without MR: attacker can easily access other users' chunk and metadata.

Removing MR Registration While Overcoming Critical Isolation Challenges. A straightforward approach to mitigate the overhead of MR registration is to pre-register the entire remote memory as a single accessible MR for each compute node. In this configuration, every application on any compute node can (de)allocate fine-grained memory chunks (subsections of the MR) by directly modifying shared allocation metadata (e.g. which chunks are available or used). This approach eliminates the need for applications to perform MR registration during runtime.

However, unsurprisingly, sharing a single MR introduces significant isolation challenges. Since the MR is associated with a single rkey, any application holding this rkey can access the entire MR, as shown in Fig. 3. This means that any entity can potentially read/write to any other users' memory chunks, leading to *privacy and security risks among applications*. Additionally, *the metadata used for memory allocation is unprotected*; each application must modify it to allocate memory, malicious or faulty applications could corrupt or interfere with the metadata, affecting other applications.

FineMem addresses the application privacy challenges by using the hardware feature of RDMA —Memory Windows (MWs) [13]. Memory windows (MWs) allow for the rapid generation and invalidation of fine-grained rkeys at a rate of one MW per microsecond. The memory node pre-binds a MW to each chunk with a specific rkey and asynchronously regenerate new rkey with background thread. When (de)allocating fine-grained chunks, FineMem uses one-sided operations to acquire and invalidate rkeys, thus removing bind-related RPCs from the critical allocation path. This approach provides isolation between applications, as each application only has access to the MWs for which it possesses the corresponding rkeys. Unlike previous work that binds MWs at runtime with RPC in a single application, such as Patronus [60], FineMem focuses on one-sided MW usage for isolating multiple applications.

To protect the memory allocation metadata from misuse, FineMem introduces a per-compute-node runtime service (inspired by software virtualization techniques [15, 28]). This trusted service delegates remote memory allocation requests on behalf of processes, ensuring that the memory management metadata is protected from direct access by applications.

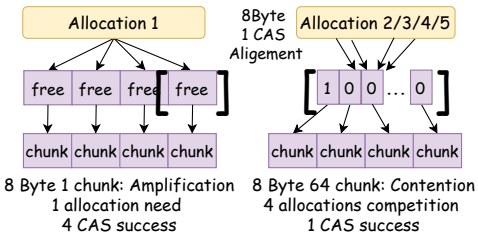


Figure 4: One-sided RDMA Based Chunk Allocation Challenge. Left: a simple chunk array may result in network amplification. Right: using bitmaps, retry time arises when multiple processes contend for the same 8B bitmap entry.

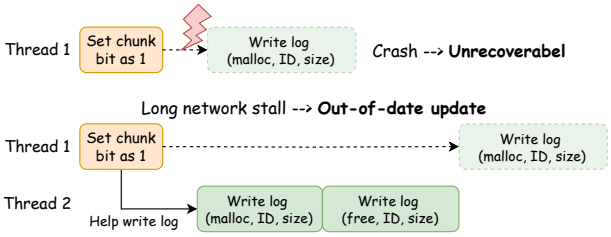


Figure 5: Two Examples of Crash Inconsistency: log lost during crash failure, and outdated update in a fail-slow failure.

Enabling Efficient, Highly Scalable Free Remote Chunk Allocation. As discussed in Sec. 2.2, FineMem uses one-sided RDMA access for remote allocations to prevent the memory node CPUs from becoming bottlenecks.

However, this one-sided approach requires a carefully designed metadata structure to minimize network round trips during chunk allocations. For instance, managing free chunks with an uncompacted free chunk array, like CXL-SHM [62], as shown in Fig. 4 (left), could require checking every chunk in the address space - each check involving a round trip - leading to unpredictable and inefficient allocation times, especially for large allocation size on multiple chunks. Alternatively, using compact bitmaps (e.g., representing 64 chunks per 8 bytes) reduces the number of round trips by allowing more chunks to be searched at once. However, this approach can still lead to frequent retries, as multiple applications may target the same (8 bytes) bitmap for allocation. In such cases, each application uses lock-free operations like Compare-And-Swap (CAS) to allocate, but only one application succeeds in grabbing a chunk, while others must retry, each retry involving another round trip (Fig. 4 right).

To address this, FineMem employs a two-layer bitmap tree structure, combining the efficiency of compact bitmaps with an additional first layer to accelerate large size allocations and reduce retry attempts. Applications read the first-layer bitmaps to allocate large memory sizes ($\geq$128KB, aligned to a power of 2) with a single CAS, or to identify which *chunk groups* are empty, full, or experiencing contention (indicating multiple concurrent allocation attempts). They then retrieve the second-layer bitmap for a selected chunk group - typically one that is neither full nor contended - to allocate small memory sizes (4KB-64KB) with free chunks. If an application detects high contention on a bitmap, it can redirect its allocations to a less contended chunk group (Sec. 4.1).

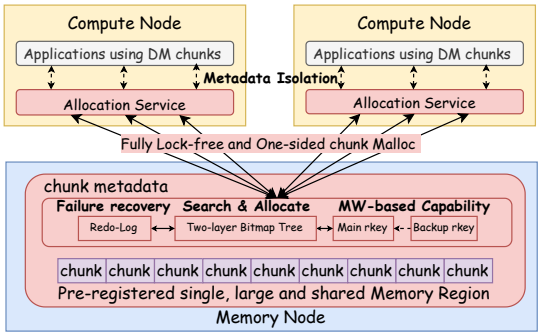


Figure 6: Remote Memory Management with FineMem.

Maintaining Metadata Consistency During Compute Node Failures. Similar to coarse-grained allocation systems, FineMem must track which memory chunks are allocated to which applications to prevent memory leaks in compute node failures. To accomplish this, FineMem uses redo logging on memory nodes, written by compute nodes via one-sided RDMA, ensuring that allocations are correctly recorded. However, there are consistency issues between the allocation bit set and the log write. For example, as shown in Fig. 5, an inconsistency during a crash failure might cause a chunk to become unrecoverable (because it has just been marked as allocated in the bitmap but lost the detailed allocation information). Even in scenarios where other threads can assist the failed thread in writing the log, fail-slow failures can still occur, where the failed thread ultimately writes an outdated log that other threads have already written.

To address this, FineMem carefully integrates a compact temporary log into the limited 64-bit bitmap, allowing each allocation success to be temporarily logged, with any thread able to update (flush) this temporary log into a persistent, full log. FineMem also defines a timestamp in the temporary log to prevent out-of-date consistency issues (Sec. 4.2). Additionally, FineMem resolves inconsistency problems between the two-layer bitmaps, based on the correctness of the commit point.

3.2 System Architecture

As shown in Fig. 6, FineMem integrates components from both memory nodes and compute nodes.

On the memory node, FineMem builds on core data structures that tracks available memory, support scalable allocations, manage capabilities, and log allocations. On the compute node, FineMem runs an allocation service on each node. It pre-mmaps the entire remote memory pool at boot time, and handles (de)allocation requests from applications for remote memory chunks. Once an application allocates a remote memory chunk, it can use the chunk for fine-grained objects, key-value pairs, etc.

FineMem provides simple APIs, including malloc(size) and free(addr), for applications. When an application invokes

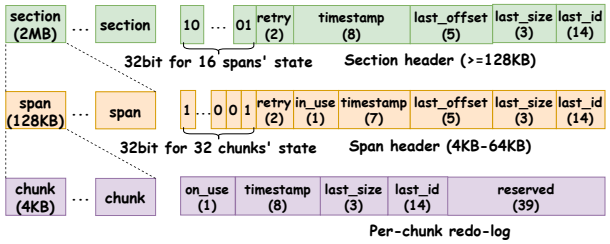


Figure 7: FineMem's Two-layer Bitmap Tree Structure with fine-grained allocation and contention control.

these APIs, per-core allocation service worker threads are woke-up to handle the protected remote allocations. Upon complete, the allocation service worker thread returns a DM address with an rkey to the application. FineMem can be ported to applications like object allocators [33], language runtimes [55], kernel swap systems [3], Key-Value stores [50], etc. FineMem supports flexible allocation granularities, starting from 4KB, and ensures memory is aligned to 4KB. Applications sharing the same FineMem-managed DM can use different allocation granularities (4KB blocks with number of power of 2).

4 FineMem Design

In this section, we detail FineMem's design for its metadata structures for concurrent, scalable allocation, logging for crash consistency, and per-comptue node allocation service.

4.1 Two-Layer Bitmaps For Concurrent, Scalable Chunk Allocation

FineMem addresses two primary remote chunk allocation challenges: 1) minimizing network round-trips during chunk searches, and 2) reducing allocation retries due to contention.

1) Two-Layer Bitmap Tree for Efficient Chunk Allocation. FineMem uses a bitmap-based free chunk index and manages the remote memory address space in two layers: sections and spans. As shown in Fig. 7, FineMem maintains bitmaps (and metadata) for both layers. It uses a 64-bit header for each of these metadata items to align with the RDMA CAS operation's work size requirement [5].

A section consists of 16 contiguous spans, with each span representing 128KB. The 32-bit bitmap for each section tracks fullness and contention (i.e. the level of concurrent allocations) of its spans, using two bits per span to indicate its status. Span statuses 00 and 11 represent where the span is entirely free or full. Span statuses 01 and 10 represent "in-use" states, where the span is partially allocated as smaller chunks.

Each span comprises 32 contiguous chunks, and a 32-bit free map to track the allocation status of these chunks. In addition to 32-bit free map of section/span headers, FineMem reserves the remaining 32 bits for contention control and logging. FineMem uses 2 bits to track the retry count relative to two thresholds (low and high), enabling contention detection. The rest bits for logging will be discussed later.

For different allocation sizes, FineMem accesses search metadata in (up to) three steps. It first searches the section headers to identify available spans. If the requested allocation size exceeds 128KB, FineMem directly modifies the set spans to 11 using a single CAS operation. Next, it reads the span header, marks the bits corresponding to the free chunks, and updates the span header with another CAS operation. Deallocation follows a similar process with one CAS to update the related bitmap. Allocations larger than one section require multiple CAS operations to modify contiguous sections.

To optimize performance, FineMem caches section and span metadata on each compute node to minimize access time, and batches multiple bitmap reads. FineMem uses a lazy update policy, updating the cache only when a CAS operation fails, indicating outdated cached metadata.

2) Contention Control to Reduce Retries. FineMem implements a simple yet effective back-pressure policy to limit contention across allocations. Each allocation header tracks the number of consecutive allocation failures. If the failure count exceeds the higher threshold (e.g. 10), the section/span is marked as contended in retry bits (and span's states in the section header).

Contention is detected by the allocations themselves. After a successful allocation in span, if the allocation process finds that the CAS failure count exceeds the threshold, it marks the section/span as 'contended (10)' to signal contention. The process will then select non-contended sections/spans, in the priority of normal (01) > empty (00) > contended (10), for future allocations. If a process using a contended (10) section/span detects that the failure count drops below the lower thread (e.g. 3), it resets the status to normal (01) .

In addition to the bitmap tree, FineMem maintains a per-chunk capability table and a per-chunk redo-log. Both are 8-bit aligned items in preserved linear arrays. One chunk's redo-log records information about its last allocation/free, including the allocation ID and timestamp, to facilitate recovery in case of compute node failures. The capability table stores per-chunk RDMA rkeys for access isolation.

4.2 Compute-Node-Failure Crash Consistency

FineMem implements a lock-free crash consistency mechanism by using the CAS update from the allocation as a commit point. It then flushes the log with timestamp validation and optimistically updates the remaining metadata, without need for CAS retries or additional consistency checks (Fig. 8).

1) A Compacted Commit Point. FineMem defines the moment an allocation successfully sets spans/chunks from free to used (or vice versa) through a CAS operation as a commit point. This commit point, along with a temporary redo-log and a timestamp, is compacted into a single CAS operation on the 64-bit section/span header.

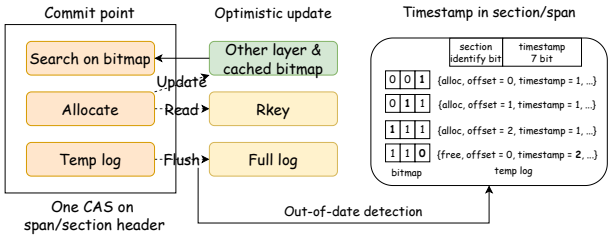


Figure 8: FineMem's Crash Consistency Design.

Given the limited space available in the 64-bit section/span header, FineMem splits the redo-log into two parts: a temporary redo-log stored inline with the header's CAS requests and a full redo-log. The temporary redo-log holds minimal information about the last allocation/deallocation of any chunks belong to this span/section. It includes the last allocated chunk offset (5 bits, which can infer the last request is malloc or free by checking bitmap), size (3 bits are sufficient with powers of 2 size alignment), timestamp (7 bits), and user ID (14 bits).

To manage the sequence of allocations and detect out-of-date log flushing, FineMem includes a timestamp in the redo log. With a 7-bit timestamp, each timestamp will overflow when it reaches 127. Therefore, FineMem can only compare two log's time sequence inner 64 timestamp increasing, which corresponds to approximately 1ms with 20us per allocation. FineMem tries to support longer out-of-date detection by slowing down the timestamp incrementing. Since different chunks update in their own redo-log entries, back-to-back allocations within a single section/span can share the same timestamp. Thus, FineMem's timestamp only increases when the previous operation was an allocation and the current operation is a deallocation (or vice versa), as shown in Fig .8.

2) Metadata Inconsistency Detection and Recovery. After a successful commit point, the allocation thread, or any other thread attempting to use the same section/span, will try to flush the temporary redo-log into the chunk's full redo-log. It checks the timestamp of the temporary redo-log against the full log using a CAS request to decide whether to flush the temporary log or simply discard it. When the thread finds that the chunk's temporary log and full log are of different layers (span and section), it further checks the span's in_use bit as the latest state to prevent outdated logs.

Additionally, FineMem uses an optimistic one-sided RDMA-write instead of CAS success to update the section header after a span commit point, and leverages inconsistency checking in the allocation path. Because a process can acquire the latest commit points in allocation path, then check and fix inconsistency between two layers.

FineMem leverages the consistency of its redo-log to enable non-blocking crash recovery for rebuilding memory allocation status. The recovery process begins by scanning the section and span metadata to flush the temporary redo-log. It then collects or reclaims the allocated chunks from the crashed node by scanning the full redo-log. To detect failures, FineMem employs a user-space monitor on the memory node, which sends heartbeats every second.

To summarize, FineMem is designed to handle allocation/deallocation in a way that ensures crash consistency, using lock-free, one-sided RDMA verbs.

Table 1: MW and MR Registration Overhead Comparison.

Average Latency per 4MB (us)	MW	MR
1*100GB Generation	1.33	456.1
25K*4MB Generation	1.34	485.5
25K*4MB Invalidate	1.33	21.9
25K*4MB Regeneration	2.37	46.5

4.3 Allocation Service For Isolation/Protection

FineMem uses memory window-based capabilities for isolating memory accesses and a per-compute node service layer to safeguard allocation metadata. FineMem addresses two key challenges: 1) eliminating CPU involvement in MW binding on the memory node during allocations, and 2) efficient communication between the user applications and allocation service layer on the compute node.

1) Memory Node Component for Chunk Protection. FineMem uses Memory Windows (MWs) [13, 47] in RDMA for generating and regenerating rkeys. In our evaluation using a ConnectX-6 RDMA NIC, MWs support up to 16 million entries (64GB for 4KB chunks), and generating a MW rkey for a 4 MB chunk takes only 1 microsecond - compared to hundreds of microseconds for MR registration, as shown in Table 1. FineMem overcomes the single-NIC 64GB limitation by generating multiple virtual functions (up to 128 VFs on a single ConnectX-6 NIC), with each VF assigned 16M memory windows. To minimize on-chip resource consumption, FineMem uses a single memory region (MR) with pre-mapped hugepages [14, 40].

MWs are pre-generated for each chunk/span/section after the MR is initialized, and the corresponding rkeys (main rkeys) are stored in the capability table. To minimize the overhead of rkey regeneration (for deallocation), the memory node pre-generates backup rkeys for each chunk/span/section. A pair of main rkey and backup rkey together requires 8 bytes, allowing the deallocation process to use a one-sided CAS operation to replace the main rkey with the backup rkey as part of the critical-path regeneration. This ensures there is no critical path dependency on the memory node. The memory node then asynchronously scans the capability table, invalidates old rkeys, and generates new ones.

2) Compute Node Service Layer for Metadata Protection. FineMem introduces an allocation service on compute nodes to manage privileged operations related to metadata. On FineMem-managed compute nodes, only the pre-initialized allocation service has the authority to (de)allocate remote memory. All processes' requests are handled and executed by this allocation service.

Like many software virtualization techniques, FineMem's service process faces the challenge of API interception performance costs. To address concurrency, FineMem assigns a worker thread to each CPU core, which is blocked by shared semaphores. Applications write their allocation requests to shared memory, wake up the allocation worker thread, and wait for the allocation to complete. In our evaluation, this inter-process communication (IPC) adds approximately 2-10µs of overhead—a reasonable trade-off compared to the significantly higher latency (10× to 100×) of fine-grained memory allocation directly on the memory node.

5 Porting Systems to FineMem

We have integrated FineMem with three type of systems: a user space memory allocation library that can allocate/free from remote memory (FineMem-User), a DM-based distributed Key-Value store (FineMem-KV), and a DM swap system (FineMem-Swap). These applications use FineMem's malloc(size) and free(addr) functions to allocate/free remote memory (flexible sizes, starting from 4KB chunk).

1) FineMem-User. FineMem-User is a memory allocation library that allows applications allocate objects (e.g., bytes) from remote memory. FineMem-User bases on mimalloc [33] but manages space in DM (using FineMem). For object-size allocations, FineMem-User handles them entirely on the compute node. On slow-path when there are no local reserved chunks, it requests/frees chunks from FineMem and subdivides them into smaller slabs, similar to how (local) allocators operate (like tcmalloc/jemalloc/mimalloc [18, 33, 52]).

2) FineMem-KV. Key-Value systems are popular applications for DM, and their remote memory management can be replaced with FineMem's allocation system. We re-implemented the slow path of FUSSE's block allocation [50] using FineMem, while keeping other components unchanged. This modification required only 300 loc. Furthermore, unlike FUSEE which requires memory node block servers, FineMem-KV becomes a completely one-sided KV system.

3) FineMem-Swap. FineMem can also be used to optimize DM memory utilization in swap-based [3] systems. FineMem-Swap includes three major components: i) an allocator in the kernel that provides APIs for allocating/deallocating remote pages (*page manager*); ii) a mapping between local swap offsets and DM memory addresses (*remapper*), enabling on-demand binding of remote pages to swap entries; and iii) DM allocation on the page's swap-out path and deallocation after swap entries invalidation.

6 Evaluation

We characterize FineMem performance and overheads, and evaluate how well applications running with FineMem compare to state-of-the-art systems. Specifically, we answer the following questions:

- How does FineMem reduce allocation latency compared to state-of-the-art systems? What is the overhead associated with isolation and crash recovery support?
- How do applications perform when using FineMem for object allocation in their critical path? How does FineMem behave when switching applications with different allocation size distributions?
- What is the end-to-end performance of popular DM applications (i.e., Key-Value stores and swapping system), when using FineMem?

6.1 Experiment Setup

Testbed. We use a Cloudlab [16] cluster with 16 compute nodes and 1 memory node, which is a reasonable setting in datacenter memory pool [35]. Both compute and memory nodes use Intel Xeon 8360Y CPUs, the Mellanox ConnectX-6 100Gb NIC and have 256GiB of memory. Applications run on compute nodes and can use up to 32 cores, while we limit memory nodes to only use single core. On each node, we run Ubuntu 22.04 with 5.15 Linux kernel, and the Mellanox OFED 5.8 RDMA driver.

Implementations and Baselines. We implemented FineMem in C++ (8.5k LOC). We also implemented a mimalloc-based user-space allocation library, the FineMem-User (1.5k LOC for adding DM fine-grained allocation) that uses FineMem DM allocator interfaces, and ported FastSwap [29] to use FineMem interfaces with the FineMem-Swap (0.7k LOC for adding new remapper and chunk manager components in the kernel module).

We compared FineMem to three state-of-art baselines: Premmap-One-sided (from CXL-SHM [62], search based on free array), Premmap-RPC (from FUSEE [50] and Patronus [60], search based on memory node side allocator memkind [9]), and OnDemand-RPC (fine-grained allocation with registration). For fair comparisons, we ported these approaches within the same artifact as FineMem. For both Premmap-One-sided and Premmap-RPC, we statically premapped 200GB of memory to avoid runtime MR registration.

6.2 Remote Allocation Performance

We characterize FineMem's remote allocation performance from three perspectives. First, we use allocation microbenchmarks to evaluate FineMem's performance across different allocation granularities and scalability. Second, we analyze FineMem's allocation latency, breaking it down into the various factors introduced by its design choices. Finally, we examine the memory-node-side CPU and memory overhead, as well as FineMem's crash recovery capabilities.

Microbenchmark. We utilize a Redis-based KV store microbenchmark derived from the memory efficiency unit test in

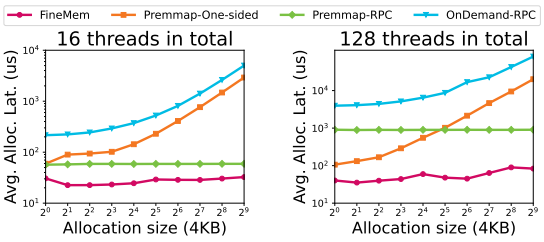


Figure 9: Allocation Performance vs. Varying Request Sizes.

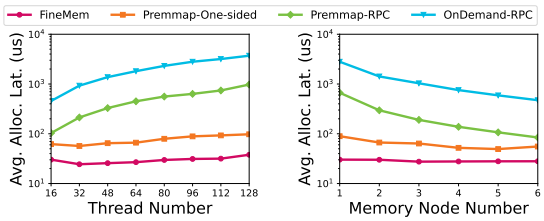


Figure 10: Allocation Performance vs. Varying Client Threads (left) and Varying Number of Memory Nodes (right). 4KB allocation granularity. Allocations distributed across multiple memory nodes using a round-robin approach.

the Redis [44] test suite. This benchmark performs memory allocations and deallocations ranging from 4KB to 2MB on remote memory. The workload initially allocates 140GB of memory, then randomly deallocates 50% and allocates another 50%, repeating this pattern. Fig.9 and 10, along with Table 2, demonstrate FineMem's performance advantages over baseline approaches across various allocation granularities, levels of client parallelism, and memory node configurations.

As shown in Fig. 10 (left), FineMem outperforms the RPC-based approaches (both OnDemand-RPC and Premmap-RPC), particularly under high client parallelism, where the RPC-based methods quickly become bottlenecked by the limited compute power of memory nodes. With a 4KB allocation granularity, we observe that the RPC-based remote allocation performance does not scale after 16 client threads. In contrast, FineMem maintains consistently low latency due to its one-sided RDMA design.

FineMem outperforms existing one-sided approaches (Premmap-One-sided, without runtime MR registration) across varying allocation granularities and levels of client parallelism, as shown in Fig. 9 and 10. This performance advantage is attributed to FineMem's optimized metadata design and contention control mechanism, which significantly reduce the number of per allocation RDMA CAS operations compared to the simple direct array design employed by existing one-sided approaches.

In the microbenchmark workloads with fragmented remote memory, the direct array metadata often requires multiple CAS operations to search for available chunks, for a 4KB allocation. The performance gap becomes even more pronounced for large allocations (Fig. 9), where each chunk in a large request requires separate CAS operations.

As shown in Table 2, the Premmap-One-sided approach

Table 2: Average/Max number of CAS retry statistics for allocation and average/P99 allocation latency. 4KB allocation granularity, 512 client threads in total.

Average / Tail	FineMem	Premmap-One-sided
Latency (us)	43.215 / 79.347	763.01 / 16143.516
CAS Retry Times	1.333 / 142	45.108 / 20637

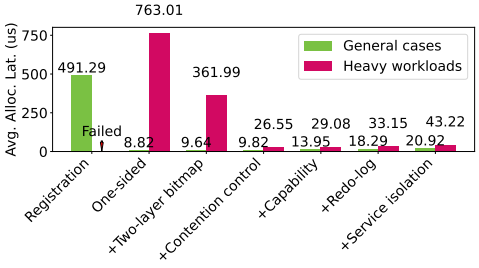


Figure 11: Factor Analysis.

can require an average of 45 CAS operations per allocation, whereas FineMem requires no more than two CAS operations on average. In Fig. 11's heavy workloads, we further drill-down the effects of FineMem's two-layer bitmap design and contention control mechanism. The two-layer bitmap reduces allocation latency by approximately 52.5% compared to the array-based approach, while the contention control mechanism contributes an additional 44% reduction by minimizing CAS retries caused by heavy contention. Combined, these two designs reduce allocation latency by approximately 96.5%, demonstrating the effectiveness of FineMem's design.

Together, these two components reduce allocation latency by approximately 96.5% compared to the Premmap-One-sided approach.

Factor Analysis. We evaluated the latency factors across different design components under both general cases (single-thread allocation) and heavy workloads (the microbenchmark), as shown in Fig. 11.

FineMem demonstrates stable latency across both general and heavy workloads. Although its chunk search mechanism involves two layers, the bitmaps are cached locally, resulting in only a slight increase in average latency compared to the Premmap-One-sided method in general cases. The rkey capability introduces an additional RDMA read to retrieve the key, and the redo-log requires one RDMA CAS to flush, with each step adding approximately 5µs. Additionally, the service layer contributes an additional 2-10µs due to IPC overhead.

Overall, FineMem reduces allocation latency by approximately 95% compared to traditional registration-based fine-grained methods. The overhead introduced by isolation and redo-log management is minimal, accounting for only 2.5% of the registration cost.

Overhead on Memory Nodes. FineMem runs two services on each memory node: the rkey regeneration thread and the recovery thread, co-running on a single core. The rkey regeneration thread scans each 100ms interval, spending 15ms for 100GB memory space (25 million 4KB chunks), and resulting

in approximately 15% of one CPU usage. The regeneration speed is fast, as mentioned earlier, with 1 million memory windows regenerated per second. In our evaluation, with a backup rkey to hide the scan and regeneration time, the peak free latency remains below 1ms. The recovery detection thread wakes up every second, which has negligible overhead. Furthermore, FineMem stores external data structures at memory nodes, consuming up to 0.4% of memory (mainly caused by an 8B rkey and an 8B redo-log for each 4KB chunk), which amounts to about 400MB for every 100GB of memory.

Recovery Capability. We show FineMem's recovery ability in out-of-date detection, recovery speed and recovery influence. 1) We ran the microbench at 1GB memory space (a narrow space with frequently malloc/free) and find that FineMem's timestamp overflow happens after about 50K memory allocation requests. As a result, update delayed less than 1s (each allocation 20us) can be detected by FineMem's 7-bit timestamp, which is enough to co-work with normal RDMA timeout settings. 2) the recovery latency is highly related to memory window rebinding time, and the recovery speed is similar to memory window regeneration speed, about 1M allocation entries (chunk/span/section) per second. Through our evaluation, all applications have no more than 100K allocation entries, which can be reclaimed in lower than 100ms. 3) The recovery process only have contention possibilities at span/section's bitmaps and the influence is similar to an external thread's allocation competition, which can be handled by FineMem's concurrency design with minimal influence.

In summary, even under heavy workloads, FineMem delivers scalable, low-latency, and predictable fine-grained memory allocation. This enables DM system achieves high memory utilization while maintaining strong performance.

6.3 Application Case Studies

Application's Allocation Critical-Path. We first use the FineMem-User to demonstrate the end-to-end application performance improvement on the allocation critical path by running FineMem-User on object-grained benchmarks.

1) ThreadTest [6] is a widely used allocator benchmark, creating a series of threads and repeatedly malloc/free objects. We set 256B as small work-set (the median size of in-memory KV [61]) and 4KB (the basic OS page size) as large work-set; 2) Shbench [39] is a stress test where some of the objects are freed in a usual LIFO order, but others are freed in reverse order. We set 8Byte-1KB as small work-set, and 256Byte-4KB as large work-set; 3) Larson [30] simulates a server workload using 100 separate threads which each allocate and free many objects but leave some objects to be freed by other threads. Work-set the same as Shbench. We set 80GB memory for the premmap baselines, which is the peak memory usage of Shbench-large. We ran all benchmarks with 16 compute node, each node has 32 threads.

The results, as shown in Fig. 12, highlight the performance

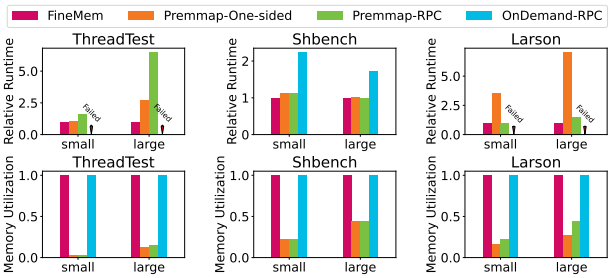


Figure 12: FineMem-User's Performance and Memory Utilization on Allocation Benchmarks.

and utilization trade-offs faced by state-of-the-art approaches:

Premmap memory waste. Premmap-One-sided and Premmap-RPC exhibit low memory utilization, even on the Shbench-large. Since their memory cannot be fully shared or utilized, the average memory utilization remains below 50%.

On-demand registration cost. OnDemand-RPC suffers from poor performance due to runtime memory registration. They even cannot finish some of the benchmark (RDMA timeout).

Allocation performance issues over network. RPC baselines encounter a high concurrency bottleneck when running ThreadTest allocation benchmarks. Meanwhile, the one-sided baseline performs poorly on the Larson benchmark, where frequent thread creation and destruction in each iteration cause new threads with Premmap-One-sided method to repeatedly start searches from the first chunk of the array.

FineMem can fully use the shared memory pool's free chunks, the same as fine-grained allocation with registration, improving memory utilization of state-of-arts with static premmap by 2.25× to 2.8×. And FineMem achieves the best performance by effectively fine-grained memory allocation in DM. It mitigates the scalability issues seen in RPC-based methods while providing fast and stable searches through its two-layer bitmap design.

Applications With Mixed Size Allocation Requests. To evaluate FineMem with applications that issue mixed-size allocations, we capture slow-path memory allocation system calls from various allocator and benchmark combinations.

We use the collected allocation system call sequences as requests to DM, preserving both the arrival time patterns and allocation size distributions. We selected four modern allocators: jemalloc [18], mimalloc [33], tcmalloc [52], and ptmalloc [19]. Benchmarks are chosen from the widely-used mimalloc-bench suite [12], including Shbench [39], Larson [30], Rptest [27], and Lean [1].

The size distributions of allocator and benchmark combinations, as shown in Fig. 13, vary significantly. Jemalloc and ptmalloc primarily use 4KB as the basic allocation size, tcmalloc primarily uses 2MB hugepages, and mimalloc has a median size of 64KB for small object allocation. These differences create diverse workloads with mixed allocation sizes for evaluation. To ensure trace randomness between nodes, we collect traces separately for each node and then execute them across 16 nodes, each running 8 threads.

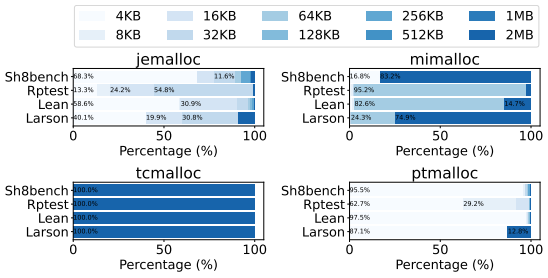


Figure 13: Mixed Size Allocation Traces' Distribution.

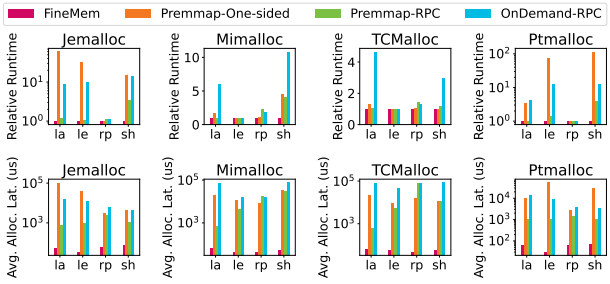


Figure 14: Performance on Mixed Size Allocation Traces.

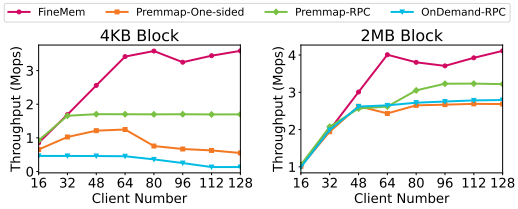


Figure 15: FineMem-KV's Throughput on YCSB-A Workload (search:update=50:50).

As shown in Fig. 14, under high concurrency and mixed-size distribution, both one-sided and RPC methods experience high average latency. Besides the CPU bottleneck of RPC methods, Premmap-One-sided performs worse at jemalloc and ptmalloc. Because there are frequent 4KB chunk allocation competing with 2MB allocations, Premmap-One-sided method is hard to find an available 512 contiguous chunks in this fragmented chunk space.

In contrast, FineMem's two-layer bitmap design effectively mitigates competition between different allocation sizes and accelerates the search for power-of-2-aligned free space. It consistently maintains latency below 100µs, with minimal impact on execution time, demonstrating FineMem's stability under high-concurrency, mixed-size allocation workloads.

KV System. We have conducted a comparative performance analysis of FineMem-KV against different baselines using standard YCSB workloads [11] (A, B, C, D, and a 100% update scenario). For these tests, we set the chunk size with 4KB (memory efficiency) and 2MB (performs better). Each Key-Value (KV) pair is 512 bytes in size.

As depicted in Fig. 15, under the YCSB-A workload, which is update-intensive with approximately 50% update operations, FineMem-KV demonstrated a bandwidth improvement of about 27%-110% over the best case of Premmap-RPC. However, in the read-intensive workloads of YCSB-B, C, and D, the performance gains were less pronounced, so we do not present those results here. FineMem's optimizations are particularly effective in enhancing FUSEE's update operations, likely due to the nature of out-of-place updates, which involve allocations for new subblocks followed by free of old ones.

With FineMem's low latency for fine-grained allocation, using smaller block sizes like 4KB becomes more attractive. It can achieve high bandiwidth similar to 2MB blocks while also reducing internal fragmentation, resulting in a 45% reduction in memory cost compared to the 2MB block setting, as shown in motivation section Fig. 2(a). What's more, when testing with larger block sizes, high allocation latency still negatively impacts throughput, as shown in Fig. 15 (right), highlighting the performance advantages of FineMem.

In summary, with a focus on optimizing out-of-place update operations, a KV system using FineMem can resolve the dilemma between efficiency and overhead, achieving both high throughput and reduced memory fragmentation through fine-grained allocation.

Swap System. This evaluation focus on comparing memory efficiency of static pre-mapped swap systems (e.g., FastSwap) and the fine-grained dynamic swap system (FineMem-Swap). Swap systems ran on a 7-machine testbed: 5 compute nodes with 80 GB of local memory and 2 memory nodes providing a total of 160 GB of shared DM. For FastSwap, each compute node had 32 GB exclusive memory on the DM.

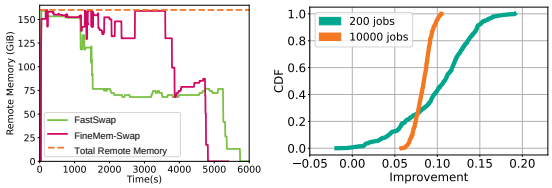
Three widely deployed applications are slected: XGBoost [10] a regularized gradient boosting framework; Snappy [20], Google's (de)compression library, used to compress enwiki articles [58]; and Redis [44], a popular in-memory database, subjected to YCSB [11] workloads. We inherited the DM-aware scheduler and simulator used by FastSwap.

We evaluated DM swap systems by placing 200 jobs with a workload ratio of XGBoost:Snappy:Redis=2:2:1 on the testbed. Compute nodes make best effort to use local memory and swap data to DM when local memory is insufficient. During system running, we collected placement data from FastSwap, and several representative states are shown in Table 3. After the 2000th second, this node had more than 50% free space in the reserved DM area, but the job queues were filled with Snappy and Redis jobs that could not be placed. As a result, FastSwap utilized only 41.39% of the total remote memory, as shown in Fig. 16 (a). In contrast, FineMem-Swap allows clients to share the DM with fine-grained allocation, resulting in an average memory utilization of 74.06%. This higher memory usage in FineMem-Swap led to a 17.71% improvement in job throughput.

To further investigate the throughput improvement in different workloads, we randomly generated 500 small workloads (each consisting of 200 jobs across 5 compute nodes sharing 160GB of DM) and 500 large workloads (each with 10,000 jobs across 40 compute nodes sharing 1280GB of

Table 3: Exclusive Fragmentation in One Compute Node.

Time(s)	1000	2000	3000	4000
Local Free Space(GB)	0	0	0	0
DM Free space(GB)	2.14	16.9	18.84	18.84
Waiting List	Snappy(33.2GB) & Redis(31GB)			



(a) Remote Memory Usage. (b) Jobs Throughput Improvement

Figure 16: FineMem-Swap's Remote Memory Utilization and Job Throughput.

DM), with varying application ratios. The CDF of FineMem-Swap's throughput improvement over FastSwap is shown in Fig. 16(b). FineMem-swap outperforms FastSwap in throughput across both small and large workloads, achieving an average throughput improvement of 8.38% to 10.69% . Therefore, FineMem improves job throughput by overcoming the static pre-mapped memory fragmentation.

Overall, with support for fine-grained, on-demand allocation in the swap system, FineMem-Swap improves job throughput compared to static pre-mapped swap systems.

7 Discussion and Limitations

While FineMem enables efficient fine-grained remote memory allocations, it does have certain limitations that we believe are worth exploring in future work.

First, FineMem primarily focuses on scalability, isolation, and memory efficiency in DM management, along with support for compute node crash consistency. It does not address other critical issues such as cache coherence [31] or memory duplication across multiple memory nodes [32, 63]. Integrating FineMem with solutions to these aspects of DM memory management is a potential area for future research.

Secondly, FineMem pre-registers the entire remote memory, requiring applications to first communicate with a designated server process on the memory nodes to establish an RDMA connection. Moreover, FineMem did not optimize data-path access performance; therefore, its RDMA access performance represents a standard baseline without additional overhead or specialized optimizations.

Lastly, the allocation services in FineMem, which uses IPC to intercept remote memory allocations, introduces overheads on the allocation path. While we believe that these overheads are justified by the security and isolation benefits provided, we are actively working on optimizing the IPC process. Future optimizations may include approaches such as kernel acceleration to further reduce these overheads.

8 Related Work

Transparent disaggregation. To enable applications to use DM without modification, many studies leverage the swap and paging mechanisms in kernel to access and manage DM. For instance, Leap [2] improves swap system's page prefetcher to better adapt to DM; Canvas [56] focuses on optimizing swap entry allocation. However, all these approaches use simple pre-registration strategies, leading to significant memory waste. Apart from these, Memliner [55] optimizes the garbage collection strategy in high-level languages to adapt to the swap system, which are orthogonal to FineMem.

Native disaggregation. Certain scenarios, such as KV-storage and distributed shared memory, allow programs to utilize DM through explicit APIs. In the realm of DM-native applications, from FaRM [14] to CoRM [51], many employ RPC-based allocators, which do not perform well under high concurrency.

Memory allocator. User-space allocators [18, 24, 34, 37, 52] are widely used in software systems to enhance efficiency and performance in memory management. FineMem learns from their design philosophy and further considers the characteristics in DM architectures. LLFree [59] is a high concurrency frame allocator in kernel, which also maintains a two-layer metadata but prioritizes exclusive entries. While FineMem adopts a sharable entry to optimize memory space efficiency.

Disaggregation memory isolation. For CXL-based DM, fabric manager offers machine-level isolation by maintaining permission tables on hardware [21, 35]. For RDMA-based DM, MIND [31] ues programmable switch and Clio [23] uses customized FPGA to replace the heavy MR-based isolation of RDMA NIC. To achieve both performance and easy promotion, FineMem choses MW mechanism that comes with commercially productive RDMA NIC. Patronus [60] also uses MW for memory protection within a single application, and it relies on RPC for memory allocation, which performed poorly in our evaluation.

9 Conclusion

We present FineMem, an efficient fine-grained remote memory management system using one-sided RDMA. FineMem removes memory region registration costs for applications while ensuring isolation and protection through Memory Windows and a per-compute node allocation service. FineMem features scalable and efficient metadata designs for remote chunk allocation and crash consistency mechanisms to handle compute node failures. FineMem reduces remote memory allocation latency by up to 95% compared to state-of-the-arts, enabling applications to achieve low memory waste with minimal performance overhead.

References

[1] Lean. https://github.com/leanprover/lean3, 2024. [Online; accessed 1-December-2024].

[2] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *USENIX ATC 20*, pages 843–857, 2020.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Eurosys 20*, pages 1–16, 2020.

[4] Dotan Barak. *ibv_reg_mr, ibv_reg_mr_iova, ibv_dereg_mr - register or deregister a memory region (MR)*, 2006. Linux Programmer's Manual.

[5] Dotan Barak, Majd Dibbiny, and Yishai Hadas. *ibv_post_send(3) - post a list of work requests (WRs) to a send queue*. RDMA Core Userspace Libraries and Daemons, 10 2006. Accessed: 2024-01-12.

[6] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[8] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[9] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[10] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *SIGKDD 16*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[11] Brian F. Cooper. Yahoo! cloud serving benchmark (ycsb). Accessed: 2022-02-22.

[12] Daanx. mimalloc-bench: Benchmark suite to test various memory allocators. https://github.com/daanx/mimalloc-bench, 2024. Accessed: 2024-01-12.

[13] Majd Dibbiny and Yishai Hadas. *ibv_bind_mw - post a request to bind a type 1 memory window to a memory region*, 2016. Linux Programmer's Manual.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI 14*, pages 401–414, 2019.

[15] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231. IEEE, 2010.

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

[17] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *ASPLOS*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[19] Wolfram Gloger. ptmalloc. http://www.malloc.de/en/, 2024. [Online; accessed 1-December-2024].

[20] Google. Snappy: A fast compressor/decompressor. Accessed: 2022-02-22.

[21] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *USENIX ATC 22*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *NSDI 17*, pages 649–667, 2017.

[23] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS 21*, pages 417–433, 2021.

[24] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *OSDI 21*, pages 257–273. USENIX Association, July 2021.

[25] Intel. 100gbe intel® ethernet network adapter e810. https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters/e810-network-adapters/products.html, 2024. [Online; accessed 1-December-2024].

[26] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS:Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *USENIX ATC 23*, pages 585–600, 2023.

[27] Mattias Jansson. rptest. https://github.com/mjansson/rpmalloc-benchmark, 2024. [Online; accessed 1-December-2024].

[28] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based virtual RDMA networking for containerized clouds. In *NSDI 19*, pages 113–126, Boston, MA, February 2019. USENIX Association.

[29] Yash Lala. Fastswap-linux-5.16.16. https://github.com/yashlala/fastswap-linux-5.16.16, 2022.

[30] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, page 176–185, New York, NY, USA, 1998. Association for Computing Machinery.

[31] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *SOSP 21*, pages 488–504, 2021.

[32] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *FAST 22*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[33] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Report MSR-TR-2019-18 ,, Microsoft, June 2019.

[34] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*, pages 244–265. Springer, 2019.

[35] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. SMART: A High-Performance adaptive radix tree for disaggregated memory. In *OSDI 23*. USENIX Association, 2023.

[37] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *ASPLOS 20*, ASPLOS '20, page 541–556, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *ASPLOS*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[39] MicroQuill. Smartheap technical specification. http://www.microquill.com/smartheap/sh_tspec.htm, 2024. [Online; accessed 1-December-2024].

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

[41] Nvidia. Connectx nics. https://www.nvidia.com/en-sg/networking/ethernet-adapters/, 2024. [Online; accessed 1-December-2024].

[42] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: compacting memory management for c/c++ applications. In *PLDI 2019*, PLDI 2019, page 333–346, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim,

and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *NSDI 23*, pages 181–198, Boston, MA, April 2023. USENIX Association.

[44] Redis. https://redis.io/. Accessed: 1-December-2024.

[45] Remote Direct Memory Access (RDMA) - Wikipedia. https://en.wikipedia.org/wiki/Remote_direct_memory_access#:~:text=In%20computing%2C%20remote%20direct%20memory,in%20massively%20parallel%20computer%20clusters. Accessed: 1-December-2024.

[46] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 01 1977.

[47] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *USENIX Security 21*, pages 4277–4292, 2021.

[48] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM:High-Performance,Application-Integrated far memory. In *OSDI 20*, pages 315–332, 2020.

[49] Huijun Shen, Guo Chen, Bojie Li, Xingtong Lin, Xingyu Zhang, Xizheng Wang, Amit Geron, Shamir Rabinovitch, Haifeng Lin, Han Ruan, Lijun Li, Jingbin Zhou, and Kun Tan. Np-rdma: Using commodity rdma without pinning memory, 2023.

[50] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. FUSEE: A fully Memory-DisaggregatedKey-Value store. In *FAST 23*, pages 81–98, 2023.

[51] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma, 2021.

[52] TCMalloc - Google. https://github.com/google/tcmalloc. Accessed: 1-December-2024.

[53] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX ATC 20*, USENIX ATC'20, USA, 2020. USENIX Association.

[54] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *SOSP 17*, pages 306–324, 2017.

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *OSDI 22*, pages 35–53, 2022.

[56] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *NSDI 23*, pages 161–179, 2023.

[57] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *OSDI 20*, pages 117–135, 2020.

[58] Wikimedia Foundation. Wikimedia downloads. Accessed: 2022-02-22.

[59] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, Daniel Lohmann, Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. Llfree: Scalable and optionally-persistent page-frame allocation. In *USENIX ATC 23*, volume 65. USENIX Association, 2023.

[60] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. Patronus: High-Performance and protective remote memory. In *FAST 23*, pages 315–330, Santa Clara, CA, February 2023. USENIX Association.

[61] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI 20*, pages 191–208, 2020.

[62] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for cxl-based distributed shared memory. In *SOSP 23*, pages 658–674, 2023.

[63] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *OSDI 22*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

[64] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. Characterizing a memory allocator at warehouse scale. In *ASPLOS*, ASPLOS '24, page 192–206, New York, NY, USA, 2024. Association for Computing Machinery.

[65] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *USENIX ATC 21*, pages 15–29, 2021.